Rely-Guarantee Protocols for Safe Interference over Shared Memory

Filipe David Oliveira Militão

CMU-CS-15-146

December 2015

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213, USA

&

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica, Portugal

Thesis Committee:

Dr. Jonathan Aldrich[1], chair
Dr. Frank Pfenning[1]
Dr. Karl Crary[1]
Dr. Neelakantan Krishnaswami[3]

([1]Carnegie Mellon University, USA)
([3]University of Birmingham, UK)

Dr. Luís Caires[2], chair
Dr. António Ravara[2]
Dr. Vasco Vasconcelos[4]

([2]Universidade Nova de Lisboa, Portugal)
([4]Universidade de Lisboa, Portugal)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was partially sponsored by *Fundação para a Ciência e Tecnologia* (Portuguese Foundation for Science and Technology) through the Carnegie Mellon | Portugal Program under grant SFRH / BD / 33765 / 2009 and the Information and Communication Technology Institute at Carnegie Mellon University, INTERFACES NGN44-2009-2012, the Aeminium project SE38-2009-2012, CITI PEst-OE / EEI / UI0527 / 2011, FCT/MEC NOVA LINCS PEst UID/CEC/04516/2013, the U.S. National Science Foundation under grant #CCF-1116907 "Foundations of Permission-Based Object-Oriented Languages", and the U.S. Air Force Research Laboratory.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: programming languages; interference control; linearity; typestates; rely-guarantee protocols; concurrency; shared mutable state.

Abstract

Mutable state can be useful in certain algorithms, to structure programs, or for efficiency purposes. However, when shared mutable state is used in non-local or non-obvious ways, the interactions that can occur via aliases to that shared memory can be a source of program errors. Undisciplined uses of shared state may unsafely interfere with local reasoning as other aliases may interleave their changes to the shared state in unexpected ways. We propose a novel technique, *rely-guarantee protocols*, that structures the interactions between aliases and ensures that only safe interference is possible.

We present a linear type system outfitted with our novel sharing mechanism that enables controlled interference over shared mutable resources. Each alias is assigned separate, local roles encoded in a protocol abstraction that constrains how an alias can legally use that shared state. By following the spirit of rely-guarantee reasoning, our rely-guarantee protocols ensure that only safe interference can occur but still allow many interesting uses of shared state, such as going beyond invariant and monotonic usages.

This thesis describes the three core mechanisms that enable our type-based technique to work: 1) we show how a protocol models an alias's perspective on how the shared state evolves and constrains that alias's interactions with the shared state; 2) we show how protocols can be used while enforcing the agreed interference contract; and finally, 3) we show how to check that all local protocols to some shared state can be safely composed to ensure globally safe interference over that shared memory. The interference caused by shared state is rooted at how the uses of different aliases to that state may be interleaved (perhaps even in non-deterministic ways) at run-time. Therefore, our technique is mostly agnostic as to whether this interference was the result of alias interleaving caused by sequential or concurrent semantics. We show implementations of our technique in both settings, and highlight their differences. Because sharing is "first-class" (and not tied to a module), we show a polymorphic procedure that enables abstract compositions of protocols. Thus, protocols can be specialized or extended without requiring specific knowledge of the interference produce by other protocols to that state. We show that protocol composition can ensure safety even when considering abstracted protocols. We show that this core composition mechanism is sound, decidable (without the need for manual intervention), and provide an algorithm implementation.

Acknowledgments

I thank my advisors, Jonathan Aldrich and Luís Caires, for all the discussions, feedback, and technical assistance. I specially thank Jonathan for the tireless openness to discussion and healthy encouragement that enabled the shift in direction that led to this work. Although doing these "late" changes was quite challenging, I personally found your passion for research and exceptional professionalism to be both inspiring and instrumental in keeping me motivated to pursue this work. Thank you Jonathan!

This work is also the culmination of several iterations, refinements, and clarifications to address feedback from countless other sources (beyond my advisors). Attempting to list all individuals would invariably risk missing important sources, although I appreciate all feedback that improved and shaped the work into its final form. Still, I specially thank all members of the Plaid group at CMU and the PLASTIC group at FCT/UNL, the thesis committee members, and the many anonymous reviewers for all the comments and feedback on this work, in all its forms.

Finally, I thank friends and family for encouragement and support throughout all these years. I specially thank João Gomes, André DaSilva, Luísa Cochito, my sister (Susana Militão), my brother (Miguel Militão), and my parents (Luísa Oliveira and José Militão). And, naturally, Lury, Gotsky, Timmy "Timótio" Ti-Nó-Ni, Boris Fagundes-Pá, Fellini, and others for healthy distractions.

Contents

Chapter 1

Introduction

The ubiquitous use of computers is plagued by the weak correctness guarantees that today's software provides. In fact, most users have experienced some form of software error. These errors can have a wide range of consequences ranging from "mostly harmless" unavailability (such as the infamous "Blue Screen of Death" of old Microsoft Windows systems) to catastrophic failures (such as accidental deaths [3, 4], or unintended destruction of expensive hardware [46]). Yet, there also exists anecdotal evidence of highly reliable software. For instance, the development of *Graphing Calculator* was believed to have reached such a degree of software reliability that any run-time error was blamed on faulty hardware [9]. However, this informal belief was based on empirical evidence and the use of extensive testing and code reviews/audits. Ideally, building software with such a degree of reliability would be cheaply and quickly attainable by any programmer—but with proven, formal certainty of having reached the desired degree of reliability.

Experienced programmers are usually able to understand and organize their own code to mitigate the possibility of errors, at least to some degree. However, the increasing complexity of software, its extensible nature, and the varied programming experience of large teams creates challenges in building, debugging, and maintaining these systems. Furthermore, the push to exploit the performance gains of concurrent executions creates an additional vector for different components to interact in erroneous ways. Testing is dependent on the adequacy and the size of the samples that are used to inspect the behavior of the software. This means that, for the general problem, testing can be highly inadequate: testing may completely miss errors in the code, may require substantial computing power to ensure minimally adequate coverage, and the tests themselves can contain subtle mistakes that result in a false sense of correctness. Alternatively, formal reasoning aims to mitigate all these issues by reasoning about a program's code to ensure that the program was built correctly, long before the code is executed. Ultimately, this kind of "mechanical" reasoning aims to complement the programmer's own reasoning abilities to enable building correct code by construction (as opposed to relying on auxiliary procedures). Once the programmer explicitly and precisely states their intent and assumptions, the checker can then automatically verify that the program "makes sense" or identify flaws such as missing corner cases.

This thesis proposes a new technique to address a particular sub-problem of software reliability. More specifically, we aim to guarantee that imperative programs use shared mutable state correctly. We aim to detect or avoid errors related to *unsafe interference* in the use of mutable state that is

shared through several aliases. To this end, we build our technique within the framework of a *type system* that is able to rule out the presence of errors caused by unsafe interference without running a program (i.e. *statically*).

1.1 Approach Overview

We develop our technique in a higher-order imperative functional language. Although a similar level of expressiveness may be reached by only using the language's functional subset, mutable state can lead to more efficient code [27, 88], can enable programming idioms that improve code clarity, or even enable increased extensibility [6]. This makes mutable state convenient to use in certain algorithms and in structuring programs. Thus, we extend the core functional language with a small set of imperative constructs to enable safe and composable use of shared mutable state.

In traditional systems, composing mutable state usually implies encapsulating its use within functions. Since the effect a function has on its internal state is then not directly visible, functions are said to have "side-effects" that may affect their results. This is in stark contrast with the mathematical notion of a function whose result is clearly and solely determined by its arguments, independent of the function's past usages. To distinguish between the two kinds of functions, the latter are normally called *pure* functions as these functions do not depend on any stateful ("impure") resources to compute their result.

Our goal is to reason about the correctness of both state*less* and state*ful* programs within the same type framework. We aim to enable the safe composition of a small set of primitive, low-level constructs that operate on shared mutable state so that they can be safely combined to form safe higher-level abstractions. Furthermore, we aim to offer more precise reasoning than traditional type systems so as to avoid overly conservative checks and more closely match the programmer's intent. For instance, while a `File` abstraction can offer a large set of distinct operations (such as `open`, `write`, `close`, etc.), these operations may manipulate the `File`'s internal state in ways that are actually interdependent. Attempts to write to a closed `File` normally result is some program error state (such as an exception) that could be avoided if the compiler was able to enforce that only opened files could be written to—leading to clearer and more efficient code by making stronger, more precise, statements about how an operation can be used.

The prevalence of multi-threaded and multi-core systems leads to a natural desire to exploit the performance benefits that these systems enable. However, these system's performance gains come as a direct consequence of breaking down a program into smaller threads of control and allowing their concurrent execution. The looser constraints on interleaving threads enables processors to pick (potentially in non-deterministic ways) from a large set of different ways to schedule threads so as to better exploit the available computing resources, which includes enabling parallel executions that would not be possible in sequential programs. Concurrency adds an additional challenge to ensuring the correctness of programs as any reasoning or safety guarantee must be resilient to all possible ways in which threads may be interleaved at run-time.

When different threads share access to the same mutable state, thread interleaving may result in the "disruption" of the local assumptions that a thread has on what that shared mutable state currently contains. Indeed, the undisciplined use of shared state can cause *unsafe* interference by

Sequential Setting:

```
let x = new 0 in
    ...
    (λ f : unit → unit.
       x := 1;
       f ();
       !x + 1) ...
```

The type of the `f` function may encapsulate that function's effects on aliased mutable state. How may `f` *interfere* with the local assumptions about the contents of `x`, if the definition of the `f` function is allowed to share access to the `x` cell?

`λ.()` `λ.x := "oops"` `λ.delete x`

Possible definitions of `f` are shown above, each using the shared state differently. While they all obey the required type, each function interleaves different uses of the `x` cell producing different interference on that cell.

Concurrent Setting:

```
let x = new 0 in
    fork ... ;
    x := 1;
    !x + 1
```

In the concurrent setting, different alias interleaving may also be produced through thread interleaving. Therefore, forked threads may share access to mutable state and potentially modify or access that state (`x`) at arbitrary times.

`fork ()` `fork x := "oops"` `fork delete x`

Figure 1.1: Interference in the sequential and concurrent settings. Controlling and structuring this interference, while ensuring safety regardless of setting, is the central topic of this thesis.

unexpectedly mutating that shared state in ways that are not locally visible by simply looking at how a single thread is locally using that shared state. Thus, this possibility of unsafe interference brings additional challenges to ensure both the correctness and the safe composition of program components. Interestingly, interference may also occur in the sequential setting. Due to how state can be encapsulated, different components may indirectly interact via shared state in non-local ways (Figure 1.1). This can produce results that are similar to thread-based interference, although at a more coarse-grained level, as even thread-based interference is rooted at how shared, aliased mutable state can interfere. Our goal is to approach the problem of interference by looking at the low-level, fundamental primitives of state-interference and enable their safe and composable use—both in the sequential and in the concurrent setting.

To achieve this goal, we build on the use of linearity [45] to model resources, namely mutable state. Linearity ensures that the information about the contents of a cell cannot be duplicated, which facilitates the tracking of type-changing ("strong updates" [5]) mutations by forbidding aliasing. Furthermore, we build our stateful abstractions (i.e. our typestates [84, 85]) directly by composing type-theoretic concepts. Although we forfeit some more pragmatic features (such as full object-oriented features, including method dispatch, etc.) this focus is intended to provide a minimal, simple, and relatively standard core language to study interference and aliasing. We start with a linear version of the λ-calculus, based on a variant of $\mathbf{L}^3$ [5], which enables a limited form of (local) aliasing but still forbids all forms of interference (i.e. sharing). From there we develop our mechanism to model (safe) interference and compose interfering aliases that share access to mutable state.

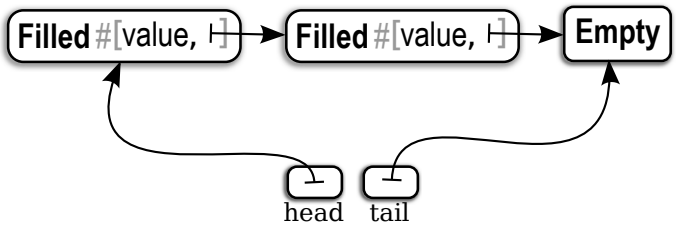

Figure 1.2: Schematics of the pipe example.

We propose a new approach to ensure safe interference by modeling interference in a protocol type. This protocol expresses how state is used locally by capturing each use of the shared state in a single "rely-guarantee" step. Each step initially *relies* on the shared state containing a value of some type, and then *guarantees* that the mutations caused by the alias will result in that state now containing a value of some guaranteed type. Since the kind of interference that an alias produces may change over time, a protocol can sequence different steps so as to model each "period" of changing interference. By individually constraining the actions of each alias we can make strong assumptions about the kind of interference that an alias may produce, in the spirit of rely-guarantee reasoning [55]. Thus, these *rely-guarantee protocols* [64] characterize the individual, local, perspective that an alias has on its uses of the shared state. Naturally, not all combinations of these local protocols are safe. *Protocol compositions* ensures that a protocol is aware of all the possible state changes that may be done through other protocols that are sharing access to that state, regardless of how their uses may be later interleaved. Once protocols are known to compose safely, they can safely and in isolation share (non-local) access to that mutable state without the risk of unsafe interference.

1.2 Example: Shared Pipe

Consider a `Pipe` abstraction that is used to communicate between two (non-local) parts of a program. Pipes include shared, mutable state in the form of an internal buffer that is used to coordinate the interaction between the pipe's clients. We assume that the pipe will have two aliases reflecting different roles: a producer alias, that `put`s values into the buffer; and a consumer alias, that `take`s values out of the buffer. Although we will assign the `put` and `tryTake` functions to separate aliases, the two functions are actually sharing state under that layer of *fictional separation* [34, 36, 54, 56]. The next two code blocks implement simplified versions of the `tryTake` and `put` functions. The internal buffer is modeled as a singly-linked list with a `head` and a `tail` pointer where each pointer will be used by the `tryTake` or the `put` function, respectively. Figure 1.2 shows a schematic of the pipe illustrating how the head and tail pointers share access to nodes of the list that contains filled and empty elements.

We start by looking at the `put` function, which will be used by the producer alias:

```
put = fun( v : Value ).let last = new Empty#{} in
                       let oldlast = !tail in
                         oldlast := Filled#{ v , last };
```

```
      tail := last
    end
  end
```

The function above takes as an argument a value, `v`, that is to be stored in the linked list that is modeling the shared buffer of the pipe. Thus, the `put` function will insert that new value at the end of the linked list, by updating the list's `tail`. Each node of that list contains a pair of elements: the stored value and the pointer to the next node of the list. Therefore, to insert a new element, we must create a new cell that will function as the new `last` node of the list. This cell is tagged as `Empty` to flag that it is the end of the list. Furthermore, the `oldlast` that was the previous node at the tail of the list must be updated to reflect the changes caused by inserting a new value. Thus, the contents of `oldlast` are now tagged as `Filled` and include a pair of elements with the new value, `v`, and the pointer to the next node of the list that we just created, i.e. `tail`. Since `put` only has access to `tail`, we see that the function will no longer be able to reach nodes that precede `tail` as any such node will become unreachable after the function ends.

The use of each shared cell of the linked list can be modeled in a rely-guarantee protocol. In this case, we can model the uses done through the producer alias above as follows:

$$\mathsf{Empty} \Rightarrow \mathsf{Filled} \;;\; \textbf{none}$$

The protocol above contains two steps that were sequenced (;) to combine their uses. The initial "Empty $\Rightarrow$ Filled" step specifies that we initially assume that the shared cell obeys the Empty state, before changing ($\Rightarrow$) that shared cell to Filled. The second step, **none**, is the empty resource that also models the empty step that grants no permission to the shared state. Consequently, the producer alias (governed by the protocol above) forfeits all access to the shared cell after it mutates the shared cell from Empty to Filled.

We now look at the `tryTake` function, which will be assigned to the consumer alias:

```
tryTake = fun().let first = !head in
                  case !first of
                    Empty#_ → NoResult#{}
                  | Filled#[ v , next ] → delete first;
                                          head := next;
                                          Result#v
                  end
                end
```

To test the first node of the list, we must read the contents of the `head` pointer. This node is a cell, and we will inspect the tag of its contents to know whether it is `Filled` or still `Empty`. If the value of that cell is tagged as `Empty`, then `tryTake` failed to extract a (meaningful) value and will signal `NoResult` back to the clients of that function. On the other hand, if we find the `Filled` tag, then we can delete the cell (since it is no longer reachable by other aliases) and update the `head` of the linked list to point to the new `next` node. Finally, the function signals the valid `Result` back to its clients so that they know that a value was extracted.

As before, we can write a rely-guarantee protocol to model the uses of that shared cell as done by the consumer alias, as follows:

$$\textbf{rec}\ X.(\ (\ \mathsf{Empty} \Rightarrow \mathsf{Empty}\ ;\ X\ )\ \ \oplus\ \ (\ \mathsf{Filled} \Rightarrow \textbf{none}\ ;\ \textbf{none}\ )\ )$$

The protocol is **rec**ursive since we may need to use the protocol (X) an arbitrary number of times if we find the shared cell as Empty. Consequently, we must consider two alternative ($\oplus$) cases depending on whether we find the cell Empty or Filled. If the cell is Empty, the protocol states that after some private uses the cell must be restored to Empty, before retrying that same protocol. Alternatively, if we find the cell as Filled we do not need to "give back" any resource, i.e. we only need to guarantee **none** (the empty resource). This enables the consumer alias to retain the Filled state in its own context, rather than return the state to the protocol, effectively recovering ownership of the Filled cell. Ownership recovery enables the `tryTake` function to safely delete the cell since it is no longer shared through any other alias. The subsequent **none** step, simply reaffirms that the protocol has no allowed remaining uses left.

The final piece of our technique is designed to ensure that the two rely-guarantee protocols compose safely, regardless of how they may be later (non-deterministically) interleaved. Although we looked at the two protocols above separately, they are only meaningful if they are used to share access to the same shared cell. For instance, if they are used to "split" a resource (such as an Empty cell) as the consumer and producer protocols described above. We use the following notation to describe the split ($\Rrightarrow$) of the Empty cell:

$$\mathsf{Empty} \Rrightarrow \underbrace{\textbf{rec}\ X.(\,(\,\mathsf{Empty} \Rightarrow \mathsf{Empty}\,;\,X\,) \oplus (\,\mathsf{Filled} \Rightarrow \textbf{none}\,;\,\textbf{none}\,)\,)}_{\text{Consumer}} \parallel \underbrace{\mathsf{Empty} \Rightarrow \mathsf{Filled}\,;\,\textbf{none}}_{\text{Producer}}$$

This split is only valid if the two protocols compose safely. Protocol composition ensures that the split above cannot be the source of unsafe interference by checking the safety of all possible interleaving configurations that may occur through those two protocols. Therefore, each protocol must be aware of the state changes that can be caused by the other protocols regardless of how the uses of the protocols may be interleaved. After the split, each protocol can be used in isolation, and perhaps even further re-split to enable an arbitrary number of new protocols—provided that they too obey our composition conditions.

1.3 Thesis Statement

We make the following statement regarding our sharing/interference-control mechanism:

> **Thesis Statement** *Rely-Guarantee Protocols* provide a modular, composable, expressive, and automatically verifiable mechanism to control the interference resulting from the interaction of non-local aliases that share access to mutable state.

The validation of the thesis statement is broken down into the following key hypothesis:

1.3.1 Hypothesis: Modularity, Composition, and Expressiveness

We claim that our rely-guarantee protocols are an expressive and flexible mechanism for reasoning about interference in a local, modular, and composable way.

Validation By using a small set of primitives as fundamental building blocks, we show that our technique is able to directly characterize many kinds of coordinated interactions that can occur through the use of shared mutable state. Our protocols aim to support a more primitive notion of sharing than related work, by mapping our protocol types to the low-level uses of shared state. We show the expressiveness of these protocols by modeling interesting sharing idioms within our protocol framework. Although in some cases we may require informal extensions to model aspects that we consider orthogonal (and thus, that are not fundamental to handle safe interference), we show how other sharing mechanisms can fit within a rely-guarantee protocol.

Finally, we show that our protocols enable the composition of isolated, modular components of code that make use of shared, mutable state—and that work safely, regardless if used sequentially or concurrently (up to deadlocks). Our protocols enable local reasoning in the face of shared memory interference by having each local interference specification be described in a protocol type that can then be composed with the other protocols of that state to ensure safe interference.

1.3.2 Hypothesis: Automation

We claim that the proposed verification technique can be checked automatically, without the need for user intervention and without relying on manual proofs. More specifically, we claim that protocol composition (the key component of our technique) is a decidable procedure.

Validation We show that protocol composition can be computed without the need for auxiliary annotations (beyond the given types) or user intervention, and that it can be implemented in a reasonably responsive and fast ("lightweight") system. Finally, we show that our composition algorithm is both correct and complete with respect to the formal, axiomatic definition of protocol composition, and that the algorithm terminates on all valid inputs.

We build a proof-of-concept prototype of the full system only for the sequential setting. This prototype shows that the remainder of system can be implemented even if potentially requiring a few additional annotations to direct typing. However, we avoid the engineering work of providing a user-friendly, full-featured, and "practical" implementation of the full language. Namely, we are not concerned in reducing the syntactic burden of the remainder of the language's features (such as finding and defining convenient idioms), or in defining a standard library that supports a large set of common tasks (such as string/integer manipulations, files, etc).

1.3.3 Hypothesis: Soundness

We claim that our approach can be developed into a system that obeys the desired properties of safe interference, as well as standard properties of correctness.

Validation We formalize the proposed system and provide proofs showing that type safety theorems, expressing the desired properties related to the coordinated and safe use of shared state, are respected by such a system. We do this formalization for the concurrent setting, while our earlier work [64] formalized the sequential setting. To show these properties, we also show that

our composition mechanism is sound with respect to ruling out unsafe interference in the use of shared mutable state.

1.4 Contributions

The main contribution of this thesis is the design of the *Rely-Guarantee Protocols* approach for safe interference. This contribution can be broken down into the following individual aspects:

Programming Language We develop a programming language that allows for both safe local aliasing and safe non-local aliasing through the use of our rely-guarantee protocols that control access to shared mutable state. This mechanism, loosely inspired by the spirit of rely-guarantee reasoning, was designed to facilitate local independent reasoning in the uses of shared mutable state. Each alias's use of the shared global state is characterized by a local protocol that models a partial view of the global state. Protocol composition is then used to combine all protocols in a way that ensures safe interference. The language design includes other smaller mechanisms that yet are important for the overall expressiveness of the system.

Proof of Soundness We show our type system is sound by obeying progress and preservation theorems that prove that safe interference cannot occur in well-typed programs. The proof is built on several auxiliary lemmas and theorems, showing standard properties such as that the usual substitution lemma holds. Although we discuss both the sequential and concurrent settings, our proofs focus only on the concurrent setting. (The core of the proof of the sequential setting was covered in [64].) Finally, we show the correctness of our algorithm and that the algorithm terminates on all valid input.

Implementation We give an algorithm to check the key mechanism of our technique, protocol composition, and include a publicly available implementation of the procedure.

1.5 Thesis Outline

The remainder of this thesis is organized into the following chapters:

Chapter 2 Introduces the base, preliminary system. Although this system does not support sharing, it allows local aliasing for reasoning about changes to mutable state. The chapter introduces the core concepts of our base type system and language, which are then used as a foundation to build more complex mechanisms in the subsequent chapters.

Chapter 3 Presents the core of our sharing mechanism detailing all the necessary aspects that are required to ensure safe interference within the setting of a sequential language. The chapter formalizes the three extensions to the base system that enable sharing: protocol specification ("how to specify interference"), protocol composition ("how to safely compose interference"), and protocol manipulation ("how to use protocols").

Chapter 4 Extends the previous system to allow thread-based concurrency, and improves the expressiveness and modularity of our protocol composition mechanism. With these extensions

protocol composition is able to account for polymorphic protocol specifications, enabling more complex (but still local) re-splittings and specializations.

Chapter 5 Presents the algorithmic implementations of subtyping (on types) and protocol composition. To show termination, we define and describe well-formed conditions on types to ensure that any well-formed type has a regular, finite type structure.

Chapter 6 Concludes the thesis with a discussion of open problems and future work, and a brief summary of the work in this document.

Chapter 2

Preliminary System

In typical typed programming languages, the use of mutable state is restricted to invariant types, as each memory cell is constrained to hold a single type of content during its entire lifetime. To obey this condition, mutable state variables are deliberately assigned overly conservative types. However, the use of such types can lead to excessive, error-prone reliance on defensive run-time tests to case analyze the actual state of a cell. Furthermore, while an invariant type can ensure basic memory safety, it cannot check higher-level *protocol* properties [5, 13, 24, 30, 43, 59] that are vital to the correctness of many programs [12]. For instance, in Java a `FileOutputStream` type represents both the open and closed states of that abstraction. Incorrect uses of the underlying state (such as writing to a closed stream) are only detected at run-time since the language cannot accurately track the changing *type of the state*.

In this chapter we introduce the base language, without our mechanism for sharing mutable state. The system shown here is based on our own published work [63], but presented in a more streamlined way. We take an iterative approach to presenting the full language of this chapter. We begin with a variant of the polymorphic λ-calculus and proceed to add functionality to that language in order to precisely check the use of mutable state. To keep our base system relatively simple, we target plain *typestate* [84, 85] state abstractions that yet are sufficiently flexible to enable many interesting and complex uses of mutable state.

The chapter is organized as follows. Section 2.1 introduces the initial iteration of the language, basically a variant of the polymorphic λ-calculus. We introduce the basic grammar, operational semantics, type system, and subtyping rules. Section 2.2 extends the initial language with mutable references (in the style of $\mathbf{L}^3$ [5]). This extension includes new types that provide additional expressiveness, namely support of typestate-like state abstractions. Section 2.3 discusses a few convenient abbreviations that are used throughout the document. Section 2.4 shows a few examples of how the different language features interact with each other and are combined to enable checking practical stateful programs. Section 2.5 exemplifies how our core language is capable of modeling hidden state and includes a discussion on the design trade-offs of building a full system around the notion of the "behavior" of hidden state. Finally, Section 2.6 discusses relevant related work to the work shown in this chapter.

$\texttt{t} \in$ Tags $\quad \texttt{f} \in$ Fields $\quad x, y \in$ Variables $\quad X, Y \in$ Type Variables

$v \in$ Values	$::=$	x	(variable)
	$\mid$	$\lambda x : A.e$	(function)
	$\mid$	$\langle X \rangle\, e$	(type abstraction)
	$\mid$	$\langle A, v \rangle$	(existential type package)
	$\mid$	$\{\overline{\texttt{f} = v}\}$	(record)
	$\mid$	$\texttt{t\#}v$	(tagged value)
$e \in$ Expressions	$::=$	v	(value)
	$\mid$	$v[A]$	(type application)
	$\mid$	$v.\texttt{f}$	(field selection)
	$\mid$	$v\ v$	(application)
	$\mid$	let $x = e$ in e end	(let block)
	$\mid$	open $\langle X, x \rangle = v$ in e end	(type unpacking)
	$\mid$	case v of $\overline{\texttt{t\#}x \rightarrow e}$ end	(case analysis)

Note: $\overline{Z}$ denotes a potentially empty sequence of Z elements.

Figure 2.1: Initial grammar of expressions (e) and values (v). Our types (A) will only be revealed further below, as they become relevant to the discussion.

2.1 Starting Language

Figure 2.1 lists our initial grammar, which is essentially a variant of the polymorphic λ-calculus. For convenience, the grammar is "let-expanded" [80] meaning that the use of expressions within a construct is only expressible by combining a construct with a let. For instance, a case analysis over some expression e' is encoded by enclosing the case with a let that will allow the expression e' to be reduced to a value, prior to the case analysis, i.e.:

$$\text{case } e' \text{ of } \overline{\texttt{l\#}x \rightarrow e} \text{ end} \quad \triangleq \quad \text{let } y = e' \text{ in (case } y \text{ of } \overline{\texttt{l\#}x \rightarrow e} \text{ end) end}$$

The language includes: first-class functions (λ) containing a type-annotated parameter; immutable records (in the style of the language that Reynolds used to research intersection types [79]) using the notation "$\{\overline{\texttt{f} = v}\}$" so that each f identifies the value v associated with that field name; tagged sums, where a value v is tagged as t in "$\texttt{t\#}v$"; and explicit universal and existential type abstraction. Existential packages, "$\langle A, v \rangle$", contain the type A that was abstracted in the value v. To use such a package, we must open it explicitly. "$\langle X \rangle\, e$" allows the type variable X to be used abstractly in the expression e. Type application "$v[A]$" applies the type A to the type abstraction. Field selection, "$v.\texttt{f}$", picks a single field of the v record. Application, "$v\ v'$", applies the v' value as the argument to the v function. case is used to analyze the tags of a sum type. And finally, we have a let block that evaluates the expression that will be bound to x before substituting any occurrence of x in the inner expression of the let. The language itself should be fairly close to standard λ-calculus, even if some notations are somewhat non-standard for conciseness.

$\boxed{H_0 \;;\; e_0 \;\mapsto\; H_1 \;;\; e_1}$ **Dynamics, (D:*)**

$$\frac{}{H \;;\; (\lambda x : A.e)\; v \;\mapsto\; H \;;\; e\{v/x\}} \text{ (D:Application)}$$

$$\frac{}{H \;;\; \{\overline{\mathtt{f} = v}\}.\mathtt{f}_i \;\mapsto\; H \;;\; v_i} \text{ (D:Selection)}$$

$$\frac{}{H \;;\; (\langle X \rangle\, e)[A] \;\mapsto\; H \;;\; e\{A/X\}} \text{ (D:TypeApp)}$$

$$\frac{}{H \;;\; \mathsf{case}\ \mathtt{t}_i \# v_i\ \mathsf{of}\ \overline{\mathtt{t}\#x \to e}\ \mathsf{end} \;\mapsto\; H \;;\; e_i\{v_i/x_i\}} \text{ (D:Case)}$$

$$\frac{}{H \;;\; \mathsf{open}\ \langle X, x\rangle = \langle A, v\rangle\ \mathsf{in}\ e\ \mathsf{end} \;\mapsto\; H \;;\; e\{v/x\}\{A/X\}} \text{ (D:TypeOpen)}$$

$$\frac{}{H \;;\; \mathsf{let}\ x = v\ \mathsf{in}\ e\ \mathsf{end} \;\mapsto\; H \;;\; e\{v/x\}} \text{ (D:Let)}$$

$$\frac{H_0 \;;\; e_0 \;\mapsto\; H_1 \;;\; e_1}{H_0 \;;\; \mathsf{let}\ x = e_0\ \mathsf{in}\ e_2\ \mathsf{end} \;\mapsto\; H_1 \;;\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}} \text{ (D:LetCong)}$$

Note: $\{v/x\}$ denotes the (capture avoiding) substitution of variable x with value v. Similarly, $\{A/X\}$ substitutes the type variable X with type A.

Figure 2.2: Operational semantics of the initial language.

2.1.1 Operation Semantics

Our small step semantics (Figure 2.2) uses judgments of the form:

$$H_0 \;;\; e_0 \;\mapsto\; H_1 \;;\; e_1$$

that step a heap H_0 and expression e_0 to another program configuration with heap H_1 and expression e_1. Thus, a program execution is given by:

$$\emptyset \;;\; e \;\overset{\star}{\mapsto}\; H \;;\; v$$

which states that starting from the empty heap ($\emptyset$) and an initial expression (e), we reach a final configuration of value v with heap H after an arbitrary number of steps, $\mapsto$. Since the language so far does not have mutable state, we will defer introducing our heap structure to a later section. The dynamics are entirely standard, following those of the polymorphic λ-calculus.

$$
\begin{array}{rcll}
A, B & ::= & !A & \text{(pure type)} \\
& | & A \multimap A & \text{(linear function)} \\
& | & \forall X.A & \text{(universal type quantification)} \\
& | & \exists X.A & \text{(existential type quantification)} \\
& | & [\overline{\mathtt{f} : A}] & \text{(record)} \\
& | & \sum_i \mathtt{t}_i \# A_i & \text{(tagged sum)} \\
& | & (\textbf{rec}\ X(\overline{X}).A)[\overline{A}] & \text{(recursive type)} \\
& | & X[\overline{A}] & \text{(type variable)}
\end{array}
$$

Notes: the parenthesis around **rec** (in gray) are only used for clarity and are not part of the syntax.

Figure 2.3: Initial type grammar.

2.1.2 Types Grammar

The initial type structure is depicted in Figure 2.3, with the remaining elements to be introduced with the addition of mutable state.

Some notations, such as the type of functions, borrow from the connectives of linear logic [45]—a logic which is also used to guide the design of the core type system. Therefore, our base type is linear meaning that it cannot be copied. To express that a value can be freely copied, the type must be preceded by a ! ("bang") as in "$!A$". Functions are linear too and use the notation "$A \multimap B$" where A is the type of the argument and B the type of the result. Universal ($\forall$) and existential ($\exists$) quantification over types follow the usual notation. Labeled records "$[\overline{\mathtt{f} : A}]$" (labeled additive conjunction) list a potentially empty list of types that are identified by a unique field name (we will normally select a field in the $\overline{\mathtt{f}}$ set as $\mathtt{f}_i$). We use "$\sum_i \mathtt{t}_i \# A_i$" to denote both a single tagged type or a sequence of tagged types separated by +, such as "$\mathtt{a}\#A + \mathtt{b}\#B + \mathtt{c}\#C$" (i.e. a linear or). We assume that sum types and the set of fields of a record are commutative.

Recursive types are equi-recursive, interpreted co-inductively, and satisfy the usual folding/unfolding principle:

$$
(\textbf{rec}\ X(\overline{Y}).A)[\overline{B}] = A\{(\textbf{rec}\ X(\overline{Y}).A)/X\}\{\overline{B}/\overline{Y}\} \qquad \text{(EQ:REC)}
$$

Recursive types must be non-bottom and can include a list of type parameters ($\overline{Y}$) that are substituted by some types ($\overline{B}$) on unfold, besides unfolding the recursion variable (X). Because of these parameters, our type grammar also accounts for type variables such as "$X[\overline{A}]$" with a list of types ("$\overline{A}$") that are to be applied to a recursive type. However, if such a list is empty, we normally omit the squared brackets of a type variable so that the type variable simply becomes "X". While in prior work [63, 64] we used subtyping rules to fold/unfold recursive types, in here we implicitly use the equivalence relation above whenever necessary.

2.1.3 Type System

We use typing judgments of the form:

$$
\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1
$$

$\boxed{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}$ **Typing rules, (T:*)**

(T:Pure)
$$\frac{\Gamma \mid \cdot \vdash v : A \dashv \cdot}{\Gamma \mid \cdot \vdash v : !A \dashv \cdot}$$

(T:Unit)
$$\frac{}{\Gamma \mid \cdot \vdash v : ![] \dashv \cdot}$$

(T:Pure-Read)
$$\frac{}{\Gamma, x : A \mid \cdot \vdash x : !A \dashv \cdot}$$

(T:Linear-Read)
$$\frac{}{\Gamma \mid x : A \vdash x : A \dashv \cdot}$$

(T:Record)
$$\frac{\overline{\Gamma \mid \Delta \vdash v : A \dashv \cdot}}{\Gamma \mid \Delta \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot}$$

(T:Selection)
$$\frac{\Gamma \mid \Delta_0 \vdash v : [\overline{\mathtt{f} : A}] \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash v.\mathtt{f}_i : A_i \dashv \Delta_1}$$

(T:Pure-Elim)
$$\frac{\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1}$$

(T:Function)
$$\frac{\Gamma \mid \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta \vdash \lambda x : A_0.e : A_0 \multimap A_1 \dashv \cdot}$$

(T:Application)
$$\frac{\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2}$$

(T:Forall-Type)
$$\frac{\Gamma, X : \mathbf{type} \mid \Delta \vdash e : A \dashv \cdot}{\Gamma \mid \Delta \vdash \langle X \rangle\, e : \forall X.A \dashv \cdot}$$

(T:Type-App)
$$\frac{\Gamma \vdash A_1\ \mathbf{type} \qquad \Gamma \mid \Delta_0 \vdash v : \forall X.A_0 \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash v[A_1] : A_0\{A_1/X\} \dashv \Delta_1}$$

(T:Type-Pack)
$$\frac{\Gamma \mid \Delta \vdash v : A_0\{A_1/X\} \dashv \cdot}{\Gamma \mid \Delta \vdash \langle A_1, v \rangle : \exists X.A_0 \dashv \cdot}$$

(T:Type-Open)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \exists X.A_0 \dashv \Delta_1 \qquad \Gamma, X : \mathbf{type} \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{open}\ \langle X, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$$

(T:Tag)
$$\frac{\Gamma \mid \Delta \vdash v : A \dashv \cdot}{\Gamma \mid \Delta \vdash \mathtt{t}\#v : \mathtt{t}\#A \dashv \cdot}$$

(T:Case)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \sum_i \mathtt{t}_i \# A_i \dashv \Delta_1 \qquad \overline{\Gamma \mid \Delta_1, x_i : A_i \vdash e_i : A \dashv \Delta_2} \qquad i \le j}{\Gamma \mid \Delta_0 \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_2}$$

(T:Let)
$$\frac{\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1, x : A_0 \vdash e_1 : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_2}$$

(T:Subsumption)
$$\frac{\Delta_0 <: \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash e : A_0 \dashv \Delta_2 \qquad A_0 <: A_1 \qquad \Delta_2 <: \Delta_3}{\Gamma \mid \Delta_0 \vdash e : A_1 \dashv \Delta_3}$$

(T:Frame)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0, \Delta_2 \vdash e : A \dashv \Delta_1, \Delta_2}$$

Notes: all bound variables of a construct must be fresh in the respective rule's conclusion (i.e. x must be fresh in (T:Let) conclusion, etc.).

Figure 2.4: Static semantics.

stating that with lexical environment Γ and linear resources Δ_0 we assign the expression e the type A and where the expression produces effects that result in the resources contained in Δ_1 (along the lines of type-and-effect systems [44]). Due to its linear meaning, the resources in Δ_0 are either fully used up by the expression e or are "threaded through" that expression and moved to Δ_1 as residual resources.

For the system shown so far, the typing contexts are defined as follows:

$$
\begin{array}{rcll}
\Gamma & ::= & \cdot & \text{(empty)} \\
 & | & \Gamma,\ x : A & \text{(variable binding)} \\
 & | & \Gamma,\ X : k & \text{(kind assertion)} \\
\Delta & ::= & \cdot & \text{(empty)} \\
 & | & \Delta,\ x : A & \text{(linear binding)} \\
k & ::= & \mathbf{type} \mid \mathbf{type} \to k &
\end{array}
$$

where *k*inds of the form "$\mathbf{type} \to k$" are used to ensure that the use of recursive types is well-formed, in an analogous way of how a function type is used to type functions and function application.

Figure 2.4 lists the typing rules of this preliminary language, that we now discuss. Observe how values (which includes functions, tagged values, etc.) have no resulting effects ($\cdot$) since they have no pending computations. We impose the expected global constraint on all constructs with bound variables (such as let, functions, etc.): these variables must be fresh in the conclusion of the respective typing rule.

If a value v does not use any linear resources (i.e. "$\Gamma \mid \cdot \vdash v : A \dashv \cdot$") then its type can be made pure ($!A$) using (T:Pure). (T:Unit) allows any value to be assigned a unit type since the unit type forbids any actual use of that value, so a unit usage is always safe. If a variable is of a pure type, then (T:Pure-Elim) allows the binding to be moved to the linear context with its type explicitly "banged" with !. Reads of variables from the lexical context, (T:Pure-Read), requires the obtained type to be preceded by !. Destructive (linear) reads from the linear context, (T:Linear-Read), make the read linear variable unavailable for further use.

The fields of a record, (T:Record), contain a set of labeled choices. Since field selection, (T:Selection), will pick one field and discard the rest, we requires all fields to produce the same effect. Therefore, even if these fields contain some linear type, that value can be safely discarded as its effects must be equal to those produced by the single, selected field. Thus, a record type is akin to a linear (labeled) intersection type.

Since a function, (T:Function), can depend on the linear resources inside of Δ (which the function captures), a functional value must be linear. However, the function can later be rendered pure (!) through the use of (T:Pure) if the set of linear resources it captures is actually empty. Thus, the fact that our basic function type is linear is not an actual restriction to the language expressiveness, since it may be combined with other type constructors which circumvent linearity. We rely on (T:Pure-Elim) to use non-linear arguments, so that any argument can be initially assumed to be of linear kind. (T:Application) is the traditional rule.

(T:Forall-Type), (T:Type-App), (T:Type-Open) and (T:Type -pack) provide ways to use type abstractions. The absence of a "$\Gamma \vdash A_1$ **type**" premise in (T:Type-Pack) is explained by the fact that,

$\boxed{A_0 <: A_1}$ **Subtyping on types, (ST:*)**

$$\frac{}{A <: A}\ (\text{ST:SYMMETRY}) \qquad \frac{A_0 <: A_1}{!A_0 <: A_1}\ (\text{ST:TOLINEAR}) \qquad \frac{A_0 <: A_1}{!A_0 <: !A_1}\ (\text{ST:PURE}) \qquad \frac{}{!A <: ![]}\ (\text{ST:PURETOP})$$

$$\frac{A_1 <: A_3 \qquad A_2 <: A_0}{A_0 \multimap A_1 <: A_2 \multimap A_3}\ (\text{ST:FUNCTION}) \qquad \frac{A_0 <: A_1}{\exists X.A_0 <: \exists X.A_1}\ (\text{ST:TYPE-EXISTS}) \qquad \frac{A_0 <: A_1}{\forall X.A_0 <: \forall X.A_1}\ (\text{ST:TYPE-FORALL}) \qquad \frac{\overline{A_i <: B_i} \qquad n \leq m}{\sum_i^n \mathtt{t}_i \# A_i <: \sum_i^m \mathtt{t}_i \# B_i}\ (\text{ST:SUM})$$

$$\frac{[\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad A_i <: B_i}{[\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : B}\ ,\ \mathtt{f}_i : B_i]}\ (\text{ST:RECORD}) \qquad \frac{[\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad \overline{\mathtt{f} : A} \neq \emptyset}{[\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : B}]}\ (\text{ST:DISCARD})$$

Note: the double-line means that the rule should be interpreted co-inductively.

Figure 2.5: Subtyping on types.

$\boxed{\Delta_0 <: \Delta_1}$ **Subtyping on deltas, (SD:*)**

$$\frac{\Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Delta_0, x : A_0 <: \Delta_1, x : A_1}\ (\text{SD:VAR}) \qquad \frac{}{\Delta <: \Delta}\ (\text{SD:SYMMETRY})$$

Figure 2.6: Subtyping on linear environments.

for A_1 to occur in A_0 it must be a well-formed **type**, or if it does not occur in A_0 then it is of no consequence whether A_1 is or not well-formed.

(T:TAG) and (T:CASE) provide the introduction and elimination of tagged sum types. The (T:CASE) rule allows the set of tags of the value that is to be case analyzed (v) to be *smaller* than those listed in the branches of the case ($i \leq j$). This conditions is safe because it amounts to ignoring the effects of those branches, instead of being overly conservative and having to always consider all branches. Although it may appear that these apparent "dead" branches are useless, they will play an important role in later sections when checking mutable state as we will discuss further below.

A (T:LET) will thread the variable x to the body of the let, so that we can assume a variable of such a type when checking e_1. As above with functions, we can safely assume that the type of x is linear, and use (T:PURE-ELIM) to move x to the pure typing context if relevant.

Our (T:FRAME) accounts for simple disjoint separation [26, 78] so that linear resources that are not needed to type check an expression cannot be changed by that expression either (i.e., are just "thread through" the expression).

Finally, the (T:SUBSUMPTION) rule allows an expression to rely on weaker assumptions while ensuring a stronger result than needed by its context. This rule is supported by subtyping rules on

types (Figure 2.5) and on linear typing contexts (Figure 2.6). Subtyping on types follows the form $A_0 <: A_1$ stating that A_0 is a subtype of A_1, meaning that A_0 can be used wherever A_1 is expected. Similar meaning is used for subtyping on linear typing environments, $\Delta_0 <: \Delta_1$. Due to our recursive types being equi-recursive, subtyping on types is to be interpreted co-inductively, thus using the double line to separate premise(s) from conclusion. Most of these rules are straightforward, we will discuss the few that may be slightly more subtle. (ST:PURETOP) allows any pure type to become the pure unit type. (ST:DISCARD) allows fields of a record to be discarded as long as there remains at least one field left. Thus, if the field captures some linear resources, these are guaranteed to not be lost. (ST:FUNCTION) employs the usual subtyping rule on function types, so that subtyping is contravariant on the argument type but covariant on the return type.

Note that we lack (explicit) subtyping rules over recursive types. This is due to our use of a co-inductive definition of subtyping combined with our implicit use of (EQ:REC) for folding/unfolding a recursive type as necessary. Therefore, any subtyping over a **rec** results in an implicit unfold via (EQ:REC) before applying some subtyping rule. Since we restrict recursive types to contain regular sub-terms (as will be discussed in chapter 5) and forbid bottom types, this unfolding will eventually enable us to close the co-inductive proof of subtyping on a **rec** term.

$$\frac{}{\textbf{rec}\ X.A\ <:\ \textbf{rec}\ X.B} \xrightarrow{\text{(EQ:REC)}} \frac{\vdots}{A\{\textbf{rec}\ X.A/X\}\ <:\ B\{\textbf{rec}\ X.B/X\}}$$

2.2 Extension: Mutable State

We handle mutable state by following the design proposed by Ahmed, Fluet, and Morrisett in $\mathbf{L}^3$ [5], which enables precise type-changing mutations to the content of mutable state. Therefore, the contents of a cell are no longer restricted to obey an invariant type. With this design, a cell is decomposed into two components: a pure *reference* (that can be freely copied), and a linear *capability* that is used to track the contents of that cell. Location-dependent types are then used to link a reference to its respective capability. For instance, a new cell has type:

$$\exists l.(\ (!\textbf{ref}\ l)\ ::\ (\textbf{rw}\ l\ A)\ )$$

This type abstracts the fresh location, l, that was created by the memory allocation. Furthermore, we are given a reference of type "**ref** l" to mean a **ref**erence to a location l, where the information about the contents of that location is stored in the capability for l. The permission to access, such as by dereference, the contents of a cell requires both the reference and the capability to be available. Our capabilities follow the format "**rw** l A" meaning a **r**ead-**w**rite capability to location l which currently has contents of type A (in this case, the type of the initial value stored in the new cell). Unlike $\mathbf{L}^3$, and to reduce some syntactical overhead, we leave capabilities as typing artifacts without an actual value. As a result, we need a way to associate a reference of some location with the capability to access that location, as occurs when the cell is first created. We do this with a *stacking* operator "::", that places the capability for l on top of the **ref**erence type. Capabilities are moved implicitly, as needed through the program, by enabling the type system to stack and

unstack capabilities on top of some other type. However, for clarity, our examples will manually stack capabilities (using the construct "$e :: A$" where A is the stacked capability) whenever relevant, although the type system can do stacking/unstacking automatically. Finally, locations must be managed explicitly in the language so that, for instance, they must be packed/unpacked with constructs that are designed specifically to deal with locations (since locations are not types).

With this scheme, all variables that reference the same location are effectively sharing access to a linear (i.e. "exclusively owned" or "unique" [18]) capability that tracks the changes to that potentially aliased cell. Therefore, local aliasing is straightforward to reason about since any change will affect the same typing structure—the capability that tracks the changes to the aliased location. Consider the following example[1]:

```
1 open <l,y> = new 0 in                 Γ = y : ref l, l : loc | Δ = rw l int
2   y := "ok!";                         Γ = y : ref l, l : loc | Δ = rw l string
3   let x = y in                        Γ = x : ref l, y : ref l, l : loc | Δ = rw l string
4     x := false;                       Γ = x : ref l, y : ref l, l : loc | Δ = rw l boolean
5     delete x;                         Γ = x : ref l, y : ref l, l : loc | Δ = ·
6     !y // type error! Cannot dereference 'y' due to missing capability to location 'l'.
7   end
8 end
```

As stated above, creating a new cell will yield a type that abstract the fresh location that was created. Therefore, line 1 must open that location package by giving a variable for the location (`l`) and a variable (y) for the reference the package contains. From that point on, we can assign (:=) new values to that cell, which may change the type of the contents of the respective capability (lines 2 and 4); we may dereference that cell (!y); or even delete the cell (line 5), which will destroy the capability thus making it unavailable for further uses (regardless from which alias). Note that the let of line 3 simply creates a new name for "**ref** `l`" but leaves `l`'s capability unchanged.

This tracking effectively creates a compile-time approximation of how variables must alias, which constrains how state can be used. Since the linear capability must be threaded through the program, this scheme forbids aliasing idioms that require "simultaneous" access to aliased state from points in the program where that location-link cannot be preserved. The technique proposed in this thesis, and presented in later chapters, enables us to share state even in that case. However, before we can present our technique, we will discuss this extension to the preliminary language in more detail to highlight its expressiveness even when only considering linear capabilities.

Besides adding support for linear capabilities (as a typing artifact), we also add support for: *alternative types* ($\oplus$) to model imprecise knowledge on the available resources, so that we can have a union of different capabilities; *intersection types* (&) to give clients a choice between different resources; and *separation types* ($*$) to bundle multiple resources together, before stacking, so that the group of stacked resources is commutative (which, by itself, stacking is not).

We now present the remainder of the preliminary language, by extending the starting language.

[1]For brevity, we omit the preceding "!" on all primitive types, such as **`int`** and **`string`** types, although these types are pure and can be freely copied.

$\rho \in$ Location Constants (Addresses) $\quad l \in$ Location Variables $\quad p ::= \rho \mid l$

$\mathtt{t} \in$ Tags $\quad \mathtt{f} \in$ Fields $\quad x, y \in$ Variables $\quad X, Y \in$ Type Variables

$v \in$ Values	$::=$	x	(variable)
	$\mid$	$\lambda x : A.e$	(function)
	$\mid$	$\langle X \rangle\, e$	(type abstraction)
	$\mid$	$\langle A, v \rangle$	(pack type)
	$\mid$	$\{\overline{\mathtt{f} = v}\}$	(record)
	$\mid$	$\mathtt{t}\#v$	(tagged value)
	$\mid$	$\langle p, v \rangle$	(pack location)
	$\mid$	$\langle l \rangle\, e$	(location abstraction)
	$\mid$	ρ	(address)
$e \in$ Expressions	$::=$	v	(value)
	$\mid$	$v[A]$	(type application)
	$\mid$	$v.\mathtt{f}$	(field)
	$\mid$	$v\ v$	(application)
	$\mid$	let $x = e$ in e end	(let)
	$\mid$	open $\langle X, x \rangle = v$ in e end	(open type)
	$\mid$	case v of $\overline{\mathtt{t}\#x \rightarrow e}$ end	(case)
	$\mid$	$v[p]$	(location application)
	$\mid$	open $\langle l, x \rangle = v$ in e end	(open location)
	$\mid$	new v	(cell creation)
	$\mid$	delete v	(cell deletion)
	$\mid$	$!v$	(dereference)
	$\mid$	$v := v$	(assign)

Note: ρ is not source-level.

Figure 2.7: Expressions (e) and values (v).

2.2.1 Grammar

The grammar from Figure 2.1 is now expanded into the grammar of Figure 2.7, with the changes highlighted with a gray background. All changes are related to the use of mutable state, and the consequent manipulation of locations. Capabilities are not present as values in the language, and are just used at the level of types. Also note that we use p to range over locations, which includes both location *variables*, l, and location *constants*, ρ (i.e. memory addresses). Because locations are of a separate kind to types, they require specific constructs to open/pack a location and abstract/apply locations. (We use lower-cased letters for location variables, while type variables use upper-cased letters.) Besides those constructs, we use new to allocate memory with initial contents of a given value and "delete v" to destroy the location referenced by v. Assignments of a value v_1 to the location referenced by a value v_0 are written "$v_0 := v_1$". To access the contents of a reference v, we write "$!v$".

2.2.2 Operational Semantics

The operation semantics of Figure 2.2 are now extended with the new constructs that can operate over a heap. The heap (H) binds addresses (ρ) to values (v) using the following format:

$$H \quad ::= \quad \emptyset \quad \text{(empty)} \quad | \quad H,\ \rho \hookrightarrow v \quad \text{(binding)}$$

The new semantics are fairly straightforward, except for the constructs that allocate and delete cells. The (D:New) and (D:Delete) reduction rules, as in [5], manipulate existential values that abstract the underlying location that was created or will be deleted, in order for the type system to properly handle these location abstractions (i.e. for the value to match the desired existential type).

2.2.3 Type System

The extended types grammar is shown in Figure 2.9. Note that we use a "flat" type grammar where both capabilities (i.e. typing artifacts without values) and standard types (used to type values) coexist. Our design does not need to make a syntactic distinction between the two kinds since the type system ensures the proper separation in their use. As above, we use a gray background to highlight the changes to the type grammar of Figure 2.3.

We now describe the main additions. The stacked type "$A :: B$" stacks resource B on top of A. This stacking is not commutative since it stacks a single resource on the right of ::. Therefore, $*$ enables multiple resources to be grouped together that, when later stacked, allow that type to list a commutative group of resources. Thus, while "$A_0 :: (A_1 :: A_2)$" and "$A_0 :: (A_2 :: A_1)$" are not (necessarily) subtypes, resource commutation is always possible with $*$ such that "$A_0 :: (A_1 * A_2) <:> A_0 :: (A_2 * A_1)$". A "**ref** p" type is a reference to location p noting that the contents of such a reference are tracked by the capability to that location and not immediately stored in the reference type. **rec**ursive types follow those presented above, in Section 2.1.2, with the addition that the parameter/argument can now also range over locations, not just types. Alternatives ($\oplus$) model imprecision in the knowledge of the type by listing different possible resources that may be valid. **none** is the empty resource (equivalent to **1** from linear logic), while "**rw** p A" is the read-write capability to location p (a memory cell currently containing a value of type A). Finally, an "$A\&B$" type means that the client can choose to use either type A or type B, but not both simultaneously (equivalent to an unlabeled additive conjunction in linear logic). $\oplus$, &, and $*$ are assumed to be associative and commutative.

Although our type structure presents capabilities and value-inhabited types together, our type system ensures that the two must be properly combined into more complex type expressions. For instance, that a **none** type is not inhabited by any value, etc. Alternatively, capabilities and value-inhabited types could also be presented separately. With our type grammar we simplify the syntax by avoiding some redundancy in types that overlap as capabilities and as standard types since such distinction is technically not very important. In fact, even if types that are not inhabited by a value are assumed (such as in a function's argument) they can never be created/introduced which effectively means that such a value/type will never be usable anyway.

$\boxed{H_0 \,;\, e_0 \;\mapsto\; H_1 \,;\, e_1}$ **Dynamics, (D:*)**

(D:Application)
$$\frac{}{H \,;\, (\lambda x : A.e)\; v \;\mapsto\; H \,;\, e\{v/x\}}$$

(D:Selection)
$$\frac{}{H \,;\, \{\overline{\mathtt{f} = v}\}.\mathtt{f}_i \;\mapsto\; H \,;\, v_i}$$

(D:TypeApp)
$$\frac{}{H \,;\, (\langle X \rangle\, e)[A] \;\mapsto\; H \,;\, e\{A/X\}}$$

(D:Case)
$$\frac{}{H \,;\, \mathsf{case}\ \mathtt{t}_i \# v_i\ \mathsf{of}\ \overline{\mathtt{t} \# x \to e}\ \mathsf{end} \;\mapsto\; H \,;\, e_i\{v_i/x_i\}}$$

(D:TypeOpen)
$$\frac{}{H \,;\, \mathsf{open}\ \langle X, x \rangle = \langle A, v \rangle\ \mathsf{in}\ e\ \mathsf{end} \;\mapsto\; H \,;\, e\{v/x\}\{A/X\}}$$

(D:Let)
$$\frac{}{H \,;\, \mathsf{let}\ x = v\ \mathsf{in}\ e\ \mathsf{end} \;\mapsto\; H \,;\, e\{v/x\}}$$

(D:LetCong)
$$\frac{H_0 \,;\, e_0 \;\mapsto\; H_1 \,;\, e_1}{H_0 \,;\, \mathsf{let}\ x = e_0\ \mathsf{in}\ e_2\ \mathsf{end} \;\mapsto\; H_1 \,;\, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}}$$

(D:New)
$$\frac{\rho\ \textbf{fresh}}{H \,;\, \mathsf{new}\ v \;\mapsto\; H \,,\, \rho \hookrightarrow v \,;\, \langle \rho, \rho \rangle}$$

(D:Delete)
$$\frac{}{H \,,\, \rho \hookrightarrow v \,;\, \mathsf{delete}\ \langle \rho, \rho \rangle \;\mapsto\; H \,;\, \langle \rho, v \rangle}$$

(D:Dereference)
$$\frac{}{H \,,\, \rho \hookrightarrow v \,;\, !\rho \;\mapsto\; H \,,\, \rho \hookrightarrow v \,;\, v}$$

(D:Assign)
$$\frac{}{H \,,\, \rho \hookrightarrow v_0 \,;\, \rho := v_1 \;\mapsto\; H \,,\, \rho \hookrightarrow v_1 \,;\, v_0}$$

(D:LocApp)
$$\frac{}{H \,;\, (\langle l \rangle\, e)[\rho] \;\mapsto\; H \,;\, e\{\rho/l\}}$$

(D:LocOpen)
$$\frac{}{H \,;\, \mathsf{open}\ \langle l, x \rangle = \langle \rho, v \rangle\ \mathsf{in}\ e\ \mathsf{end} \;\mapsto\; H \,;\, e\{v/x\}\{\rho/l\}}$$

Note: $\{v/x\}$ denotes the (capture avoiding) substitution of variable x with value v. Similarly, $\{A/X\}$ substitutes the type variable X with type A; and $\{p/l\}$ substitutes a location variable l with p.

Figure 2.8: Operational semantics.

$$u \;::=\; l \mid X \qquad\qquad U \;::=\; p \mid A$$

A, B	$::=$	$!A$	(pure type)
	$\mid$	$A \multimap A$	(linear function)
	$\mid$	$\forall X.A$	(universal type quantification)
	$\mid$	$\exists X.A$	(existential type quantification)
	$\mid$	$[\overline{\mathtt{f} : A}]$	(record)
	$\mid$	$\sum_i \mathtt{t}_i \# A_i$	(tagged sum)
	$\mid$	$\mathbf{ref}\ p$	(reference type)
	$\mid$	$(\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]$	(recursive type)
	$\mid$	$X[\overline{U}]$	(type variable)
	$\mid$	$A :: A$	(resource stacking)
	$\mid$	$A * A$	(separation)
	$\mid$	$\forall l.A$	(universal location quantification)
	$\mid$	$\exists l.A$	(existential location quantification)
	$\mid$	$A \oplus A$	(alternative)
	$\mid$	$A \,\&\, A$	(intersection)
	$\mid$	$\mathbf{rw}\ p\ A$	(read-write capability to p)
	$\mid$	$\mathbf{none}$	(empty resource)

Figure 2.9: Types (including capabilities) grammar.

We now extend our initial type system (show in Figure 2.4) by including the typing rules of Figure 2.10. To account for locations and linear resources, the typing environment is extended to be defined as follows:

Γ	$::=$	$\cdot$	(empty)
	$\mid$	$\Gamma,\ x : A$	(variable binding)
	$\mid$	$\Gamma,\ p : \mathbf{loc}$	(location assertion)
	$\mid$	$\Gamma,\ X : k$	(kind assertion)
Δ	$::=$	$\cdot$	(empty)
	$\mid$	$\Delta,\ x : A$	(linear binding)
	$\mid$	$\Delta,\ A$	(linear resource)
k	$::=$	$\mathbf{type} \mid \mathbf{type} \to k \mid \mathbf{loc} \to k$	

We now discuss the typing rules shown in Figure 2.10.

(T:Forall-Loc), (T:Loc-App), (T:Loc-Open), and (T:Loc-pack) are analogous to their -Type counterparts of Figure 2.4 but over **loc**ations. (T:Ref) types any location constant as long as it refers to a known location. Note that a location is more like a pointer or memory address, not a traditional reference since it still lacks the capability to actually access the contents stored at that location.

Allocating a new cell (typed with (T:New)) results in a type that abstracts the fresh location that was created, and stacks the new capability on top of the resulting reference. Since owning the **rw** capability of a location implies uniqueness of access, deleting (T:Delete) is only allowed for a

$\boxed{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}$ **Typing rules, (T:*)**

(T:FORALL-LOC)
$$\frac{\Gamma, l : \mathbf{loc} \mid \Delta \vdash e : A \dashv \cdot}{\Gamma \mid \Delta \vdash \langle l \rangle\, e : \forall l.A \dashv \cdot}$$

(T:LOC-APP)
$$\frac{p : \mathbf{loc} \in \Gamma \qquad \Gamma \mid \Delta_0 \vdash v : \forall l.A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash v[p] : A\{p/l\} \dashv \Delta_1}$$

(T:LOC-PACK)
$$\frac{\Gamma \mid \Delta \vdash v : A\{p/l\} \dashv \cdot}{\Gamma \mid \Delta \vdash \langle p, v \rangle : \exists l.A \dashv \cdot}$$

(T:REF)
$$\frac{}{\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\, \rho \dashv \cdot}$$

(T:LOC-OPEN)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \exists l.A_0 \dashv \Delta_1 \qquad \Gamma, l : \mathbf{loc} \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{open}\, \langle l, x \rangle = v\, \mathsf{in}\, e\, \mathsf{end} : A_1 \dashv \Delta_2}$$

(T:NEW)
$$\frac{\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{new}\, v : \exists l.((!\mathbf{ref}\, l) :: (\mathbf{rw}\, l\, A)) \dashv \Delta_1}$$

(T:DELETE)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \exists l.((!\mathbf{ref}\, l) :: (\mathbf{rw}\, l\, A)) \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{delete}\, v : \exists l.A \dashv \Delta_1}$$

(T:ASSIGN)
$$\frac{\Gamma \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_0 : \mathbf{ref}\, p \dashv \Delta_2, \mathbf{rw}\, p\, A_1}{\Gamma \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\, p\, A_0}$$

(T:DEREFERENCE-LINEAR)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\, p \dashv \Delta_1, \mathbf{rw}\, p\, A}{\Gamma \mid \Delta_0 \vdash !v : A \dashv \Delta_1, \mathbf{rw}\, p\ ![]}$$

(T:DEREFERENCE-PURE)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\, p \dashv \Delta_1, \mathbf{rw}\, p\ !A}{\Gamma \mid \Delta_0 \vdash !v : !A \dashv \Delta_1, \mathbf{rw}\, p\ !A}$$

(T:CAP-ELIM)
$$\frac{\Gamma \mid \Delta_0, x : A_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : A_0 :: A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:CAP-STACK)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

(T:CAP-UNSTACK)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}$$

(T:ALTERNATIVE-LEFT)
$$\frac{\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1 \qquad \Gamma \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:ALTERNATIVE-RIGHT)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \oplus A_2}$$

(T:INTERSECTION-LEFT)
$$\frac{\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1, A_1}{\Gamma \mid \Delta_0, A_0 \& A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:INTERSECTION-RIGHT)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \qquad \Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_2}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \& A_2}$$

Notes: all bound variables of a construct must be fresh in the respective rule's conclusion (i.e. x must be fresh in (T:LET) conclusion, etc.).

Figure 2.10: Static semantics (continued from Figure 2.4).

type that includes both the reference *and* the capability for that value. Our examples use an idiom for delete that avoids packing the location to be deleted. For consistency with the new rule we use the packed reference version in here.

We allow two kinds of pointer dereference: a linear version (T:Dereference-Linear) that destroys the contents of the capability, and a pure version, (T:Dereference-Pure), which leaves the same (pure) type behind. Note that although (T:Dereference-Linear) is destructive, operationally it will not destroy the contents of that cell. In this case, preservation of typing is ensured through the use of (T:Unit) so that the leftover value is effectively unusable "junk" from the type system's perspective when read from that capability after a linear dereference. Assigning (T:Assign) requires both the reference and the respective capability to be available.

Stacking, done through (T:Cap-Elim), (T:Cap-Stack), and (T:Cap-Unstack) enables the type system to manage resources (such as capabilities) in a non-syntax directed way, since they have no value nor associated identifier.

(T:Alternative-Left) expresses that if an expression types with both assumptions, A_0 and A_1, then it works with both alternatives. Note that the rule does not require that the resulting type to distinguish between the two different alternatives; we just require that each case can be considered independently. This means that an alternative type is only usable when there exists an expression that can satisfy all those alternative cases. Otherwise such a type can only be threaded through and never inspected on each separate case. (T:Intersection-Right) is similar but on the resulting effect of that expression. Also note that (T:Alternative-Right) and (T:Intersection-Left) are derivable through subtyping, but are shown here for consistency of the presentation.

Although the rule for typing $\oplus$ may appear too permissive, since each alternative is not anchored on some value that can be case analyzed, this is actually not a weakness. Our design relies on the combination of alternatives and sum types to enable code that distinguishes between the individual alternatives, as we now exemplify.

$\oplus$ and case interplay The importance of the interaction between alternative types and our case construct is perhaps clearer to understand in the following example. In this example we have multiple alternatives carrying different sets of available capabilities. Consider the following code where, for readability, we use the name of a variable (such as `x`) preceded by `@` as the name of the location the variable refers to (i.e. `@x`):

```
Γ = x : ref @x , @x : loc , y : ref @y , @y : loc , z : ref @z , @z : loc
                        Δ = ( rw @z HasX#![] ∗ rw @x ![] ) ⊕ ( rw @z HasY#![] ∗ rw @y ![] )
                       [a] Δ = rw @z HasX#![] , rw @x ![]    [b] Δ = rw @z HasY#![] , rw @y ![]
  case !z of           [a] Δ = rw @z ![] , rw @x ![]         [b] Δ = rw @z ![] , rw @y ![]
    HasX#_ →           [a] Δ = rw @z ![] , rw @x ![]
      delete x         [a] Δ = rw @z ![]
  | HasY#_ →                                                 [b] Δ = rw @z ![] , rw @y ![]
      delete y                                               [b] Δ = rw @z ![]
  end                      Δ = rw @z ![]
```

In the code above, each branch deletes state that the other branch does not use. This means that, although both branches know about the same set of locations, their actions over the heap

$\boxed{A_0 <: A_1}$ **Subtyping on types, (ST:*)**

$$\frac{}{A <: A}\ (\text{ST:SYMMETRY}) \qquad \frac{A_0 <: A_1}{!A_0 <: A_1}\ (\text{ST:TOLINEAR}) \qquad \frac{A_0 <: A_1}{!A_0 <: !A_1}\ (\text{ST:PURE}) \qquad \frac{}{!A <: ![]}\ (\text{ST:PURETOP})$$

$$\frac{A_1 <: A_3 \qquad A_2 <: A_0}{A_0 \multimap A_1 <: A_2 \multimap A_3}\ (\text{ST:FUNCTION}) \qquad \frac{A_0 <: A_1}{\exists X.A_0 <: \exists X.A_1}\ (\text{ST:TYPE-EXISTS}) \qquad \frac{A_0 <: A_1}{\forall X.A_0 <: \forall X.A_1}\ (\text{ST:TYPE-FORALL}) \qquad \frac{\overline{A_i <: B_i} \qquad n \leq m}{\sum_i^n \mathtt{t}_i \# A_i <: \sum_i^m \mathtt{t}_i \# B_i}\ (\text{ST:SUM})$$

$$\frac{[\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad A_i <: B_i}{[\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : B}\ ,\ \mathtt{f}_i : B_i]}\ (\text{ST:RECORD}) \qquad \frac{[\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad \overline{\mathtt{f} : A} \neq \emptyset}{[\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : B}]}\ (\text{ST:DISCARD})$$

$$\frac{A_0 <: A_1}{\exists l.A_0 <: \exists l.A_1}\ (\text{ST:LOC-EXISTS}) \qquad \frac{A_0 <: A_1}{\forall l.A_0 <: \forall l.A_1}\ (\text{ST:LOC-FORALL}) \qquad \frac{A_0 <: A_2}{A_0 <: A_2 \oplus A_1}\ (\text{ST:ALTERNATIVE}) \qquad \frac{A_0 <: A_2}{A_0 \& A_1 <: A_2}\ (\text{ST:INTERSECTION})$$

$$\frac{A_0 <: A_1 \qquad A_2 <: A_3}{A_0 :: A_2 <: A_1 :: A_3}\ (\text{ST:STACK}) \qquad \frac{A_0 <: A_1}{\mathbf{rw}\ p\ A_0 \ <:\ \mathbf{rw}\ p\ A_1}\ (\text{ST:CAP}) \qquad \frac{A_0 <: A_2 \qquad A_1 <: A_3}{A_0 * A_1 <: A_2 * A_3}\ (\text{ST:STAR})$$

$$\frac{A_0 <: A_1 \qquad A_2 <: A_3}{A_0 \oplus A_2 <: A_1 \oplus A_3}\ (\text{ST:ALTERNATIVE-CONG}) \qquad \frac{A_0 <: A_1 \qquad A_2 <: A_3}{A_0 \& A_2 <: A_1 \& A_3}\ (\text{ST:INTERSECTION-CONG})$$

Figure 2.11: Subtyping on types.

are distinct. The static semantics of our case enables these seemingly incompatible alternative program states to be obeyed simultaneously by the same case expression, by allowing a case analysis to ignore branches that are not listed in its sum type. Therefore, a program alternative will not need to type those potentially irreconcilable branches and will solely consider the relevant branches of its sum type. However, our approach may leave "dead" branches since we do not ensure that a branch in a case will be used by at least one program alternative.

2.2.4 Subtyping

The changes to subtyping are shown in Figures 2.11 and 2.12. Unlike in traditional non-linear systems, our linear capabilities only need to be read-consistent (not write) which yields the additional flexibility shown in (ST:CAP) (i.e. due to their linearity they are covariant, as in other linear/affine

$\boxed{\Delta_0 <: \Delta_1}$ **Subtyping on deltas, (SD:*)**

$$\frac{\Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Delta_0, x : A_0 <: \Delta_1, x : A_1}\ (\text{SD:VAR}) \qquad \frac{}{\Delta <: \Delta}\ (\text{SD:SYMMETRY})$$

$$\frac{\Delta, A_0, A_1 <: \Delta, A_1, A_2}{\Delta, A_0, A_1 <: \Delta, A_1 * A_2}\ (\text{SD:STAR-R}) \qquad \frac{\Delta, A_0 * A_1 <: \Delta, A_1 * A_2}{\Delta, A_0 * A_1 <: \Delta, A_1, A_2}\ (\text{SD:STAR-L})$$

$$\frac{\Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Delta_0, A_0 <: \Delta_1, A_1}\ (\text{SD:TYPE})$$

$$\frac{\Delta_0 <: \Delta_1}{\Delta_0 <: \Delta_1, \mathbf{none}}\ (\text{SD:NONE-R}) \qquad \frac{\Delta_0 <: \Delta_1}{\Delta_0, \mathbf{none} <: \Delta_1}\ (\text{SD:NONE-L})$$

$$\frac{\Delta_0, A_0 <: \Delta_1 \quad \Delta_0, A_1 <: \Delta_1}{\Delta_0, A_0 \oplus A_1 <: \Delta_1}\ (\text{SD:ALTERNATIVE-L}) \qquad \frac{\Delta_0 <: \Delta_1, A_1 \quad \Delta_0 <: \Delta_1, A_2}{\Delta_0 <: \Delta_1, A_1 \& A_2}\ (\text{SD:INTERSECTION-R})$$

Figure 2.12: Subtyping Environments.

systems [29]). (SD:STAR-*) allows us to bundle/group several linear resources together using $*$, or break this bundle into its separate components. (ST:ALTERNATIVE) enables a type to be weakened, and consider additional alternatives. (ST:INTERSECTION) enables a choice of which type to use. (SD:NONE-*) can be used to either remove or add the empty resource, **none**.

In this sytem, transitivity is admissible meaning that we do not need an explicit subtyping rule.

Lemma 1 (Subtype Transitivity)**.** *We have that:*

- *If* $A_0 <: A_1$ *and* $A_1 <: A_2$ *then* $A_0 <: A_2$.
- *If* $\Delta_0 <: \Delta_1$ *and* $\Delta_1 <: \Delta_2$ *then* $\Delta_0 <: \Delta_2$.

2.2.5 Technical Results

This system follows the design of [63] and is proven sound through *progress* and *preservation* theorems. These statements use the definition of store typing shown in Figure 2.13, which relates well-formed environments with heaps. Store typing uses judgments of the form "$\Gamma \mid \Delta \vdash H$" stating that the heap H conforms with the elements contained in Γ and Δ. Although typing rules such as the frame rule may appear to potentially extend our linear resources in arbitrary ways, our theorems show that when starting from a well-typed store we will never reach invalid store states. Note that, similarly to above, (STR:ALTERNATIVE) assumes that $\oplus$ is commutative, so we only introduce a single store typing rule for store typing alternative resources.

We now state our main theorems:

Theorem 1 (Progress)**.** *If* e_0 *is a closed expression (and where* Γ *and* Δ_0 *are also closed) such that:*

$$\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$$

then either:

$\boxed{\Gamma \mid \Delta \vdash H}$ **Store typing, (STR:*)**

$$\frac{}{\cdot \mid \cdot \vdash \cdot}\ (\text{STR:EMPTY}) \qquad \frac{\Gamma \mid \Delta \vdash H}{\Gamma, \rho : \mathbf{loc} \mid \Delta \vdash H}\ (\text{STR:LOC}) \qquad \frac{\Gamma \mid \Delta, A_0, A_1 \vdash H}{\Gamma \mid \Delta, A_0 * A_1 \vdash H}\ (\text{STR:STAR})$$

$$\frac{\Gamma \mid \Delta \vdash H}{\Gamma \mid \Delta, \mathbf{none} \vdash H}\ (\text{STR:NONE}) \qquad \frac{\Gamma \mid \Delta, A_0 \vdash H}{\Gamma \mid \Delta, A_0 \oplus A_1 \vdash H}\ (\text{STR:ALTERNATIVE}) \qquad \frac{\Gamma \mid \Delta, A_0 \vdash H \quad \Gamma \mid \Delta, A_1 \vdash H}{\Gamma \mid \Delta, A_0 \& A_1 \vdash H}\ (\text{STR:INTERSECTION})$$

$$\frac{\Gamma \mid \Delta, \Delta_v \vdash H \qquad \Gamma \mid \Delta_v \vdash v : A \dashv \cdot}{\Gamma \mid \Delta, \mathbf{rw}\ \rho\ A \vdash H, \rho \hookrightarrow v}\ (\text{STR:BINDING})$$

Figure 2.13: Store typing.

- e_0 *is a value, or;*
- *if exists* H_0 *such that* $\Gamma \mid \Delta_0 \vdash H_0$ *then* $H_0 \,;\, e_0 \;\mapsto\; H_1 \,;\, e_1$.

The progress statement ensures that all well-typed expressions are either values or, if there is a heap that obeys the typing assumptions, the expression can step to some other program state — i.e. a well-typed programs never gets stuck.

Theorem 2 (Preservation)**.** *If* e_0 *is a closed expression such that:*

$$\Gamma_0 \mid \Delta_0 \vdash e_0 : A \dashv \Delta \qquad \Gamma_0 \mid \Delta_0 \vdash H_0 \qquad H_0 \,;\, e_0 \;\mapsto\; H_1 \,;\, e_1$$

then, for some Δ_1, Γ_1*:*

$$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash H_1 \qquad \Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta$$

The theorem above requires the initial expression e_0 to be closed so that it is ready for evaluation. The preservation statement ensures that the resulting effects (Δ) and type (A) of the expression remains the same throughout the execution so that the initial typing is preserved by the dynamics of the language. Heap modifications may occur (such as on delete or new) but these preserve the previously known locations and are a consequence of the operational semantics (i.e. of the resulting H_1).

Note that no instrumentation of the operational semantics is necessary since in our system the presence of a memory cell in the heap must also have its respective capability in Δ (our capabilities are linear). Any kind of wrong use of the state can be reduced to a stuck condition in the language. For instance, deleting a cell too early will cause the program to become stuck when an alias tries to access that same location later on. Therefore, these two theorems are enough to ensure that state is properly used even if done through multiple aliases of the same underlying location. By both theorems we ensure traditional type safety in the sense that "correct programs do not go wrong" since every well-typed program either terminates with a value of the expected type or will simply run forever.

2.3 Some Abbreviations

We define a few convenient abbreviations that are used in examples and that can be encoded into the core language. Thus, we keep our core language relatively small but still support additional, convenient constructs.

Let-Expansions: Although our grammar is let-expanded, we can "revert" this expansion by combining lets with other constructs. All these constructions follow the same principle as for instance dereference of expressions ("!e"):

$$!e \triangleq \mathsf{let}\ x = e\ \mathsf{in}\ !x\ \mathsf{end}$$

where a let is used to evaluate the expression to be dereference before doing the actual dereference. Similar constructs can be built for assignments, open, etc.

Sequence: A sequence "$e_0; e_1$" evaluates e_0 before evaluating e_1, discarding e_0's result:

$$e_0; e_1 \triangleq \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end}$$

Where x is not free in either e_1 or e_0. Note that e_0 must have a pure type since the value will be discarded. However, it can have capabilities stacked on top since those get automatically threaded to e_1. This automatic threading is done by using (T:Cap-Stack) to stack those capabilities on top of x followed by (T:Cap-Elim) to make those capabilities available in the context that types e_1, without exposing the variable x to the programmer.

Tuples: In our system, tuples (akin to the multiplicative conjunction from linear logic) can be encoded using function and application. Labeled records present a choice of labeled products, so that the programmer must pick one (and only one) of a set of available fields. The alternative would be to require all those fields to be used, effectively being a labeled pair (so that the order matters). However, we can encode unlabeled pairs in our language. We show encoding binary tuples, but similar reasoning can be generalized to arbitrary (fixed) lengths.

$$\frac{\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash e_1 : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \{e_0, e_1\} : [A_0, A_1] \dashv \Delta_2}$$

The creation of a tuple follows the left-to-right evaluation order.

$$\frac{\Gamma \mid \Delta_0 \vdash e_0 : [A_0, A_1] \dashv \Delta_1 \qquad \Gamma \mid \Delta_1, x_0 : A_0, x_1 : A_1 \vdash e_1 : A_2 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{open}\ [x_0, x_1] = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_2 \dashv \Delta_2}$$

Opening a tuple (where x_0 and x_1 are fresh in the conclusion), is similar to a standard let.

The two new constructs above can be encoded as follows:

$$\begin{array}{rcl} \{e_0, e_1\} & \triangleq & \mathsf{let}\ x_0 = e_0\ \mathsf{in} \\ & & \quad \mathsf{let}\ x_1 = e_1\ \mathsf{in} \\ & & \quad\ \texttt{<R>}\lambda(\texttt{f} : A_0 \multimap A_1 \multimap R).(\ \texttt{f}\ x_0\ x_1\ ) \\ & & \quad \mathsf{end} \\ & & \mathsf{end} \end{array}$$

$$\begin{array}{rcl} \mathsf{open}\ [x_0, x_1] = e_p\ \mathsf{in}\ e_f\ \mathsf{end} & \triangleq & \mathsf{let}\ p\ =\ e_p\ \mathsf{in} \\ & & \quad p[A_f]\ (\ \lambda(x_0 : A_0).\lambda(x_1 : A_1).e_f\ ) \\ & & \mathsf{end} \end{array}$$

where e_0 has type A_0, e_1 has type A_1, and e_f has type A_f.

Recursion: For encoding recursion, we use the traditional call-by-value Y-combinator. Thus, we can encode recursion in our core language without needing additional typing rules or reductions.

```
1 let fix = <A><B>λ( f : !( ( A ⊸ B ) ⊸ ( A ⊸ B ) ) ).
2   let r = λ( x : rec X.!( X ⊸ ( A ⊸ B ) ) ).
3       f ( λ( v : A ).(x(x))(v) ) in
4     r r
5   end : !∀A.∀B.( !(( A ⊸ B ) ⊸ ( A ⊸ B )) ⊸ ( A ⊸ B ) )
```

Noting that x in line 3 has types:

$$\begin{array}{ll} !(\ \mathbf{rec}\ X.!(\ X \multimap (\ A \multimap B\ )\ )\ \multimap (\ A \multimap B\ )\ ) & \text{(function)} \\ \mathbf{rec}\ X.!(\ X \multimap (\ A \multimap B\ )\ ) & \text{(argument)} \end{array}$$

Making the argument of `f` in that line to be $A \multimap B$, and `r` to be of type:

$$\mathbf{rec}\ X.!(\ X \multimap (\ A \multimap B\ )\ ) \multimap (\ A \multimap B\ )$$

which, applied to itself, yields the type of the function without the recursive argument visible ($A \multimap B$). Therefore, to use recursion, we must create a function that takes the recursive function as argument, as usually done in the literature. In the examples, we make use of the construct "rec $x.e$" to define a recursive function (with body e), without having to use the expanded notation.

Shorter delete rule: We also define a "shorter" delete typing rule to avoid carrying existential types around:

$$\mathsf{delete'}\ x \triangleq\ \ \mathsf{open}\ \ <l_y, y>= (\ \mathsf{delete}\ \ <l_x, x>\ \ )\ \mathsf{in}\ y\ \mathsf{end}$$

where x has type $\mathbf{ref}\ l_x$ and y has type A, and where l_y does not occur in A (and therefore does not need to be packed to leave that scope).

Similar functionality could be achieved with the following typing rule:

$$\frac{\Gamma \mid \Delta_0 \vdash e : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A}{\Gamma \mid \Delta_0 \vdash \mathsf{delete'}\ e : A \dashv \Delta_1}\ \text{(T:DELETE')}$$

For consistency with (T:NEW) we leave (T:DELETE') out of the formal system.

2.4 Typestate Examples

In this Section we model two simple typestate examples: a stack object and a stateful pair. We show how our system is able to capture some complex typestate notions that were developed for more ambitious typestate systems [13], within the language that we presented so far. Although we lack some of their more advanced features (such as full object-orientation with method dispatch, inheritance, etc.), we aim to show the expressiveness of the preliminary system by reconstructing some typestate features from our core low-level language constructs.

Typestates enable abstractions to define a more precise type-like specification by exposing *abstract states* that represent the internal, changing, state of an abstraction—accurately tracking the changing properties of the *type of the state*. Pioneering work on typestate [10, 13, 14, 32, 33, 40, 84, 85] systems, including empirical evidence of typestate [12] uses, showed that there are significant reliability gains on pursuing such a kind of protocol verification and allowing the code to explicitly express typestate properties. For instance, in Java a `FileOutputStream` type represents both the open and closed states of that abstraction. Dynamic tests are then used to detect incorrect uses such as writing to a closed stream. A typestate system enables not only the programmer to explicitly express the precise states that they are assuming, but also enables the type system to statically check that the correct states are being used.

By using typestates, an abstraction can precisely (but abstractly) express the dynamic behavior of the data type. For example, a `stack` object can expose two abstract states: *empty* and *non-empty*. Furthermore, if an object statically falls into an imprecise state, client code may inspect the statically known state using *dynamic state tests* [13] (or similar techniques, such as *keyed variants* [32]). For instance, each `stack` object offers `push` and `pop` operations, but the latter requires the non-empty state as a pre-condition. Therefore, when a `stack`'s actual representation state is unknown, clients may rely on an `isEmpty` function, which performs a run-time test on the representation, without exposing that representation to clients.

With the following examples, we show that we can model abstract states (i.e. typestates) using existential quantification on the (substructural) state representation; we show how we can express indirect dynamic tests of abstracted state by leveraging alternative types and case analysis; and we combine separation with abstraction to represent disjoint abstract states.

2.4.1 Stack Object

We now consider the stack implementation of Figure 2.14. The stack object is coded using a linear singly-linked list, the only private state that the stack uses. This example illustrates many prominent features of our approach, namely how we are able to use case analysis on sums to indirectly test for capabilities to different abstract states—i.e. perform a dynamic test over the abstracted state of the object. Remember that capabilities are not values, and therefore if there are multiple possible alternative capabilities there is no direct way of distinguishing between them. However, a case analysis over tags in a value (using the syntax "`tag#`*val*") can provide ways of deciding which of the different alternatives the abstraction's state must be in, and we can use this to learn about the capabilities that must be available.

```
1  let newStack = <T> λ _ : ![] .                                   Γ = _ : [] , T : type  |  Δ = ·
2   open <h,head> = new E#{} in                                    Γ = ... , h : loc , head : ref h  |  Δ = rw h E#![]
3   {
4    push = λ e : T :: ( Nil[h] ⊕ Node[h] ) .                      Γ = ...  |  Δ = e : T , Nil[h] ⊕ Node[h]
                                                                   [a] Δ = e : T , Nil[h]   [b] Δ = e : T , Node[h]
5              open <n,next> = new !head in                        Γ = ... , n : loc , next : ref n
                                                                   [a] Δ = e : T , rw h ![] , Nil[n]   [b] Δ = e : T , rw h ![] , Node[n]
                                                                   [a] [b] Δ = e : T , rw h ![] , Nil[n] ⊕ Node[n]
6                head := N#{ e , <n,next :: (Nil[n] ⊕ Node[n])> } //tagged next node
                                                                   [a] [b] Δ = rw h N#[T, ∃p.((!ref p) :: (Nil[n] ⊕ Node[n]))]
                                                                   Δ = Node[h]
7              end,
8
9     pop = λ _ : ![] :: Node[h] .                                 Γ = ... , _ : []  |  Δ = Node[h]
                                                                   Δ = rw h N#[T, ∃p.((!ref p) :: (Nil[n] ⊕ Node[n]))]
10             case !head of
11               N#[e,n] → //sugared pair open                   Δ = rw h ![], e : T, n : ∃p.((!ref p) :: (Nil[n] ⊕ Node[n]))
12                 open <t,ptr> = n in   Γ = ..., t : loc, ptr : ref t | Δ = rw h ![], e : T, Nil[t] ⊕ Node[t]
                                         [a] Δ = rw h ![], e : T, Nil[t]   [b] Δ = rw h ![], e : T, Node[t]
13                   head := !ptr;      [a] Δ = rw t ![], e : T, Nil[h]   [b] Δ = rw t ![], e : T, Node[h]
                                                                   Δ = rw t ![], e : T, Nil[h] ⊕ Node[h]
14                   delete ptr;                                   Δ = e : T, Nil[h] ⊕ Node[h]
15                   e                                             Δ = Nil[h] ⊕ Node[h]
16                 end
17             end,
18
19    isEmpty = λ _ : ![] :: ( Nil[h] ⊕ Node[h] ) .                Γ = ... , _ : []  |  Δ = Nil[h] ⊕ Node[h]
                                                                   [a] Δ = Nil[h]   [b] Δ = Node[h]
20             case !head of // 'head' has linear content, read is destructive
21               E#v → // we must restore linear value to 'head', after inspection
22                 head := E#v;                                    [a] Δ = Nil[h]
23                 Empty#({} :: Nil[h])                            [a] Δ = ·
24             | N#n →
25                 head := N#n;                                    [b] Δ = Node[h]
26                 NonEmpty#({} :: Node[h])                        [b] Δ = ·
27             end,
28
29    del = λ _ : ![] :: Nil[h] .                                  Γ = ... , _ : [] | Δ = rw h E#![]
30             delete head                                         Δ = ·
31    }
32  end
```

Figure 2.14: Implementation of a stack showing the typing environments computed during type checking. We omit Γ and other parts of the typing environments whenever they are not relevant to express the intuition for how type checking proceeds on that expression. Explicit stacking of capabilities is shown in blue since it is not a language construct, but may be relevant to understand how type checking proceeds.

We start by defining the following types, that take a location variable p as parameter:[2]

$$\begin{aligned} \texttt{Nil}[p] &\triangleq \textbf{rw}\ p\ \texttt{E\#}[] \\ \texttt{Node}[p] &\triangleq \textbf{rw}\ p\ \texttt{N\#}[\texttt{T}, \exists q.(\ (!\textbf{ref}\ q) :: (\texttt{Nil}[q] \oplus \texttt{Node}[q])\ )] \end{aligned}$$

`Nil` encodes an empty node, while `Node` is a non-empty node whose successor may or may not be empty. Alternatives ($\oplus$) express the set of different capabilities that the successor node may have. The necessity of such a type is directly linked to the fact that, since capabilities are not values, it is not possible to wrap capabilities around a sum type to provide a distinctive tag that identifies each separate case, as otherwise we would fall into a system where capabilities must be manually threaded. Instead the type system is able to account for this uncertainty in the program state through the different alternatives listed in a $\oplus$.

Using the types above, the `newStack` function has the non-abstracted type (note that this type cannot leave the scope where location h is known):

$$\begin{aligned} \forall \texttt{T}.(\ ![] \multimap (\quad ![\quad & \texttt{push} : !((\texttt{T} :: (\texttt{Nil}[h] \oplus \texttt{Node}[h])) \multimap (![] :: \texttt{Node}[h])), \\ & \texttt{pop} : !((![] :: \texttt{Node}[h]) \multimap (\texttt{T} :: (\texttt{Nil}[h] \oplus \texttt{Node}[h]))), \\ & \texttt{isEmpty} : !((![] :: (\texttt{Nil}[h] \oplus \texttt{Node}[h])) \multimap \\ & \qquad (\texttt{Empty\#}(![] :: \texttt{Nil}[h]) + \texttt{NonEmpty\#}(![] :: \texttt{Node}[h]))), \\ & \texttt{del} : !((![] :: \texttt{Nil}[h]) \multimap ![])\ ] \\ :: \texttt{Nil}[h]\ )\ ) \end{aligned}$$

All these functions and the returned record are pure since none of them captures resources in their scope. However, the capability to the internal state is linear, and will be threaded through the functions as necessary, with the type system ensuring its non-duplication. By abstracting the capabilities we can then construct a typestate abstraction for the stack object (where `E` denotes the empty state and `NE` the non-empty state), typed as (and omitting !'s for clarity since they appear in the same relative position as in the type above):

$$\begin{aligned} \forall \texttt{T}.[] \multimap \exists \texttt{E}.\exists \texttt{NE}.(\ [\quad & \texttt{push} : (\texttt{T} :: (\texttt{E} \oplus \texttt{NE})) \multimap ([] :: \texttt{NE}), \\ & \texttt{pop} : ([] :: \texttt{NE}) \multimap (\texttt{T} :: (\texttt{E} \oplus \texttt{NE})), \\ & \texttt{isEmpty} : ([] :: (\texttt{E} \oplus \texttt{NE})) \multimap (\texttt{Empty\#}([] :: \texttt{E}) + \texttt{NonEmpty\#}([] :: \texttt{NE})), \\ & \texttt{del} : ([] :: \texttt{E}) \multimap []\ ] \\ :: \texttt{E}\ ) \end{aligned}$$

The most interesting aspect of using alternatives and sum types is shown in the `isEmpty` function. In it, we see that the result returns a sum type where the capabilities of the different alternatives are separated. Therefore, this function enables clients to test in which case their state is in, even though the state is abstracted and not immediately accessible—i.e. perform a *dynamic state test*—and with it tying the typing artifact with a tagged value that can be tested at run-time. We discuss this aspect and other details of the type system by looking at specific lines of code in the stack example.

[2]For clarity, we write $\texttt{H}[X] \triangleq ...\texttt{H}[X]$ instead of directly writing the underlying recursive type with parameters.

Stack/Unstacking Capabilities can be stacked and unstacked automatically in our system. For instance, the function definition of lines 4 to 7 results in the following typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma \mid \mathsf{e} : T,\ \mathtt{Nil}[h] \oplus \mathtt{Node}[h] \vdash \mathsf{open}\ \dots\ \mathsf{end} : ![] \dashv \mathtt{Node}[h]}}{\Gamma \mid \mathsf{e} : T,\ \mathtt{Nil}[h] \oplus \mathtt{Node}[h] \vdash \mathsf{open}\ \dots\ \mathsf{end} : (![] :: \mathtt{Node}[h]) \dashv \cdot}\ (\textsc{t:Cap-Stack})}{\Gamma \mid \mathsf{e} : (T :: (\mathtt{Nil}[h] \oplus \mathtt{Node}[h])) \vdash \mathsf{open}\ \dots\ \mathsf{end} : (![] :: \mathtt{Node}[h]) \dashv \cdot}\ (\textsc{t:Cap-Elim})}{\Gamma \mid \cdot \vdash \lambda \mathsf{e} : (T :: (\mathtt{Nil}[h] \oplus \mathtt{Node}[h])).\mathsf{open}\ \dots\ \mathsf{end} : (T :: (\mathtt{Nil}[h] \oplus \mathtt{Node}[h])) \multimap (![] :: \mathtt{Node}[h]) \dashv \cdot}\ (\textsc{t:Function})$$

Alternative States The use of alternatives means that the type checker knows that we have one of several different capabilities and consequently (to be safe) the expression must consider all those cases individually. For instance, on line 5, to be able to dereference `head` we must open the alternative type "$\mathtt{Nil}[h] \oplus \mathtt{Node}[h]$" using (T:Alternative-Left). We then typecheck the same expression but assuming the two individual alternatives (marked in Figure 2.14 as **[a]** for $\mathtt{Nil}[h]$, and **[b]** for $\mathtt{Node}[h]$) separately. By breaking the alternative, we can then access the content of `head` which is the same reference to n regardless of the alternative. On line 6, the type system automatically stacks the capabilities to location n (although we show it explicitly, in blue). In order for both alternatives to stack the same type, we must first use subsumption to allow the use of the subtyping rule (ST:Alternative) so that we have:

$$\mathtt{Nil}[n] \ <:\ \mathtt{Nil}[n] \oplus \mathtt{Node}[n] \qquad \text{and} \qquad \mathtt{Node}[n] \ <:\ \mathtt{Nil}[n] \oplus \mathtt{Node}[n]$$

Thus, on line 6, the two alternatives are now using identical assumptions and checking line 6 will result in the same type and effects.

Linearity and Destructive Reads Whenever a cell has linear contents, reading those contents will result in a destructive read, c.f. (T:Dereference-Linear). Thus, to preserve linearity, the location that is read cannot keep the same type as initially. Instead, it must be changed to have the unit type, a type that forbids all uses of that value. Because of this condition, our system may require some apparently redundant reassignments in order to restore a previously read linear value.

For instance, on lines 22 and 25, we reassign the `head` reference so as to restore its initial type after the inspection of line 20 although the value stored remains the same. Since the contents of `head` include linear types, which cannot be duplicated, the dereference of line 20 must leave the capability for h with the unit type ("**rw** h ![]") so that the linear type can be bound to the branch's variable without incurring duplication. Therefore, the apparent redundant assignment operations are necessary to counter the destructive read that occurs at the type-level by refreshing the h cell with the same value that the cell initially contained.

Alternative Types and Case Analysis Perhaps the most subtle, but important, combination of typing rules occurs with (T:Case) and (T:Alternative-Left). Our `case` gains precision by ignoring the effects of branches that are statically known to never be used.

For instance, the implementation of the `isEmpty` function distinguishes the alternatives indirectly based on the case branch that is taken as a result of the run-time value contained in the capability. Therefore, on line 20, when the type checker is case analyzing the contents of `head` on alternative **[a]** it obtains the type "`E#![]`". Instead of weakening the type to consider all the remaining branches of that case, we simply ignore the case branches that the type does not list (similar to ideas employed in [31, 38]). Consequently, for that alternative, type checking only takes into account the `E` tag and the respective branch.

Although this use may appear fastidious in this particular example, later chapters will rely on this interaction between ⊕ and case to enable different/incompatible uses of the state based on the branch that is taken at run-time.

2.4.2 Stateful Pair

We now consider a stateful pair object that uses two private memory cells, `l` and `r`, to store the left and right components of the pair. These components are private in the sense that the two references are never exposed outside the closures that represent the object's methods—only the *type* of the two components will be visible to clients. Thus, as with the previous stack example, we employ the familiar idiom of representing "objects" as records of closures.

Abstracting the two components of the pair under the same type(state) would be excessively "coarse-grained". Doing so would effectively require access to both left and right components of the pair even on functions where access to a single component would suffice. Instead, our system supports individually abstracting multiple separate, independent states. In the example below, we show how each component of the pair can be used in isolation through a separate state abstraction, and also how these typestates may be grouped together on functions that require simultaneous access to both left and right cells of the pair.

```
1  let newPair = λ _ : ![] .
2    open <pl,l> = new {} in
3    open <pr,r> = new {} in
4      {
5          initL = λ i : ( int :: ( rw pl ![] ) ). l := i,
6          initR = λ i : ( int :: ( rw pr ![] ) ). r := i,
7            sum = λ _ : ( ![] :: ( rw pl int * rw pr int ) ). !l+!r,
8       destroy = λ _ : ( ![] :: ( rw pl int * rw pr int ) ). delete l; delete r
9      }
10   end
11   end
```

The code above defines a function, `newPair`, which creates a new pair object. As with the stack object, clients can only use the pair object through the functions in the labeled record ("{`initL` : ..., `initR` : ...}") that `newPair` returns.

Capabilities can be stacked in a function's arguments, as is the case on line 5 with the `initL` function. An argument capability of this form does not need to be present when the function is defined, but rather must be provided by the caller when the function is invoked. Consequently, and since capabilities are threaded implicitly (i.e. the resulting type has capabilities automatically

stacked on top), `initL` has the following type:

$$!(\ (\mathsf{int} :: (\mathbf{rw}\ pl\ ![])) \multimap\ (![] :: (\mathbf{rw}\ pl\ \mathsf{int}))\ )$$

where the enclosing ! signals that the linear function ($\multimap$) is pure — its definition does not capture any enclosing linear resources (capabilities), and so the function is safe to be invoked multiple times. Instead, the `initL` function "borrows" [17, 67] the linear resources it requires (the capability to *pl*), returning them together with the result of the function.

Thus, the `newPair` function has type:

$$\begin{array}{l}!(\ ![]\ \multimap\ (\ ![\ \ \texttt{initL}: \ !(\ (\mathsf{int} :: (\mathbf{rw}\ pl\ ![])) \multimap\ (![] :: (\mathbf{rw}\ pl\ \mathsf{int}))\ ),\\ \qquad\qquad\quad \texttt{initR}: \ !(\ (\mathsf{int} :: (\mathbf{rw}\ pr\ ![])) \multimap\ (![] :: (\mathbf{rw}\ pr\ \mathsf{int}))\ ),\\ \qquad\qquad\qquad \texttt{sum}: \ !(\ (![] :: ((\mathbf{rw}\ pl\ \mathsf{int}) * (\mathbf{rw}\ pr\ \mathsf{int}\ ))) \multimap\ (\mathsf{int} :: ((\mathbf{rw}\ pl\ \mathsf{int}) * (\mathbf{rw}\ pr\ \mathsf{int})))\ ),\\ \qquad\qquad\ \texttt{destroy}: \ !(\ (![] :: ((\mathbf{rw}\ pl\ \mathsf{int}) * (\mathbf{rw}\ pr\ \mathsf{int})) \multimap\ ![]\ )\ ]\\ \qquad\qquad :: ((\mathbf{rw}\ pl\ ![]) * (\mathbf{rw}\ pr\ ![]))\ )\ )\end{array}$$

When we need to stack multiple capabilities, the use of the :: type constructor becomes a technical burden since it does not allow for reordering of capabilities. With this in mind, we introduce $*$ to form an unordered *bundle* of separate capabilities as is the case with the arguments for `sum` and `destroy`, and the initial capabilities (that resulted from the memory allocation) that are stacked on top of the resulting record.

This group of disjoint state models the notion of *state dimensions* [13, 86] such that the global state of the pair object obeys several, separate, typestates that can be used independently in certain functions (such as `initL` and `initR`) but are required together on others (such as for calling `sum` and `destroy` functions). Our design differentiates the two basic type checking operations of moving a capability on top of some other type (::) and grouping sets of capabilities together ($*$), so that each operation is independent of the other and, although frequently used together, they are modeling separate typing aspects. However, other systems [50] do not make such distinction in their design.

The type above is not free to leave the scope of the open constructs, since it depends on local names for the *pl* and *pr* locations. A first attempt to fix this issue is to provide *location polymorphism* [83] to abstract those location variables once again. If we change the previous code so that we pack both locations using the following construct to wrap the previous record definition:

```
4   <pl,<pr, { /* same functions */ } > >
```

we now obtain the following version of the `newPair` function type where the result is an existential type where *pl* is replaced by *l* and *pr* by *r* (hiding !'s for clarity, since they are the same as above):

$$\begin{array}{l}[]\ \multimap\ \exists l.\exists r.(\ [\ \ \texttt{initL}: \ (\mathsf{int} :: (\mathbf{rw}\ l\ [])) \multimap\ ([] :: (\mathbf{rw}\ l\ \mathsf{int})),\\ \qquad\qquad\quad \texttt{initR}: \ (\mathsf{int} :: (\mathbf{rw}\ r\ [])) \multimap\ ([] :: (\mathbf{rw}\ r\ \mathsf{int})),\\ \qquad\qquad\ \ \texttt{sum}: \ ([] :: ((\mathbf{rw}\ l\ \mathsf{int}) * (\mathbf{rw}\ r\ \mathsf{int}))) \multimap\ (\mathsf{int} :: ((\mathbf{rw}\ l\ \mathsf{int}) * (\mathbf{rw}\ r\ \mathsf{int}))),\\ \qquad\quad \texttt{destroy}: \ ([] :: ((\mathbf{rw}\ l\ \mathsf{int}) * (\mathbf{rw}\ r\ \mathsf{int}))) \multimap\ []\ ]\\ \qquad\qquad :: ((\mathbf{rw}\ l\ []) * (\mathbf{rw}\ r\ []))\ )\end{array}$$

With this version of the `newPair` function, the result can now safely leave the scope of the function's definition. However, it still exposes the internal representation of the pair object's state to client code. To fully abstract that representation, we can use standard type abstraction mechanisms, using existential types but now to pack the types of the capabilities by wrapping the record with the following constructs:

```
4   < rw pl ![], // hides capability as "Empty Left" (EL)
5   < rw pr ![], // "Empty Right" (ER)
6   < rw pl int, // "Left initialized" (L)
7   < rw pr int, // "Right initialized" (R)
8     { /* same functions */ } > > > >
```

This expression will produce a type that completely abstracts the representation of the pair object's state, exposing only the requirements for using the pair in terms of abstracted linear capabilities (i.e. typestates). To provide some intuition on the meaning of the abstracted types, we choose names such as `EL` to represent the abstracted capability for the `L`eft part of the pair when that part is `E`mpty, etc. Therefore, the final version of our `newPair` function is simply (again with !'s omitted for clarity):

$$
\begin{aligned}
[] \multimap \exists \mathtt{EL}.\exists \mathtt{ER}.\exists \mathtt{L}.\exists \mathtt{R}.([\ \ & \mathtt{initL} : (\mathsf{int} :: \mathtt{EL}) \multimap ([] :: \mathtt{L}), \\
& \mathtt{initR} : (\mathsf{int} :: \mathtt{ER}) \multimap ([] :: \mathtt{R}), \\
& \mathtt{sum} : ([] :: (\mathtt{L} * \mathtt{R})) \multimap (\mathsf{int} :: (\mathtt{L} * \mathtt{R})), \\
& \mathtt{destroy} : ([] :: (\mathtt{L} * \mathtt{R})) \multimap [] \ ] \\
& :: (\mathtt{EL} * \mathtt{ER}))
\end{aligned}
$$

The following client code exemplifies how calling `initL` and `initR` only affects the parts of the (type)state that the respective functions require and use.

```
1 open < EL, < ER, < L, < R, x > > > > = newPair({}) in //sugared open of all abstractions   Δ = EL, ER
2   x.initL(12);                                                                            Δ =  L, ER
3   x.initR(34);                                                                            Δ =  L,  R
4   x.sum({});                                                                              Δ =  L,  R
5   x.destroy({})                                                                           Δ = ·
```

where the typing environments at the beginning of line 2 contain:

$$\Gamma = \mathtt{x} : [\ldots] \ , \ \mathtt{EL} : \textbf{type} \ , \ \mathtt{ER} : \textbf{type} \ , \ \mathtt{L} : \textbf{type} \ , \ \mathtt{R} : \textbf{type} \quad | \quad \Delta = \mathtt{EL} \ , \ \mathtt{ER}$$

Observe how the left and right typestates of the pair operate over independent dimensions of the complete state of the pair, making each change separate [26, 78] up until they are required together (such as for invoking `sum` or `destroy`). In those cases, we stack both types on top of the {} value (grouped together as $*$) and unstack them if/when returned.

The last version of the type of the `newPair` function shows a complete state abstraction of the internal details of the pair's implementation. Thus, its type expresses only the *type of the state* as in *typestate* [13, 84, 85] approaches. Through the use of standard type abstraction mechanisms, we can abstract the actual (internal) type representation of the state. Client code only needs to be aware of these typestates, enabling implementation independence that yet ensures that all constraints related to the use of the pair's internal resources are obeyed.

Extra Separation Example Perhaps a clearer use of this separation is exemplified by the following (untyped) function:

```
λ x, y.( x.initL(12); y.initR(34) )
```

where `initL` and `initR` are called over seemingly unrelated names.

In our system we can give that function the type:

$$\forall A.\forall B.\forall C.\forall D.(\,[\texttt{initL} : (\mathsf{int} :: A) \multimap (![] :: B)] \multimap [\texttt{initR} : (\mathsf{int} :: C) \multimap (![] :: D)] \multimap (![] :: B * D)\,)$$

Such that the two function calls work completely independently, regardless if the state involved is or not referring to the same underlying pair object—fully exploiting the disjointness of the pair's state.

2.5 Modeling the "Behavior" of Hidden State

The language discussed in this chapter is sufficiently flexible to, on its own, model a form of hidden state. While other forms of hidden state exist (such as by considering an "anti-frame rule" [75]), our focus here is on modeling the "behavior" of hidden state *without* considering language extensions. Instead we exploit how state can be encapsulated within a closure.

In this scheme, stateful changes are expressed through usage constraints that enforce a specific *sequence of types*, the "behavior". By favoring encapsulation, clients rely on less detailed information about the inner state of objects. Changes to the underlying state are only indirectly noticeable by the changing sequence of operations that a type allows on a specific value. These *behavioral types* [21, 22, 24, 61, 65] offer a complementary view to typestates by focusing on hidden state, as opposed to handling abstract states.

We now exemplify how a type akin to a simple behavioral type can be encoded in our language. Recall that, when typing a functional value, we allow the function type to capture in its context a set of linear resources (Δ).

$$\text{(T:Function)}\quad \frac{\Gamma \mid \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta \vdash \lambda x : A_0.e : A_0 \multimap A_1 \dashv \cdot}$$

The following behavioral encoding will exploit that capture of resources to hide the captured resources from clients. Thus, the code will thread the underlying resources without letting clients directly "see" those resources. For this example we will revisit the pair object of Section 2.4.2.

```
open <EL,<ER,<L,<R, pair>>> = newPair({}) in                          EL, ER
  // build's behavior for Pair object
  λ r : int. // 1st, initialize right component                       Δ = EL, ER
    {
    pair.initR(r),                                                    Δ = EL, R
    λ l : int. // 2nd, initialize left component                      Δ = EL, R
      {
      pair.initL(l),                                                  Δ = L, R
      λ _ : ![] . // 3rd, sum both components                         Δ = L, R
        {
```

```
        pair.sum({}),                                          Δ = L, R
        λ _ : ![] .(pair.destroy({}))                          Δ = ·
        }
      }
    }
end
```

Each function returns a pair value with the result of the specific operation over the Pair object, and the next "behavior". This encodes a simple form of behavior where the remainder of the "usage protocol" is bundled together with the return of the function, yielding the following type:

$$\overbrace{\textbf{int} \multimap [\ ![]\ ,\ \overbrace{\textbf{int} \multimap [\ ![]\ ,\ \overbrace{![] \multimap [\ \textbf{int}\ ,\ \underbrace{![] \multimap ![]}_{\texttt{destroy}}\ ]}^{\texttt{sum}}\ ]}^{\texttt{initL}}\ ]}^{\texttt{initR}}$$

We use immutable pairs[3] to express the result of a function and the next behavior/type that must be respected. In contrast to the typestate style interface given above, this "behavioral type" does not just abstract, but rather completely hides the underlying capabilities of the stateful pair. The above type describes only one usage for using the object, others could have been assigned instead (for example initializing the left element of the pair before the right element).

Using a more natural annotation, we can highlight the behavior encoded above as enabling the following sequence of operations on the underlying Pair object:

$$\texttt{initR}\ ;\ \texttt{initL}\ ;\ \texttt{sum}\ ;\ \texttt{destroy}$$

This behavioral type expresses that the value initially (only) offers `initR`. After that call returns, the type then allows the use of `initL`, and so on. After `destroy` is invoked, the value no longer has any remaining behavior, meaning the type finished its usage. Such a behavioral type completely hides the underlying capabilities of that state, favoring instead to expose a fixed sequence of function calls that threads the relevant capabilities completely transparently to clients.

Mixing behavior and typestate Typestates and behavioral types offer complimentary views of the same phenomena. With typestates, the states are named, which can be a convenient abstraction, especially when there are multiple possible paths though the typestate/usage protocol. With behavioral types, the states are implicit, which simplifies the description of linear usages and makes it easy to provide structural equivalences. In theory, the two formalisms are interchangable and have similar expressiveness, so which one is preferred depends on the details of the particular situation.

Fortunately, the choice between typestate and behavioral types does not need to be fixed as, in our system, we can have a typestate object go through a "behavioral phase" and back. With such a scheme we can hide the state temporarily in a behavioral type and then later return to using abstract typestates—allowing the exchange or mixing of the two different code modularity approaches.

[3] Of the form "$[A, A]$", which can be encoded into the core language as shown in Section 2.3.

We illustrate this expressiveness by once again revisiting the pair example. However, in here we use an alternative encoding of behavior by storing the remaining behavior in a auxiliary cell (similar to how the this pointer works in most object-oriented languages) so that the result of each function is unobstructed by the required continuation of behavior, and more closely resembles object-based behavioral types. Likewise, for clarity, the usage of the object is encoded in a record so that the next behavior is accessible by field selection and clearly identified by that field name.

In our illustration, the definition of the `newPair` function is unchanged, and returns an object with a state-based type. The client code—which knows nothing of the implementation details of the pair—then creates a behaviorally-typed wrapper for the object. Although the wrapper is similar to the one shown above, it places the remaining "behavior" in the auxiliary `this` cell, as follows:

```
open < EL, < ER, < L, < R, pair > > > > > = newPair({}) in
  // 'this' reference to store the next behavior
  open <t,this> = new {} in
    // field names of record picked to help readability
    this := { // set next behavior
      initLeft = λ i : int :: ( rw t ![] ).
        let result = pair.initL(i) in
          this := {
            initRight = λ i : int :: ( rw t ![] ).
              let result = pair.initR(i) in
                this := { // note that it returns the captured capability
                  addBoth = λ _ : [] :: ( rw t ![] ).
                    ( delete this ; pair.sum({}) )
                };
                result
              end
          };
          result // result
        end
    };
    <t,(this:: (rw t initLType))>
in
... // client code that uses the behavior
```

The resulting value has type:

$$\exists t.(\ (!\mathbf{ref}\ t)\ ::\ (\mathbf{rw}\ t\ \texttt{initLType})\ )$$

where:

$$\begin{aligned}
\text{initLType} &\triangleq [\ \texttt{initLeft} : \mathbf{int} :: (\mathbf{rw}\ t\ ![]) \multimap ![] :: (\mathbf{rw}\ t\ \texttt{initRType})\ ] \\
\text{initRType} &\triangleq [\ \texttt{initRight} : \mathbf{int} :: (\mathbf{rw}\ t\ ![]) \multimap ![] :: (\mathbf{rw}\ t\ \texttt{addType})\ ] \\
\text{addType} &\triangleq [\ \texttt{addBoth} : ![] :: (\mathbf{rw}\ t\ ![]) \multimap \mathbf{int} :: (\mathrm{L} * \mathrm{R})\ ]
\end{aligned}$$

Note how the typestates of the original Pair object are only returned after the specific sequence of behaviors is completed. Therefore, with this type, we support a client that is more specific in the kind of uses it makes of the state, but oblivious to how those uses are correlated with the object's typestates. In fact, by adding a polymorphic type we can express a behavioral type entirely in terms

of the functions that are called, so that the final resulting typestate (U) is an abstracted type:

$$\forall U.(\ \exists t.(\ (!\mathbf{ref}\ t)\ ::\ (\mathbf{rw}\ t\ \texttt{initLType})\ )\ )$$

$$\begin{aligned}
\texttt{initLType} &\triangleq [\ \texttt{initLeft} \ :\ \mathbf{int} :: (\mathbf{rw}\ t\ ![]) \ \multimap\ ![] :: (\mathbf{rw}\ t\ \texttt{initRType})\ ] \\
\texttt{initRType} &\triangleq [\ \texttt{initRight} \ :\ \mathbf{int} :: (\mathbf{rw}\ t\ ![]) \ \multimap\ ![] :: (\mathbf{rw}\ t\ \texttt{addType})\ ] \\
\texttt{addType} &\triangleq [\ \texttt{addBoth} \ :\ ![] :: (\mathbf{rw}\ t\ ![]) \ \multimap\ \mathbf{int} :: \texttt{U}\ ]
\end{aligned}$$

This type allows clients to be abstract over the resulting capability U, while only needing to concern themselves with obeying the sequence of types that is encoded in such behavioral type. For instance, it enables the following client code:

```
<U>λ o : ∃t.( (!ref t) :: (rw t initLType) ) . // polymorphic function on type U
  open <t,ptr> = o in
    (!ptr).initLeft(1);
    (!ptr).initRight(2);
    (!ptr).addBoth({})
  end
```

with maximum reuse since it only depends on that specific sequence of calls to be available, not on any particular number or kinds of typestates. Note that its return type would then be the capability U that is abstract, meaning it could be a set of capabilities, or even the result of combining several objects together to offer such behavior. Other practical applications of such behavioral generalization could be to model iterators that, once closed, return the (abstracted) capability to the collection from which they were extracted from without depending on specific kinds or numbers of abstract states of that collection.

This "digression" was meant to illustrate the expressiveness of our preliminary system. Next we discuss how the benefits of hidden state can be mapped back into abstract manipulations of (type)states—even if considering language extensions. Intuitively, similar flexibility can be achieved by instead extending the core language with mechanisms to automatically recombine or mix abstract states.

2.5.1 Closer Comparison with "Behavior"-Oriented Designs

In prior work [61, 65] we developed an implementation of a "behavior"-oriented type system (called *yak*). We expanded on ideas from using behavioral types to model mutable state [21, 22] but implemented our system in a Java-like language, while eyeing more practical uses. This familiarity and prior experience with this kind of behavioral type systems gives us a privileged position to compare the design trade-off of those systems with the system develop in this document. The following discussion aims to better motivate our work, while clarifying the relation of the two systems and the capabilities of our design choices. Note that our focus here is on the major design differences rather than discuss minor "stylistic" choices (such as whether resources should be linear or affine, how copies are tracked and with what syntax, etc.).

While the general notion of a behavioral type embodies concepts that were also applied in other domains (such as process calculus), the discussion in here is focused on using behavioral types to model the properties of mutable state.

Recall that the "behavior" modeled by a behavioral type is essentially a form of hidden state, that re-interprets the concept of a typestate to one where state must be hidden and encapsulated within an object. Because typestates are hidden, stateful conditions cannot be expressed directly in abstract (type)states and instead are expressed by changing the interface of that object in order to reflect the state changes of its (internal) typestate. This kind of typing was also referred to as exposing a "dynamic interface" [43] to reflect this nonuniform, changing nature of the available methods of the object. [24] classified the distinction between explicit state designs as "state-based" and hidden state designs as "transition-based" by seeing the object's typestate specification in terms of a state machine with (type)states and transitions (labeled functions). Note, however, that both interpretations have equivalent typing granularity. Since a program essentially transitions program states through a fixed set of "atomic" reductions; reasoning about a program through that program's reductions or that program's states is basically a matter of preference.

Intuitively, the underlying typing constraints modeled by a behavioral type or a typestate are equivalent. This enables a straightforward translation between the typing constraints imposed by the two formats by simply exposing the (abstract) state, or by sequencing the type while hiding that state. For instance, and in very simple terms, we can have a typestate of the form:

$$\exists A, B.(\ ![\ \ \texttt{init}:\ ![] :: A \multimap \textbf{int} :: B,\quad \texttt{done}:\ ![] :: B \multimap\ ![]\ \ ] :: A\ )$$

where A and B are the typestates that abstract the state of the object represented by the record that contains two functions, `init` and `done`. If instead we were to hide such typestates, then the resources modeled by those typestates would be encapsulated within the record's function. This means that the use of the record must now become linear (since it captured linear resources in its scope) and, furthermore, we must hide the non-relevant functions until the internal resources reach the desired typestate. We can model those constraints at the object/module level, while hiding the internal typestate, with the following behavioral type:

$$[\ \texttt{init}:\ ![] \multimap \textbf{int}\ ]\ ;\ [\ \texttt{done}:\ ![] \multimap\ ![]\ ]$$

which can be further shortened to highlight the behavior of the type as: "`init` ; `done`" meaning that the type first allows a call to `init` and only after that call (;) will it enable the use of `done`. While the formal rules that enable this translation are beyond the scope of this discussion, the basic intuition should be straightforward to grasp.

In the following discussion, we "dissect" some of the core distinctive components of the behavioral model. We illustrate how several notions developed within the domain of behavioral types for checking mutable state map back to existing concepts of explicit state models.

Structural Equivalence In the example above, we showed that the notion of behavior, with its use of encapsulated/hidden state, can facilitate showing structural equivalences between stateful types. Since all behavioral types hide their typestates, the remaining type information is basically a sequence of functions describing each of the object' usages—regardless of the underlying typestate. The observation was that this common, uniform, basic representation underlying all behavioral types caused by the encapsulation of state would lead to an advantage of this model towards checking structural equivalences between two stateful types.

Although we do believe that a behavioral type provides a more concise type syntax in the corner case of a strictly linear stateful use, it turns out that showing structural equivalences within the typestate setting can be just as straightforward. The key observation is that the state recombination implicitly modeled by behavioral types can be modeled explicitly by expressing the underlying manipulation of state via standard abstraction mechanisms. Thus, provided that these typestates are typing artifacts with no underlying supporting value representation (as will be done in later chapters), structural equivalence is simply a matter of allowing subtyping rules to enable recombining typestates. Therefore, by pushing capabilities and abstraction to typing artifacts, we can enable the "re-interpretation"/"re-combination" of abstract capabilities seamlessly, such as via existential packages. For instance, the following structural equivalence (via subtyping):

$$\exists X.\exists Y.\exists Z.(\ [\ \texttt{init} : [] :: \underline{X * Z} \multimap [] :: Y\ ]\ ) \quad <: \quad \exists W.\exists Y.(\ [\ \texttt{init} : [] :: \underline{W} \multimap [] :: Y\ ]\ )$$

relies on abstracting "$X * Z$" under a new typestate and discarding the unused variables (as they no longer occur in the type). Similarly, we could also add "weakening" subtyping rules to add "vacuous" typestates to match the pre/post typestates of other types:

$$[] \multimap [] \quad <: \quad \exists A.(\ [] :: \underline{A} \multimap [] :: \underline{A}\ )$$

Therefore, from the perspective of structural equivalency both systems have equivalent expressiveness. However, behavioral types generally rely on the implicit recombination of state that occurs "under the hood" of the module/object layer that hides the typestate. Our design requires explicit subtyping rules to offer such recombination of abstract states, but these rules generally follow standard concepts of type abstraction—even if some more complex equivalences may also be defined. In a way, a behavioral type "pre-prepares" a type for computing structural equivalences while a typestate only does it on a need-driven basis.

Behavioral Borrowing Because state is hidden, notions such as borrowing of state become more challenging when using behavioral types as clients cannot directly refer which state they expect to borrow and get back. Instead, behavioral types require indirect reasoning about the internal borrowed state via the observable set of behavior that is to be used by clients.

In *yak* we developed the notion of *behavioral borrowing* to enable behavioral types to be "sliced" in two. Part of the behavior was then borrowed by the scope of a function, leaving the remaining ("residual") behavior of the object to only be valid after that function returns.

For instance, we can assign the following behavioral type to an object:

$$\texttt{init}\ ;\ \texttt{read}\ ;\ \texttt{done}\ ;\ \texttt{bye}$$

such that the object is then passed on as an argument to the following function type:

$$(\ \texttt{init}\ ;\ \texttt{read}\ )\ \multimap\ \texttt{int}$$

to mean that the function will use the behavior "`init ; read`" of the object given as parameter in its scope, thus leaving the borrowed object with the residual type "`done ; bye`" available only after

that function returns. In other words, behavioral borrowing enables the behavior of the function over the argument to be simulated statically:

$$\texttt{init}\ ;\ \texttt{read}\ ;\ \texttt{done}\ ;\ \texttt{bye} \xrightarrow{\texttt{init}\ ;\ \texttt{read}} \texttt{done}\ ;\ \texttt{bye}$$

To enable more complex forms of this "slicing" of behavior, Caires and Seco [24] added variable qualification to parts of the behavior in order for a behavioral type to express and compose borrowing at the level of variables (instead of just at the function-level) and threads. Similarly, to enable capture of components of behavior, they include a "$\circ T$" type to mean that the behavior T can be isolated—enabling that behavior to be stored, etc. In *yak*, a (much) simpler form of this isolation was expressed using the owned keyword. owned simply meant that that behavior was to be captured by the function and, thus, could be stored or returned. Qualification explicitly attaches behavior to a local variable, so that for instance we can get a type "(x : init ; done) ; (y : bye)" to mean that variable y's "bye" behavior only becomes available after x is used as "init" and "done".

Both the notion of "slicing" the behavior to be borrowed and the need to distinguish between isolated and non-isolated types are captured in a direct and uniform way once the underlying resources are named and made explicit. Thus, in our system, borrowing is simply returning the resources that were supplied as argument, and isolated types simply state that the resources will not be returned. Consequently, the same degree of expressiveness can be achieved for free without the need to introduce special variants of borrowing by simply expressing the underlying resource-typing concerns explicitly:

$$\forall X, Y, Z.(\ (\ [\ \texttt{init} : [] :: X \multimap [] :: Y,\ \texttt{read} : [] :: Y \multimap [] :: Z\ ] :: X\ ) \multimap \textbf{int} :: Z\ )$$

for borrowing, or omitting the return of Z for the isolated case.

Similarly, qualification of behavior is (indirectly) modeling the underlying aliasing of locations which can be captured in a more generic way via location-dependent types. Due to the lack of a proper way to quantify the qualified types, [24]'s use of qualification is limited to lexcially-scoped uses. While the type's syntax can become simpler by leaving the equivalent location quantification implicit, it requires significantly more complex and non-standard machinery. Likewise, the imposed aliasing conditions are overly restrictive as they freeze which variable must wait for a resource to be available. In our system, all variables simply wait for a resource to be returned making such kind of type specification more flexible (i.e. we wait until an abstract state "re-appears" versus waiting until a very specific sequence of variables is fully used).

Our past experience implementing a very simple notion of behavioral borrowing showed that this kind of borrowing can become very complex rather quickly. While borrowing small, linear behavior is generally manageable, borrowing anything more complex generally resulted in hard to foresee "split" types. While the expressiveness of both kinds of borrowing specifications appear equivalent, the overhead of behavioral borrowing appears to be higher. At its core, behavioral borrowing requires a form of simulation to step the behavior of the object that lends, to match the expected usages that will be taken by the borrowing context. In our opinion, mixing this kind of simulation with complex type constructs can result in very non-obvious borrowing results/errors. In contrast, typestate-like borrowing is simply a matter of replacing the types of the argument's resources with those of the result since the states are modeled directly and explicitly.

Global Usage Behavioral types rely on a "global specification" of the object's behavior since, by design, the programmer is not allowed to explicitly state the local pre/post state conditions of a function (the design is centered on hiding the object's state information). However, because the global behavior is detached from the function's local uses, the type checker must ensure that the two are coherent so that the global specification is consistent with the individual function's uses. This means that the object-level behavioral specification must be used to guide the type checker in essentially inferring the local uses of the object's internal state done through each of its functions. To do this, the type checker must traverse the global behavioral specification and "glue" together the individual behavioral uses done by each individual method of that object (which we called the "*consistency check*" phase in *yak*), to ensure that the used internal behavior of each method is compatible with the global behavioral specification declared for that object.

In contrast, typestate specifications are completely local by having explicit pre/post conditions describe the expected type of the object's internal state. Therefore, typestates do not need to traverse an object's global specification as the local specification suffices to characterize the object's stateful usages. However, this means that typestates do not avoid "disconnected" usages while the behavioral check above will have the consequence of ensuring that the object's state machine is a connected graph.

Although behavioral types essentially employ a mechanism that is similar to type inference to check the object's internal behavior, it is difficult to judge if there is a decreased annotation burden since these annotations may need to re-appear elsewhere to ensure decidability. Unfortunately, at the time of this writing, it appears that only our own (and very much outdated) prototype is publicly available and its expressiveness is no match to compare to modern explicit-state systems. Likewise, a more detailed comparison would have to do so by comparing to an equivalent typestate inference system with the one used in behavioral types to compose local behaviors.

Interestingly, because typestates are purely local this will enable not only local reasoning from the programmer (independently of the necessary context described by the global behavioral specification) and enable complex code composition (such as inheritance) by simply replacing a function while still obeying the same typestate pre/post conditions. To achieve similar change in the behavioral types case would require recomputing the inferred global internal behavior.

From the practicality standpoint, an error when checking the consistency of the global behavior may be the result of either the programmer assuming the wrong context when they defined a method, or a wrong global behavior. In our experience, even without inheritance, errors in composing local behavior quickly become overwhelming if it is not the corner case of very simple and strictly linear behavior.

Perhaps the most important distinction is that a typestate is automatically the most generic use of that object, while a behavioral type requires the programmer to worry about and explicitly state which functions should be available next. The complexity of this design choice is noted in [24] as most examples do not use the most general behavioral type possible for a given object.

While using a "global" type may be useful to capture additional contextual information if the "local" types lack a way to model such expressiveness, we did not find evidence of examples of behavioral types that exploited this additional contextual information in any meaningful way—in our view, making the verification overhead and specification complexity hard to justify.

Final Remarks Our overall impression is that the design choice of omitting state causes several problems related to the indirect manipulation of the underlying hidden state—problems that are solved essentially for free when the programmer is simply allowed to reason directly about state.

In our opinion, by hiding state, behavioral types also denies a useful reasoning mechanism to the programmer with no apparent benefit from such a design choice; in fact, it actually requires significantly *more* effort to implement and use the behavioral approach. The programmer will often need to try to understand from the context the reason as to why a method is unavailable. For instance, for us, being told that `divide` is unavailable until `increment` is called is somewhat more contrived than explaining that the state may be zero and consequently using `divide` is potentially not safe. While in simple, small, linear examples this may not be all that difficult, when considering more practical usages (such as the doubly linked list in the TR of [24]) such behavioral type quickly becomes overwhelming. In contrast, typestates enable the information on how a function uses the (type)state and the current typestate to be decoupled. Thus, regardless of how complex the object state manipulations are or how many operations it offers, the current typestate information remains concise and generally more manageable to understand. Furthermore, in our experience and understanding of work on the behavioral types, we did not identify any technical advantage of the behavioral approach over the explicit state approach, beyond at most stylistic syntactical differences. From that stylistic perspective, our work takes a more "bare-bones" low-level, precise modeling of state that favors composing these generic low-level blocks to build higher-level abstractions. Consequently, our design does not address the issue of how to best hide or present more complex mechanisms such as location-dependent types, which may naturally result in some syntactical burden.

Although both approaches are already quite complex, behavioral types often incur in extra verification overhead by introducing or requiring additional "single-purpose" reasoning methods that are unnecessary when the underlying state manipulations are modeled explicitly. Therefore, we did not find a compelling reason to use the behavioral model as any perceived expressiveness advantaged can be reduced to reasoning about explicit (even if abstract) state manipulations within a fairly standard language core. In our opinion, behavioral types face a significant challenge not only in matching the level of clarity/elegance of comparable explicit state designs, but also in offering a competitive verification overhead for practical uses.

2.6 Related Work

We now discuss closely related work. The discussion on the handling of interference is postponed to later chapters. Instead we focus on giving an overview and research context on the different techniques to check the use of mutable state. To avoid some repetition and keep the text focused, we omit some work that only becomes relevant once interference is considered.

The goal of avoiding erroneous uses of mutable state has spawn a wide range of research work, concurrently developed under a vast set of different concepts. Strom proposed *typestate checking* [85] to associate a finite-state grammar with a variable's type. The compiler tracks all typestate changes, ensuring that a program obeys the specified typestate declaration. Later work

by Strom and Yemini [84] further refined and clarified the typestate concept, enabling typestates to determine the subset of operations allowed within a particular program context (which includes variable initialization, deallocation, etc.). Other contemporary works also make use of protocol-like concepts, but at a higher-level, to offer some degree of safety. For instance, PROCOL [93] associates a state machine with an object to ensure safe communication between multiple objects in an object-oriented language. Similarly, *Regular Types* [70] views object state changes as a problem of nonuniform service availability, defining protocols to model those changes in types that track how objects interact. Yet support for aliasing remained challenging [52] since aliased state may break local tracking of (type)state changes. Subsequent works focus on mechanisms to not only track stateful resources but also constraining the permissions each alias has to access shared resources.

Wadler's *Linear Types* [94] use linearity [45] to track resources, ensuring that linear types are used only once (therefore, are "unique" and can never be aliased). To relax the linear requirement, Wadler also proposes a "let!" block to temporarily, but safely, enable multiple reads of the same cell in a scoped block where writes to that cell are forbidden until uniqueness is restored. More pragmatic approaches, generalize the notion of linearity/uniqueness to that of different *permission* kinds that govern how (aliased) mutable state can be accessed. Managing the permission-flow [16, 17, 18, 19, 67] becomes of crucial importance as permissions are split ("copied") and merged to match different sets of aliases or may be temporarily borrowed to be used within a well-defined block of code before their guaranteed return to the lending program context. Technically, only the work in later chapters will enable more flexible sharing akin to what permissions allow. In this chapter our language only supports local aliasing, where location-dependent types can be used to enable aliasing whenever the type system is able to track which variables alias some same state.

An alternative approach to permissions uses capabilities for memory management [30, 83, 95, 96]. We highlight the work of Ahmed, Fluet, and Morrisett in $\mathbf{L}^3$ [5] as we use $\mathbf{L}^3$ as a foundation for our core language. $\mathbf{L}^3$ builds on concepts from earlier work on *alias types* [83, 95] in the way that it expresses aliasing information within the types. However, $\mathbf{L}^3$ not only proposes a crucial separation of pure references from the respective linear (and first-class) capability, but also develops its formalization within the framework of a linear type system. By focusing on reasoning about type-changing mutations ("*strong-updates*"), $\mathbf{L}^3$ targets a lower-level of abstraction, with manually-threaded capabilities and no mechanism for abstraction beyond location variables. However, their core system addresses concerns of program termination, and invariant-based sharing (which is only comparable to our sharing mechanism shown in later chapters). Our development expands on basic $\mathbf{L}^3$ concepts, extending them to a type-and-effect system (allowing for implicitly threaded capabilities, reducing the language's syntactic burden) and supporting additional practical type expressiveness (such as using sum types, etc.). However, our own "flavor" of $\mathbf{L}^3$ does not address concerns of program termination (while other more recent uses of $\mathbf{L}^3$ ideas [57] do). Alternative designs use affine tracking (a weakening of linear), such as in *Alms* [90] which supports, among other things, manually threaded capabilities to express a variety of resource-aware idioms. They design an expressive kind system to distinguish between affine and non-affine types (as others also use [60]) that is able to restrict how types may or may not be copied, guaranteeing that affine types (such as capabilities) are preserved and that they interact safely with non-affine types,

including supporting kind polymorphism. Unlike *Alms*, by following $\mathbf{L}^3$ we enable the distinction between reference and capability to not require a module intermediary while providing closer (or clearer) links to the underlying concepts from linear logic (instead of indirectly through the kind system).

Fähndrich and DeLine pioneered more practical uses of linear types in *Vault* [32, 33, 39], including enabling typestate verification for objects. They introduced novel aliasing control mechanisms such as adoption and focus [33, 39] to enable aliasing of stateful resources that can temporarily be restored to linear when "focused" (a technique that was later further improved in [20]), and the notion of `pack`/`unpack` [32, 33] to distinguish when an object is or not consistent with its internal (typestate) invariant, ensuring that inconsistent states should not be publicly visible. Bierhoff et al. [10, 11, 13, 15] further improved the practicality of these typestate systems by expanding the set of available permission kinds, and supporting object-oriented features such as inheritance. *Masked types* [77] adapts the typestate idea into a system that is specially targeted to solve the problem of object initialization, while allowing complex pointer topologies such as cyclic dependencies (including inheritance/dispatch) to remain safe. Our work was heavily influenced by prior typestate work, although we implemented our ideas within a more low-level type-theoretic framework. This enables us to model typestates directly using existential types, but our language lacks support for more advanced object-oriented features such as method dispatch and inheritance. We believe these OO-features are orthogonal to our main topic, interference control, enabling our core language to be simpler.

More recently, the *Plaid* [7, 67, 86] language explored the idea of typestate-oriented programming where the language design employs typestates as first-class entities. Their design enables a cleaner separation and specification of typestate changes since individual objects are characterized by their single typestate invariant, as opposed to having a single object carrying multiple typestates. First-class state changes are then used to explicitly transition between different typestates. They also showed how each typestate object can be composed using trait-based inheritance, enabling their language to model several practical use cases within their language paradigm. Our particular language design takes a more type-theoretic, low-level approach overlooking more user-level design concerns. We build typestate abstractions by composing and abstracting individual uses of reference cells rather than requiring explicit language support for typestate. However, we recognize that better support for common idioms and convenient composition abstractions can make a language more practical and perhaps more attractive to developers. Indeed, it remains an open problem to understand what is the best way to expose our language constructs to programmers and in ways that would facilitate clear, easily extensible program designs.

In earlier work, we developed a notion of *view* typestate [62] as a way to decompose and recombine smaller components of an object (called "views") and handle aliasing by assigning views to aliases. Splitting and merging were governed by explicit *view equations* that defined a basic algebra for recombining views. In *Capable* [28] Castegren and Wrigstad follow an analogous idea but developed well beyond our simpler, basic language. They support a larger array of recombination operators and include support for parallel programming. While their expressiveness is not on par with $\mathbf{L}^3$-based systems, they target a simpler programming paradigm that is claimed to still capture relevant idioms. Our core language retains some (very basic) recombination mechanism

somewhat resembling our initial work on view typestates. We showed that our notion of views can map to more standard notions of separation and abstraction. In this system, view equations must be expressed via explicit functions that recombine those abstract typestates in ways that are hidden from clients, but that must be explicitly done by the programmer. Our sharing mechanism of later chapters adds support for more expressive splittings, beyond simple separation, by allowing protocols to represent overlapping uses of the same state rather than just disjoint separation.

Mandelbaum *et al.* [59] use an alternative modeling of state through type refinements that provide descriptions on the effects a computation has on the state. For instance, function arguments have two components to describe both the conventional type and a logical formula that describes the state. This technique resembles a generalized (through refinement types) way to model the same separation of references and capabilities as we use. Our approach is focused on aliasing conditions, leaving the treatment of type refinements as future work. Their approach is unable to express sharing in the same extent as we will show in later chapters since they must preserve the link between resources and forbid (coordinated) duplication of linear resources.

In [71] Parkinson and Bierman introduce the notion of *abstract predicates* which enriches a logical framework with abstraction over higher-order predicates whose representation is only known inside a module, and use it to prove functional properties of programs. Developments of *Hoare Type Theory* [68, 69] also include support for higher-order predicates and abstraction within the framework of an expressive dependent type theory. Later work by Krishnaswami, *et al.* [57] integrated linear and dependent types, also includes support for this form of abstraction. Abstract predicates encode a similar notion to typestates that is, however, not limited to a finite number of abstract states, since predicates can be parametric on some variables. Consequently, typestates generally target a more "lightweight" verification. We intentionally target this simpler notion of abstraction, and leave for future work support for more expressive forms of abstraction, since our focus is mainly on interference control.

In [21, 22] Caires developed a notion of "spatial-behavioral types" to model resources by sequencing their usages. This work was used as the basis for our own prior work on using behavioral types for modeling mutable state in an object-oriented language. That work led to the implementation of an approach to verify "behavior" [61, 65], that basically enables an analogous kind of linear typestate-like checking while hiding state. In this work we introduced the notion of "behavioral borrowing", to enable limited aliasing of a behavioral object for the scope of a method call, and developed an implementation of the technique. Concurrently with that work, Gay *et al.* [43] modeled (object) protocols through the use of *session types* [89], both locally and in a distributed environment, by generalizing the notion of session types (normally associated with nonuniform service availability) to include typestate constraints on how method calls to an object can be sequenced. However, their work forbids all forms of aliasing but employs a similar "global usage" protocol to constrain how objects can be used. Later work by Caires and Seco [24], following ideas from process algebras, introduce the notion of "behavioral separation" where behavioral types are used to model complex usage protocols with the possibility of aliasing and where separation ensures safety in the use of aliased state. By allowing types to express local variables (through qualification) aliasing is allowed whenever the types can express the threading behavior of usages. However, qualification still limits aliasing to be lexically-scoped as there appears to be no way to

quantify the use of qualification within types. Therefore qualification does not reach the same level of expressiveness as an $\mathbf{L}^3$-based design, although it enables their work to employ simpler syntax to express local aliasing.

Because our own prior work fits within these notions of "behavior", we included in Section 2.5.1 a more detailed comparison with this kind of language design and the incurred trade-offs. Overall, in our opinion and experience, the design choice of hiding state causes several expressiveness problems that require "single-purpose" methods (such as behavioral borrowing, behavioral isolation, etc.) in attempts to circumvent these expressiveness limitations. However, these additional mechanisms increase the verification complexity, introducing significant overhead considering that similar expressiveness is essentially achieved for free on comparable typestate systems. Furthermore, we did not find any evidence of an expressiveness advantage of using hidden state. Intuitively, any manipulation of hidden state can be translated and expressed just as well by using abstract explicit manipulations of the underlying state. Consequently, we believe that the behavioral approach is not competitive with comparable typestate systems—leading us to develop our system within a more standard and flexible $\mathbf{L}^3$-based design.

Chapter 3

Rely-Guarantee Protocols in the Sequential Setting

The preliminary system introduced in chapter 2 uses linear capabilities that must be threaded linearly throughout a program. Consequently, there is only one capability to access a cell and that capability cannot be duplicated, limiting the language's expressiveness. For instance, any alias to a location will lose access to that location once the capability of that location is packaged, as in:

```
Γ = x : ref p , y : ref p , p : loc                                Δ = rw p int
  x := false;                                                      Δ = rw p boolean
  <p,y:: (rw p boolean)>; // typed as ∃t.((!ref t) :: (rw t boolean))   Δ = ·
  !x // Error: Missing capability to location 'p'
```

Once the capability to p is packed, any access to that location by prior aliases becomes illegal. Even opening the package does not restore access since the package type refers a fresh location that is unrelated to its prior name (p).

The most straightforward solution to this expressiveness problem is to simply impose, once again, an invariant type over that shared state. Instead, our technique offers a more precise control over the interactions that can occur through shared, mutable state by modeling those interactions using a protocol type. Our protocols model the interference that may be produced through sharing and ensures that the different protocols to one same location can be safely composed.

This chapter introduces the three main concepts of our sharing mechanism: 1) how protocols are specified; 2) how protocols are used; and, 3) how protocols are composed. We do this development within the sequential setting, leaving the concurrent setting to chapter 4.

Interference can also occur (at the type-level) in the sequential setting, due to encapsulation. When access to shared state is encapsulated or packaged, aliases appear to access different locations (at the type checking level) giving the programmer sufficient "leeway" to interleave their uses in ways that are not deterministic or not discernible to the type checker. Consequently, the type checker must ensure safety regardless of how those uses are actually interleaved—even in the sequential setting. The result is that our mechanism is generally "agnostic" to how interference may be produced, up to a few technical details, since both sequential and concurrent forms of interference are rooted at how aliases may be interleaved. Any type safe program must be resilient

to all possible interference over shared state, concurrency will simply increase the set of possible run-time interleaving that may occur—but a safe program is already required to account for all "alias interleaving" even in the smaller run-time interleaving set of the sequential setting. Therefore, we can start by showing how interference is controlled in the simpler sequential setting. The presentation follows analogous structure to that of our published paper [64] but using an axiomatic definition of protocol composition that was developed later.

3.1 Approach in a Nutshell

The interactions that can occur via shared mutable state can be a source of program errors. When different variables mutate the same shared state that is used by separate components, their actions can potentially *interfere*. If left uncontrolled, this interference may cause a variable to assume that the wrong type is stored in that shared cell. When that happens, the program may fault due to the unexpected, *unsafe*, interference that was caused by sharing state.

Crucially, sharing introduces the possibility of non-obvious (or even non-deterministic) interleaving of aliases to some same shared state. Therefore, for shared variables, the type system must ensure correctness regardless of when other aliases to one same shared state are used. While in concurrent systems such uncertainty in interleaving is linked to non-deterministic thread scheduling, a similar phenomenon can also occur in the sequential setting at the type-level.

For this reason, we consider state to be shared whenever the type system cannot precisely track that different variables alias the same state, even if used by the same thread. In that situation the exact interleaving/scheduling of the uses of those aliases cannot be statically known. Although the execution of a program can be deterministic, the imprecision in typing means that the programmer has sufficient freedom to pick different ways to interleave aliases. To ensure safety, the type system must require that a program is correct regardless of when those aliases are actually interleaved. This requirement effectively ensures correctness for *all* possible alias interleavings—regardless of how many of those different interleavings may actually happen at run-time.

Interference due to aliasing is analogous to the interference caused by thread interleaving [49, 98]. This occurs because mutable state may be shared by aliases encapsulated in unknown or non-local program contexts. Such a boundary effectively negates the use of static mechanisms to track exactly which other variables alias some state. Therefore, we are unable to know precisely if the shared state aliased by a local variable will be used when the execution jumps off (e.g. through a function call) to non-local program contexts. However, if that state is used, then the aliases may change the state in ways that invalidate the local alias's assumptions on the current contents of the shared state. This interference caused by "alias interleaving" occurs even without concurrency, but is analogous to how thread interleaving may affect shared state. Consequently, techniques to reason about thread interference (such as *rely-guarantee reasoning* [55]) can be useful to reason about aliasing even in our sequential setting. The core principle of rely-guarantee reasoning that we adapt is its mechanism to make strong local assumptions in the face of interference. In our system, to handle such interference, each alias has its actions constrained so that the resulting state fits within a *guarantee* type and at the same time is free to assume that the state changes done by

other aliases of that state must fit within a *rely* type. The duality between what aliases can rely on and must guarantee among themselves yields significant flexibility in the use of shared state, when compared for instance to invariant-based sharing.

Our rely-guarantee protocols are formed by a sequence of rely-guarantee steps. Each step contains a rely type, stating what an alias currently assumes the shared state contains; and a guarantee type, a promise that the changes done by that alias will fit within this type. Using these primitive building blocks, our technique allows strong local assumption on how the shared state may change, while not knowing when or if other aliases to that shared state will be used—only how they will interact with the shared state, if used. Since each step in a protocol can have distinct rely and guarantee types, a protocol is not frozen in time and can model different "temporal" uses of the shared state directly. A protocol is, therefore, an abstracted local perspective on the actions done by each individual alias to the shared state, and that is only aware of the potential resulting effect the other aliases may produce on that shared state. A *protocol composition* mechanism ensures the sound collaboration of all protocols to the same shared state, at the moment of their creation. From there on, each protocol is "stable" (known to be free of unsafe interference) since protocol composition attested that each protocol, in isolation, is aware of all observable effects that may occur from all possible "alias interleaving" originating from the remaining aliases.

Our technique is able to capture the following features:

1. Each protocol provides a *local* type so that an alias need not know the actions that other aliases are doing, only their resulting (observable) effect on the shared state;
2. Sharing can be done *asymmetrically* so that the role of each alias in the interaction with the shared state may be distinct from the rest;
3. Our protocol paradigm is able to *scale* by modeling sharing interactions both at the reference level and also at the abstract state level. Therefore, sharing does not need to be embedded in an ADT [56], but can also work at the ADT level without requiring a wrapper reference [49];
4. State can be shared individually or simultaneously in groups of state. By enabling sharing to occur underneath a layer of apparently disjoint state, we naturally support the notion of *fictional disjointness* [34, 54, 56];
5. Our protocol abstraction is able to model complex interactions that occur through the shared state. These include invariant, monotonic and other coordinated uses. Moreover, they enable both *ownership transfer* of state between non-local program contexts and *ownership recovery*. Therefore, shared state can return to be non-shared, even allowing it to be later shared again and in such a way that is completely unrelated to its previous sharing phases;
6. Although protocol composition is checked in pairs, *arbitrary aliasing* is possible (if safe) by further sharing a protocol in ways that do not conflict with the initial sharing. Therefore, global safety in the use of the shared state by multiple aliases is assured by the composition of individual binary protocol splits, with each split sharing the state without breaking what was previously assumed on that state;
7. We allow *temporary inconsistencies*, so that the shared state may undergo intermediate (private) states that cannot be seen by other aliases. Using an idea similar to (static) mutual exclusion, we ensure that the same shared state cannot be inspected while it is inconsistent.

```
1  let newPipe = λ _ : ![] .                                  Γ = _ : []                                  | Δ = ·
2   open <n,node> = new Empty#{} in                           Γ = _ : [], node : ref n, n : loc | Δ = rw n Empty#![]
3    share (rw n Empty#![]) as H[n] || T[n];                                                  Γ = ... | Δ = T[n], H[n]
4    open <h,head> = new <n, node:: H[n]> in
                                                  Γ = ..., head : ref h, h : loc | Δ = T[n], rw h ∃p.((!ref p) :: H[p])
5     open <t,tail> = new <n, node:: T[n]> in
                                                  Γ = ..., tail : ref t, t : loc | Δ = rw t ∃p.((!ref p) :: T[p]), ...
6      < rw h exists p.((!ref p) :: H[p]), // packs a type, the capability to location 'h'
7      < rw t exists p.((!ref p) :: T[p]), // packs a type, the capability to location 't'
8      { // creates a labeled record with 'put', 'close' and 'tryTake' as members
9           put = λ e : ( int :: ( rw t exists p.((!ref p) :: T[p]) ) ).⁄*...*⁄,
19        close = λ _ : ( ![] :: ( rw t exists p.((!ref p) :: T[p]) ) ).⁄*...*⁄,
26      tryTake = λ _ : ( ![] :: ( rw h exists p.((!ref p) :: H[p]) ) ).⁄*...*⁄
47     } :: ( (rw h ∃p.((!ref p) :: H[p])) * (rw t ∃p.((!ref p) :: T[p])) ) > >
48    end
49   end
50  end
```

Figure 3.1: The pipe example. Some implementation details are omitted (and only shown further below) but stacking is shown explicitly in blue.

> This kind of critical section, which does not incur in any run-time overhead, is sufficiently flexible to support multiple simultaneously inconsistent states—when these states are sure to not be aliasing the same shared state, and while avoiding re-entrant uses of the same state.

We now build on the core system shown in the previous chapter by extending that language to support sharing through the use of our rely-guarantee protocols. We also show soundness through the use of standard progress and preservation theorems that show that all allowed interference is safe. These theorems ensure that a program cannot get stuck, while still allowing the shared state to be legally used in complex ways via different aliases.

3.2 Pipe Example Overview

We now revisit the pipe example that was briefly discussed in Section 1.1. We will use the pipe example to introduce our ideas by showing how pipes can be modeled as rely-guarantee protocols that ensure safe interaction between aliases of some shared cells. Figure 3.1 shows part of the pipe's code while hiding some details of the implementation of the `put`, `close`, and `tryTake` functions that will only be shown later.

Pipes are used to support a *consumer-producer* style of interaction (using a shared internal buffer as mediator), often used in a concurrent program but here used in a single-threaded environment. The shared internal buffer is implemented as a shared singly-linked list where the consumer keeps a pointer to the *head* of the list and the producer to its *tail*. By partitioning the pipe's functions (where the consumer alias uses `tryTake`, and the producer both `put` and `close`), clients of the pipe can work independently of one another, provided that the functions' implementation is aware of the potential interference caused by the actions of the other alias. It is on specifying and verifying this interference that our rely-guarantee protocols will be used.

The function in Figure 3.1 creates a pipe by allocating an initial `node` for the internal buffer, a cell to be shared by the `head` and `tail` pointers. The newly allocated cell (line 2) contains a tagged (as `Empty`) empty record ({}). As discussed in the previous chapter, the new construct (line 2) is assigned a type that abstracts the fresh location that was created by new, "$\exists l.($ (!**ref** l) :: (**rw** l `Empty#![]`))". On the same line, we open the existential by giving it a location variable n and a regular variable `node` to refer that reference. From there on, the capability is automatically unstacked and moved implicitly as needed through the program. Recall that, for clarity, we manually stack capabilities although the type system does not require it. For instance, on line 4, we use the construct "$e :: A$" where A is the stacked capability (written in blue).

To circumvent the linear restriction on the use of the capability, we must share that capability by splitting it into rely-guarantee protocols. Therefore, on line 3 we share the capability to n by splitting it in two rely-guarantee protocols, H and T[1]. Each protocol is then assigned to the `head` and `tail` pointers (lines 4 and 5, respectively), since they encode the specific uses of each of those aliases. The protocols and the details of our sharing mechanisms will be introduced in Section 3.3.

The type of `newPipe` is a linear function ($\multimap$) that, since it does not capture any enclosing linear resource, can be marked as pure (!) so that the type can be used without the linear restriction. On line 6 we pack the inner state of the pipe (so as to abstract the capability for `t` as `P`, and the one for `h` as `C`), resulting in `newPipe` having the type:

$$!(\ ![] \multimap \exists C.\exists P.(\ ![\ \ldots\ ] :: (\ C * P\ )\ )\ )$$

where the separate capabilities for the Consumer and Producer are stacked together in a commutative group ($*$). In this type, `C` abstracts the capability "**rw** `h` $\exists p.((!\textbf{ref}\ p) :: H[p])$", and `P` abstracts "**rw** `t` $\exists p.((!\textbf{ref}\ p) :: T[p])$".

Although we have not yet shown the implementation, the type of the elided record ([...]) contains function types that should be unsurprising noting that each argument and return type has the respective capabilities for the `head`/`tail` cells stacked on top. Thus, those functions are closures that use the knowledge about the reference to the `head`/`tail` pointers from the surrounding context, but do not capture the capability to those cells and instead require them to be supplied as arguments when calling the function.

```
![ put      :  !( int :: P  ⊸  ![] :: P ),
  close    :  !( ![] :: P  ⊸  ![] ),
  tryTake  :  !( ![] :: C  ⊸  NoResult#(![] :: C) + Result#(int :: C) + Depleted#![] )  ]
```

We have that `put` preserves the producer's resource, but `close` destroys `P`. The result of `tryTake` is a sum type of either `Result` or `NoResult` depending on whether the still open pipe has or not contents available, or `Depleted` to signal that the pipe was closed (and therefore that the resource `C` has vanished).

Observe that the state that the functions depend on is, apparently, disjoint although underneath this layer the state is actually shared (but coordinated through a protocol) so that (safe) interference must occur for the pipe to work correctly—i.e. it is *fictionally disjoint* [34, 54, 56].

[1] As a brief glimpse, T is "(**rw** n `Empty#![]`) $\Rightarrow$ ((**rw** n `Node#R`) $\oplus$ (**rw** n `Closed#![]`)) ; **none**" which relies on n containing "`Empty#![]`", ensures n then contains either "`Node#R`" or "`Closed#![]`", and then loses access to n. Both "$\Rightarrow$" and ";" (and R) will be discussed in detail in the next Section.

3.3 Using Protocols to Share Mutable State

The goal is to enable reads and writes to a cell through multiple aliases, without requiring the type system to precisely track the link between aliased variables. In other words, the type system is aware that a variable is aliased, but does not know exactly which other variables alias that same state. In this scenario, it is no longer possible to implicitly move capabilities between aliases. Instead, we split the original capability into multiple *protocols* to that same location, and ensure that these multiple protocols cannot interact in ways that unsafely interfere with each other. A *rely-guarantee* protocol accounts for the effects of other protocols (the *rely*), and limits the actions of this protocol to *guarantee* that they do not contradict the assumptions relied on by other aliases. This allows independent, but constrained, actions on the different protocols to the same shared state without unsafe interference. However, it also requires us to leverage additional type mechanisms to ensure safety, namely:

Hide intermediate states. A rely-guarantee protocol restricts how aliases can publicly use shared resources. However, we allow this specification to be temporarily broken provided that all unexpected changes are private, invisible to other aliases. Therefore, the type system ensures a kind of static mutual exclusion, a mechanism that provides a "critical section" with the desired level of isolation from other aliases to that same state. Consequently, other resources that may overlap with the resources currently being inspected will simply become unavailable while those resources are undergoing private changes. Although our solution is necessarily conservative, we are able to avoid any run-time overhead while preserving many relevant usages.

To achieve this, we build on the concept of focus [39] (in a non-lexically scoped style, so that there is also a defocus) clearly delimiting the boundary in the code of where shared state is being inspected. Thus, on focus, all other resources that may directly or indirectly see inconsistencies must be temporarily concealed only to reappear when those inconsistencies have been fixed, on defocus.

In this chapter, we employ a sequential semantics which results in "automatic" assurances of mutual exclusion at run-time as each step of the operational semantics is essentially executed in isolation. Consequently, focus and defocus are only needed by the type checker and have no operational meaning. Later, in the concurrent setting, this assurance of mutual exclusion disappears as multiple threads may be executing arbitrarily interleaved code. In that setting, we must add some operational meaning to focus/defocus such that they operate as traditional lock/unlock constructs.

Ensure that each individual step of the protocol is obeyed. In our system, sharing properties are encoded in a protocol composed of several rely-guarantee *steps*. As discussed in the previous paragraph, each step must be guarded by focus since private states should not be visible to other aliases. Consequently, the focus construct serves not only to safeguard from interference by other aliases, but also to move the protocol forward through each of its individual steps. At each such step, the code can assume on entry (focus) that the shared state will be in a given well-defined *rely* state, and must ensure on exit (defocus) that the shared state satisfies a given well-defined *guarantee* state. By characterizing the sequence of actions of each

$$
\begin{array}{rcll}
A & ::= & \ldots & \\
 & | & A \Rightarrow A & \text{(rely)} \\
 & | & A \; ; \; A & \text{(guarantee)}
\end{array}
$$

Figure 3.2: Extension to the types grammar of Figure 2.9.

alias with an appropriate protocol, one can make strong local assumptions about how the shared state is used without any explicit dependence on how accesses to other aliases of that shared state are interleaved. This feature is crucial since we cannot know precisely if that same shared state was used between two focus-defocus operations, as the static information linking variables together (through a location variable) may no longer be available.

The remainder of this subsection will present the core mechanisms of our technique by using the previously omitted code of the pipe example as a guide. We start by presenting how protocols can be specified, followed by how two protocols's composability is verified and checked by the share construct when introducing protocols. Finally, we show how protocols can be used through focus and defocus constructs, and our mechanism to avoid re-entrant uses of currently inconsistent resources.

3.3.1 Specifying Rely-Guarantee Protocols

We now extend the type grammar of chapter 2 with our rely and guarantee types (shown in Figure 3.2) that are used as the basic building blocks of our rely-guarantee protocols.

A rely-guarantee protocol is a type of capability (i.e. is a typing artifact without a value) consisting of potentially many steps, each of the form "$A_C \Rightarrow A_P$". Each such step states that it is safe for the current client to assume that the shared state satisfies A_C, and that the client is required to obey the guarantee A_P. The guarantee A_P, usually of the form "$A'_C; A'_P$", requires the client to establish (guarantee) that the shared state satisfies A'_C, before allowing the protocol to continue to be used as A'_P. Note that our design constrains the syntactic structure of these protocols through *protocol composition* (Section 3.3.2), and not directly via the grammar. This means that the grammar allows the programmer to write a larger number of protocol types than what our type system legally allows a program to use. To simplify our system, we allow the programmer to write "bogus" protocols such as "**int** $\Rightarrow$ **int**" that can never be introduced, since that protocol disobeys our composition conditions.

In this chapter, abstractions such as "$\forall X.(\, X \Rightarrow X \,)$" are only legal *after* the protocols are known to safely compose. Since during that composition check we must look into the protocol's structure, abstractions could hide relevant interference information. The next chapter details mechanisms to check protocol composition even when (some) abstraction is considered. In here we simplify composition by restricting the syntax of our protocols at the composition stage so that they cannot abstract or refer abstracted shared resources or steps. To clarify the type structure of our protocols, we define the following sub-grammar of our type grammar (A) with the types that may legally appear in a protocol, P, such that its safe composition can be checked.

$$P \;::=\; (\textbf{rec}\; X(\overline{u}).P)[\overline{U_P}] \;\mid\; X[\overline{U_P}] \;\mid\; P \oplus P \;\mid\; P \,\&\, P \;\mid\; R \Rightarrow P \;\mid\; R\,;\, P \;\mid\; \textbf{none}$$

where R is also a sub-grammar of A containing the types of A that denote resources that can be shared via a protocol, and are not themselves a protocol. In this chapter, it suffices to consider R as the following grammar:

$$R \ ::= \ (\textbf{rec}\ X(\overline{u}).R)[\overline{U_R}] \ \mid \ X[\overline{U_R}] \ \mid \ R \oplus R \ \mid \ R \ \& \ R \ \mid \ R * R \ \mid \ \textbf{rw}\ p\ A \ \mid \ \textbf{none}$$

Note the intentional "**rw** p A" so that other protocols may be nested inside the capability (in A), but not grouped ($*$) together with other *R*esources during protocol composition. Also note U_P and U_R are restrictions on U to use P or R (instead of A), respectively.

Pipe's protocols We can now define the protocols for the shared list nodes of the pipe's buffer. Each node follows a rely-guarantee protocol that includes three possible tagged states: `Node`, which indicates that a node contains some useful data; `Empty`, which indicates that the node will be filled with data by the producer (but does not yet have any data); and finally `Closed`, which indicates that the producer has sent all data through the pipe and that no more data will be added (thus, it is the last node of the list).

Remember that the producer component of the pipe has an alias to the tail node of the internal list. Because it is the producer, it can rely on that shared node still being `Empty` (as created) since the consumer component will never be allowed to change that state. The rely-guarantee protocol for the tail alias (for some location p) is as follows:

$$(\textbf{rw}\ p\ \texttt{Empty\#![]}) \Rightarrow (\ (\textbf{rw}\ p\ \texttt{Node\#R}) \ \oplus \ (\textbf{rw}\ p\ \texttt{Closed\#![]})\ )\ ;\ \textbf{none}$$

This protocol expresses that the client code can safely assume (on focus) a capability stating that location p initially holds type "`Empty#![]`". It then requires the code that uses such state to leave it (on defocus) in one of two possible alternatives ($\oplus$) depending on whether the producer chooses to close the pipe or insert a new element into the buffer. To signal that the node is the last element of the pipe, the producer can just assign to that location a value of type "`Closed#![]`". Insertions are slightly more complicated because that action implies that the tail element of the list will be changed. Therefore, after creating the new node, the producer component will keep an alias of the new tail for itself while leaving the old tail with a type that is to be used by the consumer. In this case, the node is assigned a value of type "`Node#R`", where R denotes the type "[int , $\exists p.$((!**ref** p) :: H[p])]" (a pair of an integer and a reference to the next shared node of the buffer, as seen from the head pointer). Regardless of its action, the producer then forfeits any ownership of that state which is modeled by the empty resource (**none**)[2] to signal protocol termination.

We now present the abbreviations H and T, the rely-guarantee protocols that govern the use of the shared state of the pipe as seen by the `head` and `tail` aliases, respectively. Both abbreviations define a type that is parametric in a location, p, since we will be using these protocols over each individual node of the shared list. The types are defined as follows:

$$\begin{array}{lcl} \mathsf{T}[p] & \triangleq & \mathsf{E} \Rightarrow (\ \mathsf{N} \ \oplus \ \mathsf{C}\ ) \\ \mathsf{H}[p] & \triangleq & (\ \mathsf{N} \Rightarrow \textbf{none}\ ) \ \oplus \ (\ \mathsf{C} \Rightarrow \textbf{none}\ ) \ \oplus \ (\ \mathsf{E} \Rightarrow \ (\ \mathsf{E}\ ;\ \mathsf{H}[p]\ )\ ) \end{array}$$

[2]We frequently omit the trailing "; **none**" for conciseness.

where N is an abbreviation for a capability that contains a node "**rw** *p* `Node#R`", C is "**rw** *p* `Closed#![]`", and E is "**rw** *p* `Empty#![]`".

The T type was presented above, so we can now look in more detail to H. Such a protocol contains three alternatives, each with a different action on the state. If the state is found with an E type (i.e. still `Empty`) the consumer is not to modify such a state (i.e., just reestablish E), and can retry again later to check if changes occurred. Observe that the remaining two alternatives have a **none** guarantee. This models the recovery of ownership of that particular node. Since the client is not required to reestablish the capability it relied on, that capability can remain available in the client context even after defocus—as opposed to require clients to "return" the capability to the protocol.

Each protocol describes a partial view of the complete use of the shared state. Consequently, ensuring their safety cannot be done alone. In our system, protocols are introduced explicitly through the share construct, which declares that a type (in practice limited to resources, which includes capabilities and protocols) is to be split into two new rely-guarantee protocols. Safety is checked by simulating their actions in order to ensure that they preserve overall consistency in the use of the shared state, no matter how their actions may be interleaved. Since a rely-guarantee protocol can subsequently continue to be re-split, this technique does not limit the number of aliases provided that the protocols compose safely.

3.3.2 Checking Protocol Splitting

The key principle of ensuring a correct protocol split is to verify that both protocols consider all visible states that are reachable by stepping, ensuring a form of progress. Protocols are not required to always terminate and may be used indefinitely, for instance when modeling invariant-based sharing. However, regardless of interleaving or of how many times a shared alias is (consecutively) used, no unexpected state can ever appear in a well-formed protocol. Thus, the type information contained in a protocol is valid ("stable") regardless of when the specified interference may occur.

We now discuss the technical details of checking protocol composition. Intuitively, a binary protocol split will generate an infinite binary tree representing all possible combinations of interleaved uses of the two new protocols. Each node of that tree has two children based on which protocol remains stationary while the other is stepped. Since this tree may be infinite, a correct split must build a co-inductive proof of safe interference that shows that the two protocols compose correctly.

Protocol composition ensures that a resource, S (a capability or protocol), can be shared ("split") into two protocols, P and Q, noted:

$$S \Rrightarrow P \parallel Q$$

We use a set of *configurations*, C, to represent the positions of each protocol as we simulate all possible interleaved uses of the two new protocols, where C is defined as:

$$\begin{array}{rcll} C & ::= & \langle\, S \Rrightarrow P \parallel Q \,\rangle & \text{(configuration)} \\ & | & C \cdot C & \text{(union)} \end{array}$$

$\boxed{C \mapsto C}$ **Composition, (C:*)**

$$\frac{\langle\, S \Rrightarrow \mathcal{R}_L[P] \,\rangle \mapsto C_0 \qquad \mathcal{R}_L[\square] = \square \parallel Q \qquad \langle\, S \Rrightarrow \mathcal{R}_R[Q] \,\rangle \mapsto C_1 \qquad \mathcal{R}_R[\square] = P \parallel \square}{\langle\, S \Rrightarrow P \parallel Q \,\rangle \mapsto C_0 \cdot C_1}\ (\text{C:Step}) \qquad \frac{C_0 \mapsto C_2 \qquad C_1 \mapsto C_3}{C_0 \cdot C_1 \mapsto C_2 \cdot C_3}\ (\text{C:AllStep})$$

Composition — Reduction Step, (C-RS:*)

$$\frac{}{\langle\, S \Rrightarrow \mathcal{R}[\textsf{none}] \,\rangle \mapsto \langle\, S \Rrightarrow \mathcal{R}[\textsf{none}] \,\rangle}\ (\text{C-RS:None})$$

$$\frac{\langle\, S_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \qquad \langle\, S_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_1}{\langle\, S_0 \oplus S_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \cdot C_1}\ (\text{C-RS:StateAlternative}) \qquad \frac{\langle\, S \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C}{\langle\, S \Rrightarrow \mathcal{R}[P_0 \oplus P_1] \,\rangle \mapsto C}\ (\text{C-RS:ProtocolAlternative})$$

$$\frac{\langle\, S_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, S_0 \& S_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}\ (\text{C-RS:StateIntersection}) \qquad \frac{\langle\, S \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C_0 \qquad \langle\, S \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C_1}{\langle\, S \Rrightarrow \mathcal{R}[P_0 \& P_1] \,\rangle \mapsto C_0 \cdot C_1}\ (\text{C-RS:ProtocolIntersection})$$

Composition — State Stepping, (C-SS:*)

$$\frac{}{\langle\, R_0 \Rrightarrow \mathcal{R}[R_0 \Rightarrow R_1; P] \,\rangle \mapsto \langle\, R_1 \Rrightarrow \mathcal{R}[P] \,\rangle}\ (\text{C-SS:Step})$$

Composition — Protocol Stepping, (C-PS:*)

$$\frac{}{\langle\, (R_0 \Rightarrow R_1; Q) \Rrightarrow \mathcal{R}[R_0 \Rightarrow R_1; P] \,\rangle \mapsto \langle\, Q \Rrightarrow \mathcal{R}[P] \,\rangle}\ (\text{C-PS:Step})$$

Recall the grammar restrictions for checking safe composition:

$$\begin{array}{rcl} P, Q & ::= & (\textbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \mid X[\overline{U_P}] \mid P \oplus P \mid P \,\&\, P \mid R \Rightarrow P \mid R\,;\, P \mid \textsf{none} \\ R & ::= & (\textbf{rec}\ X(\overline{u}).R)[\overline{U_R}] \mid X[\overline{U_R}] \mid R \oplus R \mid R \,\&\, R \mid R * R \mid \textbf{rw}\ p\ A \mid \textsf{none} \\ S & ::= & R \mid P \end{array}$$

Also recall that our recursive types are automatically unfolded, through (EQ:Rec) meaning that stepping rules omit any such unfolding that may occur.

Figure 3.3: Protocol composition, stepping rules.

(c-rs:Subsumption)
$$\frac{S_1 <: S_0 \qquad \langle\, S_0 \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C \qquad P_0 <: P_1}{\langle\, S_1 \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C}$$

(c-ss:Step)
$$\frac{R_0 <: R_1}{\langle\, R_0 \Rrightarrow \mathcal{R}[R_1 \Rightarrow R_2; P] \,\rangle \mapsto \langle\, R_2 \Rrightarrow \mathcal{R}[P] \,\rangle}$$

(c-ps:Step)
$$\frac{R_0 <: R_1 \qquad R_3 <: R_2}{\langle\, R_0 \Rightarrow R_2; Q \Rrightarrow \mathcal{R}[R_1 \Rightarrow R_3; P] \,\rangle \mapsto \langle\, Q \Rrightarrow \mathcal{R}[P] \,\rangle}$$

Figure 3.4: Changes to the definition of protocol composition to consider subtyping.

A well-formed protocol split requires that the two protocols compose safely. Protocol composition ensures that all configurations reachable through stepping are themselves able to take a step:

(wf:Split)
$$\frac{\langle\, S \Rrightarrow P \parallel Q \,\rangle\uparrow}{S \Rrightarrow P \parallel Q} \qquad \text{(wf:Configuration)} \quad \frac{C_0 \mapsto C_1 \qquad C_1\uparrow}{C_0\uparrow}$$

Where $C\uparrow$ signals the *divergence* of stepping, consistent with the co-inductive nature of protocol composition. Recall that we use a double line, as in (wf:Configuration), to mean that a rule is to be interpreted co-inductively. This definition accounts for protocols that never terminate and also ensures that all protocols can take a step with a given resource. To be consistent with this definition, a terminated protocol (**none**) simply "spins" without changing the state of the shared resource.

We now go over the rules of Figure 3.3 that step configurations. Note that S ranges over both protocol and non-protocol types, allowing rules such as (c-rs:StateAlternative) to be used to step a shared (non-protocol) *R*esource or an existing *P*rotocol, without needing to duplicate this rule.

(c:AllStep) synchronously steps all existing configurations, where each configuration is stepped through (c:Step). We use $\mathcal{R}_*$ to specify the $\mathcal{R}$eduction context of one of the protocols of a configuration, while the remaining protocol remains stationary, i.e.:

$$\begin{array}{lcll} \mathcal{R}_L[\square] & = & \square \parallel Q & \text{(for the } L\text{eft protocol, } Q \text{ is stationary)} \\ \mathcal{R}_R[\square] & = & P \parallel \square & \text{(for the } R\text{ight protocol, } P \text{ is stationary)} \end{array}$$

The subsequent stepping rules use $\mathcal{R}$ to range over both $\mathcal{R}_L$ and $\mathcal{R}_R$.

We use three distinct label prefixes to group the stepping rules based on whether a rule is stepping over a protocol (c-ps:*), stepping over a state (i.e. a non-protocol) (c-ss:*), or is applicable in both cases (c-rs:*). (c-rs:None) "spins" a configuration since a terminated protocol cannot use the shared state but must be stuck-free for consistency with our definition. The following (c-rs:*Alternative) and (c-rs:*Intersection) rules dissect the state or protocol based on the alternative ($\oplus$) or choice ($\&$) presented. Each different alternative state must be individually considered

by a protocol, while only one alternative step of a protocol needs to be valid. The situation is the reverse for choices: all choices of a protocol must have a valid step, but a step of a protocol can choose which state to consider when stepping. The rule for state stepping, (c-ss:Step), transitions the step of the protocol and changes the state to reflect the guarantee state of the protocol. Protocol stepping rules, (c-ps:*), are used to simulate an existing protocol over some newly split protocol. They follow similar design to state stepping, but require a matching simulation of the rely and guarantee types of both protocols, as shown in (c-ps:Step). Protocol stepping is used to specialized or re-split an existing protocol. Therefore, the interference produced by the new protocol must be consistent with those produced by the original protocol.

Subtyping Extension We now extend stepping to consider subtyping. By using the rules of Figure 3.4, we can make stepping of state or stepping of a simulated protocol more flexible by considering subtyping on a protocol's rely and guarantee types. All rules follow similar intuition: a protocol can rely on a weaker state and guarantee a stronger state. This will ensure that subtyping cannot invalidate old assumptions. For instance in (c-rs:Subsumption), we may consider a weaker state (S_0) and a more precise protocol (P_0) when stepping.

Thus, the protocol may consider fewer steps as long as they are still enough to check all states of the shared resource. Since the premise ensures that safe composition is respected with stronger conditions, the conclusion remains sound.

Merging Protocol composition is defined as a "split", left-to-right ($\Rrightarrow$). Simply reading the rules as right-to-left ($\Lleftarrow$) to compute a "merge" is not safe. For instance, it would enable merging to arbitrary choices with (c-rs:StateIntersection). Intuitively, merging needs to intertwine the uses of both protocols. However, since we do not track copies we lose the information that two split protocols are actually aliasing the same shared state. Consequently, merging cannot "collapse" a protocol into a non-protocol resource. Thus, "merging" would be equivalent to simply having the two non-merged protocols available in Δ, making it unnecessary to include a procedure for merging protocols.

Extending Protocols In this chapter, the only form of specialization that we allow is by extending an existing ownership recovery step, essentially appending a new protocol to a step that would otherwise terminate the protocol.

Therefore, if we have a protocol step ($R' \Rightarrow \textbf{none} \; ; \; \textbf{none}$) that recovers ownership of the shared resource (R'), then we can append (noted $\bowtie$) to that protocol some other protocol (Q) that shares resource (R) such that the split "$R \Rrightarrow Q \parallel \textbf{none}$" is valid.

$$(\, R' \Rightarrow \textbf{none} \; ; \; \textbf{none} \,) \;\; \bowtie \;\; Q \;\; = \;\; R' \Rightarrow R \; ; \; Q$$

The append operation simply attaches Q to all terminating steps of that protocol, extending the use of that protocol while leaving the previous steps unchanged. Note that, if $R' = R$ then we can simplify the step extension to simply replace that step directly with Q.

In the next chapter we will use a more seamless way to extend protocols by automatically switching from protocol stepping rules to state stepping rules, when appending new uses to a

terminating step. In here, however, the design choice of enabling ownership recovery via a "$\Rrightarrow$ **none**; **none**" step leads to the slightly more complicated implementation of these protocol extensions discussed above.

Example We now illustrate how protocol composition works by going back to the pipe's protocols. We introduce the protocols for the head and tail aliases through the share construct, as shown on line 3:

```
3   share (rw n Empty#![]) as H[n] || T[n];
```

where share is checked by the (T:SHARE) typing rule, using protocol composition, as follows:

$$\text{(T:SHARE)}\quad \frac{A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \mid \Delta, A_0 \vdash \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2 : ![] \dashv \Delta, A_1, A_2}$$

With share we can split a resource (A_0) to be shared by two new protocols (A_1 and A_2) whose individual roles in the interactions with that shared resource safely compose ($\Rrightarrow$). Consequently, the conclusion states that if such a splitting is correct, then in some linear typing environment initially consisting of resource A_0 and resources Δ, the share construct produces effects that replace A_0 with the two new protocols A_1 and A_2 but leaves Δ unmodified (i.e. Δ is just threaded through).

We now show how protocol composition checks that the split above is correct. For conciseness we use the abbreviations without stating the location that they are using. Thus, we use E as an abbreviation for "**rw** *n* `Empty#![]`" and analogous abbreviations for C and N. Likewise, we abbreviate the name of the `Head` and `Tail` protocols to just `H` and `T`. The use of the share construct on line 3 yields the following split:

$$\mathsf{E} \Rrightarrow \mathtt{H} \parallel \mathtt{T}$$

that results in the following set of configurations:

$$\begin{array}{l}\{\ ❶\langle\ \ \mathsf{E}\ \Rrightarrow\ (\mathsf{N} \Rightarrow \mathbf{none}) \oplus (\mathsf{C} \Rightarrow \mathbf{none}) \oplus (\mathsf{E} \Rightarrow (\mathsf{E}\ ;\ \mathtt{H}))\ \parallel\ \mathsf{E} \Rightarrow ((\mathsf{N} \oplus \mathsf{C})\ ;\ \mathbf{none})\ \rangle,\\ \ \ ❷\langle\ \mathsf{N} \oplus \mathsf{C}\ \Rrightarrow\ (\mathsf{N} \Rightarrow \mathbf{none}) \oplus (\mathsf{C} \Rightarrow \mathbf{none}) \oplus (\mathsf{E} \Rightarrow (\mathsf{E}\ ;\ \mathtt{H}))\ \parallel\ \mathbf{none}\ \rangle,\\ \ \ ❸\langle\ \mathbf{none}\ \Rrightarrow\ \mathbf{none}\ \parallel\ \mathbf{none}\ \rangle\ \ \}\end{array}$$

The split above results in the initial configuration ❶. If we step the left protocol (`H`) of configuration ❶, we see that we must remain in configuration ❶ since only step "$\mathsf{E} \Rightarrow \mathsf{E}\ ;\ \mathtt{H}$" expects resource E.

$$\frac{\dfrac{}{\langle\ \mathsf{E} \Rrightarrow \mathcal{R}[\ \mathsf{E} \Rightarrow \mathsf{E}\ ;\ \mathtt{H}\ ]\ \rangle\ \mapsto\ \langle\ \mathsf{E} \Rrightarrow \mathcal{R}[\ \mathtt{H}\ ]\ \rangle}\ \text{(C-SS:STEP)}}{❶\langle\ \mathsf{E} \Rrightarrow \mathcal{R}[\ \mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\ ;\ \mathtt{H}\ ]\ \rangle\ \mapsto\ \langle\ \mathsf{E} \Rrightarrow \mathcal{R}[\ \mathtt{H}\ ]\ \rangle ❶}\ \text{(C:PROTOCOLALTERNATIVE)}$$

On the other hand, stepping the right protocol (`T`) will change the type of the shared resource to "N $\oplus$ C" resulting in configuration ❷, that also signals that `T` has terminated (i.e. becomes **none**).

$$\frac{}{❶\langle\ \mathsf{E} \Rrightarrow \mathcal{R}[\ \mathsf{E} \Rightarrow (\ \mathsf{N} \oplus \mathsf{C}\ )\ ;\ \mathbf{none}\ ]\ \rangle\ \mapsto\ \langle\ \mathsf{N} \oplus \mathsf{C} \Rrightarrow \mathcal{R}[\ \mathbf{none}\ ]\ \rangle ❷}\ \text{(C-SS:STEP)}$$

From ❷, we separate the $\oplus$ type of the resource and step each individual type with the respective step in H. This will yield configuration ❸ as both steps consume the shared resource and terminate the H protocol.

$$\dfrac{\dfrac{\vdots}{\text{❷}\ \langle\ \mathsf{N} \oplus \mathsf{C} \Rrightarrow \mathcal{R}[\ \mathtt{H}\ ]\ \rangle\ \mapsto\ \langle\ \textbf{none} \Rrightarrow \mathcal{R}[\ \textbf{none}\ ]\ \rangle\ \text{❸}}\ \text{(C-RS:StateAlternative)}}{\langle\ \mathsf{C}\ \Rrightarrow \mathcal{R}[\ \mathtt{H}\ ]\ \rangle\ \mapsto\ \langle\ \textbf{none} \Rrightarrow \mathcal{R}[\ \textbf{none}\ ]\ \rangle}\ \text{(C-RS:ProtocolAlternative)}$$
$$\dfrac{}{\langle\ \mathsf{C}\ \Rrightarrow \mathcal{R}[\ \mathsf{N} \Rightarrow \textbf{none}\ ;\ \textbf{none}\ ]\ \rangle\ \mapsto\ \langle\ \textbf{none} \Rrightarrow \mathcal{R}[\ \textbf{none}\ ]\ \rangle}\ \text{(C-SS:Step)}$$

No new configurations can be produced when stepping ❸ since all protocols have terminated, and we can close the co-inductive proof.

Regardless of how the use of the state is interleaved at run-time, the shared state cannot reach an unexpected (by the protocols) state. Thus, protocol composition ensures the stability of the type information contained in a protocol in the face of all possible "alias interleavings". Protocols, as defined in this chapter, contain only a finite number of possible (relevant) states, meaning that it suffices for protocol composition to consider the smallest set of configurations that obeys the protocol composition rules above, up to subtyping. Since there must also exist a finite number of possible way to interleave the two protocols, the result of mixing the steps of the two protocols must also be a finite number of distinct (relevant) steps. Effectively, protocol composition resembles a form of bisimulation or model checking (where each protocol is modeled using a graph) with a finite number of states, ensuring composition remains tractable.

Note that safe protocol composition is only respected if E is the resource to be shared by the two protocols. If instead we had shared, for instance, C we would get the next set of configurations:

$$\begin{array}{l}\{\ \text{❶}\langle\ \ \mathsf{C}\ \ \Rrightarrow\ \ (\mathsf{N} \Rightarrow \textbf{none}) \oplus (\mathsf{C} \Rightarrow \textbf{none}) \oplus (\mathsf{E} \Rightarrow (\mathsf{E}\ ;\ \mathtt{H}))\ \|\ (\mathsf{E} \Rightarrow (\ \mathsf{N} \oplus \mathsf{C}\ ))\ \rangle, \\ \ \ \text{❷}\langle\ \textbf{none}\ \ \Rrightarrow \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{none}\ \|\ (\mathsf{E} \Rightarrow (\ \mathsf{N} \oplus \mathsf{C}\ ))\ \rangle\ \ \}\end{array}$$

The set above does *not* satisfy our composition conditions, as not all protocols can take a step with resource C. Both the resource in configuration ❶ and **none** in configuration ❷ are not expected by the right protocol (T). Thus, those configurations are "stuck", as the two protocols may not be able to take a step. Although splittings are checked from a high-level and abstracted perspective, their consequences link back to concrete invalid program states that could occur if such invalid splittings were allowed. For instance, in ❷, the configuration would imply that the alias that used the right protocol (T) would assume E on focus long after the ownership of that shared resource was recovered by some other alias of that cell (that used protocol H). Consequently, allowing such usages could enable unsafe interference by allowing unexpected changes to be observed by the alias that recovered ownership, potentially resulting in a program stuck on some unexpected value.

3.3.3 Using Shared State

Access to shared state is delimited by two constructs: focus, that exposes the shared state contained in a protocol; and defocus, that returns the exposed state to the protocol. Checking these constructs is combined with our modified version of the frame rule that will only be discussed in Section 3.3.4.

We now describe how focus is checked through the (T:Focus-Rely) rule. In general, focus may be applied over a disjunction ($\oplus$) of program states and expected to work on any of those alternatives. By requiring focus to supply a list of types ($\overline{A}$), the programmer can list each type that may become available after focus, describing what they expect to gain by focus in a way that guides the type checker on which protocol to pick and focus on.

(T:Focus-Rely)

$$\frac{A_0 \in \overline{A}}{\Gamma \mid A_0 \Rightarrow A_1 \vdash \mathsf{focus}\ \overline{A} : ![] \dashv A_0,\ (A_1 \triangleright \cdot)}$$

The effect of focus results in a typing environment where the step of the protocol that was focused on ($A_0 \Rightarrow A_1$) has its rely type (A_0) available to use. However, it is not enough to just make A_0 available since that resource my be aliased and shared through other protocols. Therefore, we must also *hide* all other linear resources that may use that same shared state (directly or indirectly) in order to avoid interference due to the inspection of private states.

To express this form of hiding, the linear typing environments may include a *defocus-guarantee*. This element, written as $A \triangleright \Delta$, means that we are hiding the typing environment Δ until A is satisfied. In our system, the only meaningful type for A is a guarantee type of the form $A'; A''$ that is satisfied when A' is offered and enables the protocol to continue to be used as A''—which is exactly what the defocus typing rule will check.

Although the typing rule shown above only includes a single element in the initial typing environment (and, consequently, the defocus-guarantee contains the empty typing environment, $\cdot$), this is not a limitation. In fact, the full potential of (T:Focus-Rely) is only realized when combined with (T:Frame). Together they allow for the non-lexically scoped framing of potentially shared state, where the addition of resources that may conflict with focused state will be automatically nested inside the defocus-guarantee ($\triangleright$). Operationally share, focus, and defocus are no-ops which results in those expressions having type unit (![]).

(T:Defocus-Guarantee)

$$\frac{}{\Gamma \mid \Delta_0,\ A',\ ((A'; A'') \triangleright \Delta_1) \vdash \mathsf{defocus} : ![] \dashv \Delta_0,\ A'',\ \Delta_1}$$

The complementary operation, defocus, simply checks that the required guarantee type (A') is available. Because the inconsistency in the state of the shared resources was fixed (and the public protocol specification is now assured to be respected), the typing environment (Δ_1) that was hidden on the right of $\triangleright$ can now safely be made available to use once again. At the same time, that step of the protocol is concluded leaving the remainder protocol (A'') in the typing environment.

Nesting of defocus-guarantees is possible, but is only allowed to occur on the right of $\triangleright$. Note that defocus-guarantees can never be captured (such as by functions, as will be shown below with the modifications to the core language of chapter 2) and, therefore, pending defocus operations cannot be forgotten or ignored. The structure of the environments listed in a defocus-guarantee essentially express an order by which the inconsistencies were made, and by which order they should be fixed.

```
9  put = λ e : ( int :: rw t exists p.((!ref p) :: T[p]) ).
                                  Γ = ..., tail : ref t,  t : loc,  e : int | Δ = rw t ∃p.((!ref p) :: T[p])
10   open <l,last> = new Empty#{} in                Γ = ..., last : ref l,  l : loc | Δ = ...,  rw l Empty#![]
11     open <o,oldlast> = !tail in          Γ = ...,  oldlast : ref o | Δ = rw t ![],  rw l Empty#![],  T[o]
12       focus (rw o Empty#![]);    Δ = ...,  rw o Empty#![],  (rw o Node#R) ⊕ (rw o Closed#![]); none ▸ ·
13       share (rw l Empty#![]) as H[l] || T[l];                                  Δ = ..., T[l], H[l], ...
14       oldlast := Node#{ e, <l,last :: H[l]> };                                Δ = ..., rw o Node#R, ...
15       defocus;                                                                Δ = rw t ![], T[l], none
16       tail := <l, last :: T[l]>                                  Δ = rw t ∃p.((!ref p) :: T[p])
17     end
18   end,
19 close = λ _ : ( ![] :: rw t exists p.((!ref p) :: T[p]) ).
                                  Γ = ..., tail : ref t,  t : loc,  _ : [] | Δ = rw t ∃p.((!ref p) :: T[p])
20   open <l,last> = !tail in                     Γ = ..., last : ref l,  l : loc | Δ = rw t ![], T[l]
21     delete tail;                                                                          Δ = T[l]
22     focus (rw l Empty#![]);          Δ = rw l Empty#![],  (rw l Node#R) ⊕ (rw l Closed#![]); none ▸ ·
23     last := Closed#{};             Δ = rw l Closed#![],  (rw l Node#R) ⊕ (rw l Closed#![]); none ▸ ·
24     defocus                                                                             Δ = ·
25   end,
```

Figure 3.5: Implementation of `put` and `close` functions.

Example We now look at the implementation of the `put` and `close` functions (Figure 3.5) to exemplify the use of focus and defocus. Both functions are closures that capture an enclosing Γ where `t` is a known location such that `tail` has type "**ref** `t`". Recall that T[p] was defined above as:

$$(\mathbf{rw}\ p\ \texttt{Empty\#![]}) \Rightarrow ((\mathbf{rw}\ p\ \texttt{Node\#R}) \oplus (\mathbf{rw}\ p\ \texttt{Closed\#![]}))$$

where R is a pair of an integer and a protocol for the head, H (whose definition, also given above, is not important here).

The `put` function takes an integer stacked with a capability for `t`. The capability is automatically unstacked to Δ. Since we are inserting a new element at the end of the buffer, we create a new node that will serve as the new `last` node of that list. On line 11, the `oldlast` node is read from the `tail` cell by opening the abstracted location it contains. This location refers a protocol type, for which we must use focus (line 12) to gain access to the resources that the protocol shares. Afterwards, we modify the contents of that cell by assigning the new node to the cell. This node contains the alias for the new tail as will be used by the head alias. The T component of that split (line 13) is stored in the `tail`. The defocus of line 15 completes the protocol for that cell, meaning that the alias will no longer be usable through there. Carefully note that the share of line 13 takes place *after* focus. If this were reversed, then the type system would conservatively hide the two newly created protocols making it impossible to use them until defocus. By exploiting the fact that this capability is not shared, we can allow it to not be hidden inside ▸ since it cannot interfere with shared state. `close` should be straightforward to understand since it simply assigns a `Closed` node to the shared cell before forfeiting any further use through that cell.

3.3.4 Framing State

On its own, (T:FOCUS-RELY) is very restrictive since it requires a single rely-guarantee protocol to be the exclusive member of the linear typing environment. This happens because more complex applications of focus are meant to be combined with our version of the frame rule. Together, the two rules enable a kind of mutual exclusion that also ensures that the addition of any potentially interfering resources will necessarily be on the right of $\triangleright$, and thus inaccessible until defocus.

The framing rule is as follows:

$$\text{(T:FRAME)}\quad \frac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \circledast \Delta_2 \vdash e : A \dashv \Delta_1 \circledast \Delta_2}$$

Framing serves the purpose of hiding ("frame away") parts of the footprint (Δ_2) that are not relevant to typechecking a given expression (e), or can also be seen as enabling extensions to the current footprint. In our system, this operation is slightly more complex than traditional framing since we must also ensure that any such extension will not enable unsafe interference by allowing access to inconsistent resources.

We use a special operand, $\Delta_1 \circledast \Delta_2$, to filter out which of the resources in Δ_2 are allowed to be accessed, and which of those resources may alias inconsistent state in Δ_1 and must be placed "inside" (on the right of $\triangleright$) any existing defocus-guarantee of Δ_1. However, we can only make this (static) distinction somewhat conservatively by detecting types that are sure to not refer to shared state, which we say are **non-shared**. We define A **non-shared** to mean that the type A cannot include access to shared parts of the heap:

$$\frac{A \textbf{ non-shared}}{\textbf{rw } p\ A \textbf{ non-shared}} \qquad \frac{\overline{A_i \textbf{ non-shared}}}{\sum_i \mathtt{t}_i \# A_i \textbf{ non-shared}} \qquad \frac{}{!A \textbf{ non-shared}} \qquad \frac{}{\textbf{none non-shared}}$$

Alternative, perhaps more fine-grained, definitions of **non-shared** may exist but the one above suffices for our purposes. This means that all other linear types (including abstracted resources and linear functions) must always be assumed to be potential sources of conflicting interference. For instance, these types could be abstracting or capturing a rely-guarantee protocol that could then result in a re-entrant inspection of the shared state. Due to fictional separation, we are not able to track which resources may not overlap in their access to shared state, and must assume that any such resource may access the same heap space. Perhaps surprisingly, we can consider the type "**rw** $p\ A$" to be non-shared. Because capabilities are linear (there can only exist one), any duplication of a capability to one same location will violate our store typing definition. In that situation, framing will produce a set of resources that can never be instantiated into a real heap.

To build the extended typing environment, we define an *environment extension* operation that takes into account frame defocus-guarantees up to a certain depth. This means that one can always consider extensions of the current footprint as long as any added shared state is hidden from all

focused state. By conservatively hiding it behind a defocus-guarantee, we ensure that such state cannot be touched. This enables locality on focus: if a protocol is available, then it can safely be focused on.

Definition 1 (Environment Extension)**.** *Given environments* Δ *and* Δ' *we define environment extension, noted* $\Delta \circledast\!\!- \Delta'$*, as follows. Let*

$$\Delta = \Delta_n, \Delta_s \qquad\qquad \Delta' = \Delta'_n, \Delta'_s$$

where n-indexed environments only contain non-shared elements and s-indexed environments contain the remaining elements (i.e. all those that may, potentially, include sharing).

Extending Δ *with* Δ' *corresponds to:*

(a) if $\Delta_s = \Delta_{s_0},\ A \triangleright \Delta_{s_1}$ *then* $\Delta \circledast\!\!- \Delta' = \Delta_n,\ \Delta'_n,\ \Delta_{s_0},\ A \triangleright (\Delta_{s_1} \circledast\!\!- \Delta'_s)$.

(b) otherwise, $\Delta \circledast\!\!- \Delta' = \Delta_n,\ \Delta'_n,\ \Delta_s,\ \Delta'_s$.

that either (a) further nests the shared part of Δ' *deeper into* Δ_s*'s defocus-guarantee; or (b) simply composes* Δ' *with* Δ *if* Δ *does not carry any defocus-guarantee.*

The definition works just like regular environment composition when Δ does not contain a defocus-guarantee, i.e. the (b) case. For instance, we have that $(A, B) \circledast\!\!- (C, D) = A, B, C, D$. The complexity of the definition arises from the need to nest defocus-guarantees when they do occur in Δ, which results in the inductive definition above. In that situation, we must ensure that any potentially interfering shared state is placed deep inside all previously existing defocus-guarantees, so as to remain inaccessible. For instance, if we have $(A \triangleright B) \circledast\!\!- S$ where S is a shared resource, placing S next to A would (wrongly) enable direct access to S. Since we only know that S is a shared resource, we cannot guarantee that the state it represents is currently consistent with the agreed public definition of interference, since there is one pending guarantee. Instead, to be safe, our definition of environment extension enforces that the resulting environment is $A \triangleright (B, S)$ so that S is (conservatively) nested inside the existing defocus-guarantee. Thus, S will only be made available when the inconsistency (modeled by the defocus-guarantee) is corrected. This definition is compatible with the basic notion of disjoint separation, but (from a framing perspective) allows us to frame-away defocus-guarantees beyond a certain depth. Through (T:Frame) we see that such part of the state can be safely hidden if the underlying expression will not reach those resources (by defocusing).

Interestingly, the definition allows a (limited) form of *multi*-focus. For instance, while a defocus is pending we can create a new cell and share it through two new protocols. Then, by framing the remaining part of the typing environment, we can now focus on one of the new protocols. The old defocus-guarantee is then nested *inside* the new defocus-guarantee that resulted from the last focus. This produces a "list" of pending guarantees in the reverse order on which they were created through focus. Through framing we can hide part of that "list" after a certain depth, while preserving the purpose of the defocus-guarantee in guarding against potentially re-entrant uses of focused state.

Example We now look back at the focus of line 12 (in Figure 3.5). However, our example lacks the need to frame other shared resources since no other shared resource is present on that line 12. Therefore, to illustrate framing of other shared resources, we consider an extra (artificial) linear

type S (thus, that is *not* **non-shared**), to show how it will become hidden (on the right of $\triangleright$) after focus. We also abbreviate the two non-shared capabilities ("**rw** `t` ![]" and "**rw** *l* `Empty#`![]") as A_0 and A_1, and abbreviate the protocol notation so that it does not show the application of location o. With this, we get the following derivation on line 12:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{E} \in \mathsf{E}}}{\Gamma \mid \mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C}) \vdash \mathsf{focus}\ \mathsf{E} : ![] \dashv \mathsf{E}, ((\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright \cdot)}\ (\textsc{t:Focus-Rely})}{\Gamma \mid (\mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C})) \circledast\!\!- S, A_0, A_1 \vdash \mathsf{focus}\ \mathsf{E} : ![] \dashv (\mathsf{E}, (\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright \cdot) \circledast\!\!- S, A_0, A_1}\ (\textsc{t:Frame})}{\Gamma \mid \mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C}), S, A_0, A_1 \vdash \mathsf{focus}\ \mathsf{E} : ![] \dashv \mathsf{E}, ((\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright S), A_0, A_1}\ (\textsc{by}\ \circledast\!\!-)$$

Note that frame may add elements to the typing environment that cannot be instantiated into valid heaps. That is, the conclusion of the frame rule states that an hypothesis with the extended environment typechecks the expression with the same type and resulting effects. Not all such extensions obey store typing, for instance they may enable adding multiple capabilities to one same location that can never be realized in an actual, correct, heap. However, our preservation theorem ensures that starting from a correct (stored typed) heap and typing environment, we cannot reach an incorrect heap state.

3.3.5 Consumer Code

We now go over the implementation of the last function of the pipe example, `tryTake` listed in Figure 3.6. The code should be straightforward up to the use of alternative program states ($\oplus$). At that point in the code, the interaction between our case and $\oplus$ typing rules becomes crucial since it enables the different branches to obey incompatible guarantees on defocus.

An imprecise state means that we may have one of several different alternative resources and, consequently, the expression must consider all of those cases separately. On line 28, to use each individual alternative of the protocol, we check the expression separately on each alternative (marked as **[a]**, **[b]**, and **[c]** in the typing environments), cf. (T:Alternative-Left) in Figure 2.10. Recall that our case gains precision by ignoring branches that are statically known to not be used. On line 29, when the type checker is case analyzing the contents of `first` on alternative **[b]** it obtains type "`Closed#`![]". Therefore, for that alternative, type checking only examines the `Closed` tag and the respective case branch. This feature enables the case to obey different alternative program states simultaneously, although the effects/guarantee that each branch fulfills are incompatible. For instance, the assignment of line 33 must only occur on alternative **[a]** as the assignment restores the linear value that was previously read from `head` to do the case analysis, while the remaining branches either delete that state or modify the value of head.

3.4 Summary of Extensions

We now list the changes to the system shown in chapter 2 to include support for sharing through the use of our rely-guarantee protocols.

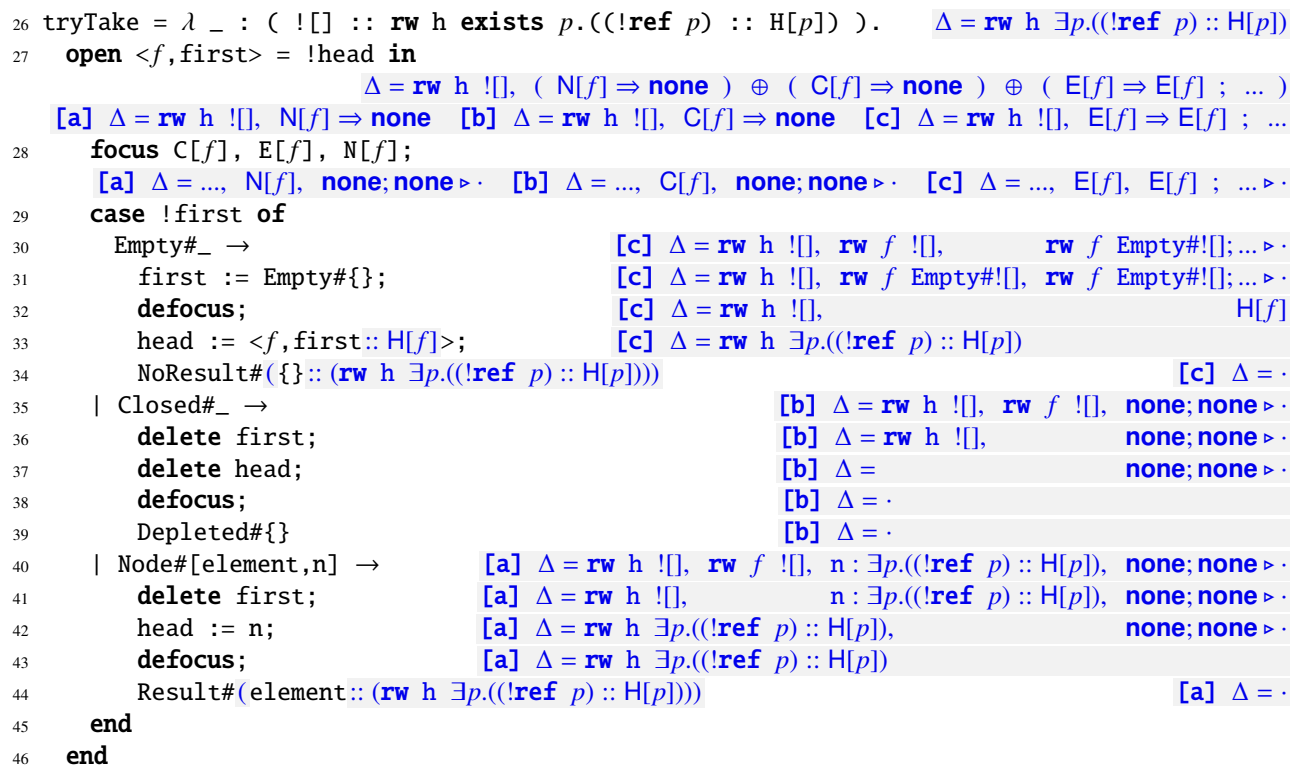

Figure 3.6: Implementation of the `tryTake` function. The abbreviations of `C`, `E`, and `F` are the same as shown above.

The extended grammar, shown in Figure 3.7, includes new constructs to introduce sharing (share), begin a private use of the shared state (focus), and end the private use of the shared state (defocus). The grammar of types was also extended to include our rely and guarantee types, as higlighted in Figure 3.8. Finally, although the format of our typing judgments remains the same, the typing environment is extended to include defocus-guarantees:

Γ	::=	$\cdot$	(empty)
	|	$\Gamma,\ x : A$	(variable binding)
	|	$\Gamma,\ p : \mathbf{loc}$	(location assertion)
	|	$\Gamma,\ X : k$	(kind assertion)
Δ	::=	$\cdot$	(empty)
	|	$\Delta,\ x : A$	(linear binding)
	|	$\Delta,\ A$	(linear resource)
	|	$\Delta_0^G,\ ((A_0; A_1) \triangleright \Delta_1)$	(defocus-guarantee)
k	::=	$\mathbf{type} \mid \mathbf{type} \to k \mid \mathbf{loc} \to k$	

where Δ_0^G syntactically restricts Δ_0 to not include a defocus-guarantee. Suffices to note that this restriction ensures that defocus-guarantees are nested on the right of $\triangleright$ in Δ_1 and that, at each level, there exists only one pending defocus-guarantee. Δ^G is also used to forbid capture of defocus-

$\rho \in$ Location Constants (Addresses) $\quad l \in$ Location Variables $\quad p ::= \rho \mid l$

$\mathtt{t} \in$ Tags $\quad \mathtt{f} \in$ Fields $\quad x \in$ Variables $\quad X \in$ Type Variables

$v \in$ Values	$::=$	x	(variable)
	$\mid$	ρ	(address)
	$\mid$	$\lambda x : A.e$	(function)
	$\mid$	$\langle X \rangle\, e$	(type abstraction)
	$\mid$	$\langle l \rangle\, e$	(location abstraction)
	$\mid$	$\langle A, v \rangle$	(pack type)
	$\mid$	$\langle p, v \rangle$	(pack location)
	$\mid$	$\{\overline{\mathtt{f} = v}\}$	(record)
	$\mid$	$\mathtt{t}\#v$	(tagged value)
$e \in$ Expressions	$::=$	v	(value)
	$\mid$	$v[A]$	(type application)
	$\mid$	$v[p]$	(location application)
	$\mid$	$v.\mathtt{f}$	(field)
	$\mid$	$v\ v$	(application)
	$\mid$	let $x = e$ in e end	(let)
	$\mid$	open $\langle X, x \rangle = v$ in e end	(open type)
	$\mid$	open $\langle l, x \rangle = v$ in e end	(open location)
	$\mid$	case v of $\overline{\mathtt{t}\#x \rightarrow e}$ end	(case)
	$\mid$	new v	(cell creation)
	$\mid$	delete v	(cell deletion)
	$\mid$	$!v$	(dereference)
	$\mid$	$v := v$	(assign)
	$\mid$	share A_0 as $A_1 \parallel A_2$	(share)
	$\mid$	focus $\overline{A}$	(focus)
	$\mid$	defocus	(defocus)

Note: ρ is not source-level.

Figure 3.7: Extensions to the grammar of chapter 2 to include our sharing constructs.

$$u \;::=\; l \mid X \qquad U \;::=\; p \mid A$$

A	$::=$	$!A$	(pure type)
	$\mid$	$A \multimap A$	(linear function)
	$\mid$	$\forall X.A$	(universal type quantification)
	$\mid$	$\exists X.A$	(existential type quantification)
	$\mid$	$[\overline{\mathtt{f} : A}]$	(record)
	$\mid$	$\sum_i \mathtt{t}_i \# A_i$	(tagged sum)
	$\mid$	**ref** p	(reference type)
	$\mid$	(**rec** $X(\overline{u}).A)[\overline{U}]$	(recursive type)
	$\mid$	$X[\overline{U}]$	(type variable)
	$\mid$	$A :: A$	(resource stacking)
	$\mid$	$A * A$	(separation)
	$\mid$	$\forall l.A$	(universal location quantification)
	$\mid$	$\exists l.A$	(existential location quantification)
	$\mid$	$A \oplus A$	(alternative)
	$\mid$	$A \,\&\, A$	(intersection)
	$\mid$	**rw** $p\ A$	(read-write capability to p)
	$\mid$	**none**	(empty resource)
	$\mid$	$A \Rightarrow A$	(rely type)
	$\mid$	$A \;;\; A$	(guarantee type)

Figure 3.8: Extensions to the types grammar of chapter 2 to include our rely and guarantee types.

(T:Function)
$$\frac{\Gamma \mid \Delta^G, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta^G \vdash \lambda x : A_0.e : A_0 \multimap A_1 \dashv \cdot}$$

(T:Forall-Loc)
$$\frac{\Gamma, t : \mathbf{loc} \mid \Delta^G \vdash e : A \dashv \cdot}{\Gamma \mid \Delta^G \vdash \langle t \rangle\, e : \forall t.A \dashv \cdot}$$

(T:Forall-Type)
$$\frac{\Gamma, X : \mathbf{type} \mid \Delta^G \vdash e : A \dashv \cdot}{\Gamma \mid \Delta^G \vdash \langle X \rangle\, e : \forall X.A \dashv \cdot}$$

(T:Focus-Rely)
$$\frac{A_0 \in \overline{A}}{\Gamma \mid (A_0 \Rightarrow A_1) \vdash \mathsf{focus}\ \overline{A} : ![] \dashv A_0,\ (A_1 \triangleright \cdot)}$$

(T:Defocus-Guarantee)
$$\frac{}{\Gamma \mid \Delta_0,\ A_0,\ ((A_0; A_1) \triangleright \Delta_1) \vdash \mathsf{defocus} : ![] \dashv \Delta_0,\ A_1,\ \Delta_1}$$

(T:Frame)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \circledast \Delta_2 \vdash e : A \dashv \Delta_1 \circledast \Delta_2}$$

(T:Share)
$$\frac{A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \mid \Delta, A_0 \vdash \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2 : ![] \dashv \Delta, A_1, A_2}$$

Figure 3.9: Changes and additions to the static semantics of chapter 2.

guarantees by functions and other constructs that can keep part of the linear typing environment for themselves.

The changes to the typing rules are listed in Figure 3.9, with the remaining typing rules and subtyping left unchanged. Crucially, functions can only capture a Δ^G linear environment to ensure that they will not hide a pending defocus-guarantee and similarly on $\forall$ abstractions. This restriction is necessary because our types do not express pending defocus-guarantees and would otherwise enable such pending operation to be erased or hidden—which would impact safety. The remaining four typing rules were discussed above.

3.4.1 Technical Results

As in the previous chapter, our soundness results use progress and preservation theorems. Since the preservation theorem changed slightly, we re-state both theorems.

Theorem 3 (Progress)**.** *If e_0 is a closed expression (and where Γ and Δ_0 are also closed) such that $\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$ then either:*

- *e_0 is a value, or;*
- *if exists H_0 such that $\Gamma \mid \Delta_0 \vdash H_0$ then $H_0 \,;\, e_0 \;\mapsto\; H_1 \,;\, e_1$.*

The progress statement ensures that all well-typed expressions are either values or, if there is a heap that obeys the typing assumptions, the expression can step to some other program state — i.e. a well-typed program never gets stuck, although it may diverge.

Theorem 4 (Preservation)**.** *If e_0 is a closed expression such that:*

$$\Gamma_0 \mid \Delta_0 \vdash e_0 : A \dashv \Delta \qquad \Gamma_0 \mid \Delta_0 \circledast \Delta_2 \vdash H_0 \qquad H_0 \,;\, e_0 \;\mapsto\; H_1 \,;\, e_1$$

then, for some Δ_1 and Γ_1 we have:

$$\Gamma_0, \Gamma_1 \mid \Delta_1 \circledast \Delta_2 \vdash H_1 \qquad \Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta$$

The theorem above requires the expression e_0 to be closed so that it is ready for evaluation ($\mapsto$). The preservation statement ensures that the resulting effects (Δ) and type (A) of the expression remains the same throughout the execution. Therefore, the initial typing is preserved by the dynamics of the language, regardless of possible environment extensions ($\circledast \Delta_2$). This formulation respects the intuition that the heap used to evaluate an expression may include other parts (Δ_2) that are not relevant to check that expression but that are, nonetheless, consistent with the heap structure. Furthermore, by considering Δ_2 separate from Δ_0, the statement simplifies the proof for framing cases.

To support these theorems, we extend the store typing definition of the previous chapter with the rules of Figure 3.10. With (STR:SHARE), we are able to introduce protocols by splitting an existing resource (A_0). (STR:DEFOCUS-GUARANTEE) ensures that the promised state (A_0) of a pending guarantee ($A_0; A_1$) can be used to compose the next step of the protocol (A_1) with some other protocol (A_2) that must necessarily be nested in that defocus-guarantee since the shared state it refers is inconsistent. Due to the support of H, it ensures that A_2 is either **none** or a type that is a protocol for the state of A_0. All other parts of the heap must be supported by Δ'. The use of $\setminus$ is

$$\frac{\Gamma \mid \Delta, A_0 \vdash H \qquad A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \mid \Delta, A_1, A_2 \vdash H} \text{(STR:SHARE)}$$

$$\frac{\Gamma \mid \Delta' \vdash H' \qquad \Gamma \mid \Delta'' \vdash H \qquad \Delta \setminus A_2 = \Delta'' \qquad A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \mid \Delta', ((A_0; A_1) \triangleright \Delta) \vdash H', H} \text{(STR:DEFOCUS-GUARANTEE)}$$

Figure 3.10: Extension to the store typing rules of chapter 2.

to highlight that A_2 (a protocol that is the result of merging all other protocols to the inconsistent shared state state that may be in Δ) may be at any defocus depth, however it *must* be nested on the right of $\triangleright$ since that shared state is (potentially) inconsistent.

With $\Delta \setminus A$ we are able to extract A from Δ such that A is the result of "merging" all other possible protocols that are compatible in Δ, up until there are no more. We define $\Delta \setminus A$ as follows:

$$\frac{}{(\Delta, A) \setminus A = \Delta} \qquad \frac{A_0 \Rrightarrow A_1 \parallel A_2 \qquad \Delta_2 \setminus A_2 = \Delta_1 \qquad \Delta_1 \setminus A_1 = \Delta_0}{\Delta_2 \setminus A_0 = \Delta_0} \qquad \frac{A_0 \Rrightarrow A_1 \parallel A_2 \qquad \Delta_1 \setminus A_1 = \Delta_0 \qquad \Delta_2 \setminus A_2 = \Delta_3}{(\Delta_1, A' \triangleright \Delta_2) \setminus A_0 = \Delta_0, A' \triangleright \Delta_3}$$

Unsafe interference occurs when an alias assumes a type that is incompatible with the real value stored in the shared state, potentially causing the program to become stuck. However, we proved that any well-typed program in our language cannot become stuck. Thus, although our protocols enable a diverse set of uses of shared state, these theorems show that when rely-guarantee protocols are respected those uses are safe.

Informally, correctness of protocol composition is based on the properties: 1) a split results in protocols that can always take a step with the current state of the shared resources, thus are never stuck; and, 2) protocol composition is a partial commutative monoid (associative, commutative, and with **none** as the identity element). Because of property 2), iterative splittings of existing protocols remain struck-free, unable to cause unsafe interference. We now state these properties formally. The following two lemmas show stuck freedom by properties that resemble progress and preservation but over protocols:

Lemma 2 (Composition Progress)**.** *If* $S \Rrightarrow P \parallel Q$ *then* $\langle\, S \Rrightarrow P \parallel Q \,\rangle \mapsto C$.

Meaning that if two protocols, P and Q, safely compose then their configuration can take a step to another set of configurations, C.

Lemma 3 (Composition Preservation)**.** *If* $\langle\, S \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, S' \Rrightarrow P' \parallel Q' \,\rangle \cdot C$ *and* $S \Rrightarrow P \parallel Q$ *then* $S' \Rrightarrow P' \parallel Q'$.

The lemma ensures that if two protocols compose safely, then any of the next configurations that result from stepping will still compose safely.

Perhaps surprisingly, protocol composition does not enforce that the shared resources are not lost. Instead our concern is on safe interference. Indeed, resources that are never used will never be able to unsafely interfere. To avoid losing resources, we must forbid the use of (C-RS:NONE)

on non-terminated protocols and that both P and Q cannot have both simultaneously terminated if there are non-**none** resources left. Once that restriction is considered, our splitting induces a monoid in the sense that for any P and Q for which $S \Rrightarrow P \parallel Q$ is defined there is a single such S (defined up to subtyping and equivalent protocol/state interference specification). Since for any two protocols there may not always exist an S that can be split into P and Q, this is a partial monoid.

Protocol composition obeys the following properties:

Lemma 4. $S \Rrightarrow S \parallel \textbf{\textit{none}}$.

Lemma 5. *If* $S \Rrightarrow P_0 \parallel P_1$ *then* $S \Rrightarrow P_1 \parallel P_0$.

Lemma 6. *If we have* $S \Rrightarrow P_0 \parallel P$ *and* $P \Rrightarrow P_1 \parallel P_2$ *then exists* W *such that* $S \Rrightarrow W \parallel P_2$ *and* $W \Rrightarrow P_0 \parallel P_1$. *(i.e. if* $S \Rrightarrow P_0 \parallel (P_1 \parallel P_2)$ *then* $S \Rrightarrow (P_0 \parallel P_1) \parallel P_2$ *)*

3.5 Additional Examples

We now exemplify some other sharing idioms that can be modeled by our rely-guarantee protocols.

3.5.1 Sharing a Stack

Our protocols are capable of modeling monotonic [40, 74] uses of shared state, although our type's expressiveness is limited to describe a finite number of states. To illustrate monotonic usages, we use the linear stack from Section 2.4.1 where the stack object has two possible typestates: Empty and Non-Empty. The object, with an initial E(mpty) typestate, is accessible through closures returned by the following "constructor" `newStack` function (omitting !'s for conciseness):

$$\begin{aligned}\forall \mathtt{T}.\, [] \multimap \exists \mathtt{E}.\exists \mathtt{NE}.(\ [\ \ & \mathtt{push} : \mathtt{T} :: \mathtt{E} \oplus \mathtt{NE} \multimap [] :: \mathtt{NE},\\ & \mathtt{pop} : [] :: \mathtt{NE} \multimap \mathtt{T} :: \mathtt{E} \oplus \mathtt{NE},\\ & \mathtt{isEmpty} : [] :: \mathtt{E} \oplus \mathtt{NE} \multimap \mathtt{Empty\#}([] :: \mathtt{E}) + \mathtt{NonEmpty\#}([] :: \mathtt{NE}),\\ & \mathtt{del} : [] :: \mathtt{E} \multimap [] \ ]\\ & :: \mathtt{E}\ )\end{aligned}$$

Although the stack is a linear resource, we can use protocols to share the stack. This enables multiple aliases to that same object to coexist and use it simultaneously from non-local contexts. The following protocol converges the stack to a non-empty typestate, starting from an imprecise alternative that also includes the empty typestate.

$$\mathtt{S} \triangleq (\ \mathtt{NE} \oplus \mathtt{E}\ ) \Rightarrow \mathtt{NE}\ ;\ \textbf{rec}\ X.(\ \mathtt{NE} \Rightarrow \mathtt{NE}\ ;\ X\ )$$

Monotonicity means that the type becomes successively more precise, although each alias does not know when those changes occurred. Our protocols can capture such meaning by transitioning from an imprecise alternative type (NE ⊕ E) to a single, more precise, type (NE). Note that, due to focus, the object can undergo intermediate states that are not compatible with the required NE guarantee. However, on defocus, clients must provide NE such as by pushing some element to the stack.

For example, a possible client code is as follows:

```
let stack = newStack[int]({}) in
  open <E,<NE,x>> = stack in
    share E as S || S;
    unknown( <E,<NE,x:: S>> ); //passes an alias of the stack to some function
    focus E ⊕ NE;
    case x.isEmpty( {} ) of
      Empty#_ →
        x.push( 123 );
        defocus
    | NonEmpty#_ →
        x.pop( {} );
        x.push( 123 );
        defocus
    end;
    // from now on can rely on stack being NonEmpty
    focus NE;
    // ... use x in some way ...
    defocus
    // ... still NonEmpty
  end
end
```

The protocol itself can be continuously re-split into equal protocols, since each copy will produce the same effects as the original protocol. In fact, the new copies not only cannot introduce unexpected (unsafe) interference nor is their existence observable to other aliases of the stack.

Protocol composition is applied as follows:

$$\mathtt{E} \Rrightarrow \mathtt{S} \parallel \mathtt{S}$$

Resulting in the following set of configurations:

$$\begin{array}{l}
\{\ ❶ \langle\ \mathtt{E} \Rrightarrow (\mathtt{NE} \oplus \mathtt{E}) \Rightarrow \mathtt{NE}; \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \parallel (\mathtt{NE} \oplus \mathtt{E}) \Rightarrow \mathtt{NE}; \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \rangle, \\
\quad ❷ \langle\ \mathtt{NE} \Rrightarrow \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \parallel (\mathtt{NE} \oplus \mathtt{E}) \Rightarrow \mathtt{NE}; \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \rangle, \\
\quad ❸ \langle\ \mathtt{NE} \Rrightarrow (\mathtt{NE} \oplus \mathtt{E}) \Rightarrow \mathtt{NE}; \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \parallel \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \rangle, \\
\quad ❹ \langle\ \mathtt{NE} \Rrightarrow \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \parallel \mathbf{rec}\ X.(\mathtt{NE} \Rightarrow \mathtt{NE}; X) \rangle\ \ \}
\end{array}$$

By using (c-ss:Step) with the subtyping extension since we have that "$\mathtt{E} <: (\mathtt{NE} \oplus \mathtt{E})$".

Similarly, we have that "$\mathtt{S} \Rrightarrow \mathtt{S} \parallel \mathtt{S}$", which is straightforward to see by (c-ps:Step) that simulates a step of the old protocol in the newly split protocols (again using the subtyping extension). Namely, we use the same subtyping shown above on the rely type of the protocols.

$$\frac{\mathtt{NE} <: (\mathtt{NE} \oplus \mathtt{E}) \qquad \mathtt{NE} <: \mathtt{NE}}{\langle\ (\mathtt{NE} \Rightarrow \mathtt{NE}\ ;\ \ldots)\ \Rrightarrow \mathcal{R}[\ (\mathtt{NE} \oplus \mathtt{E}) \Rightarrow \mathtt{NE}\ ;\ \ldots\ ]\ \rangle \mapsto \ldots}\ (\text{c-ps:Step})$$

3.5.2 Capturing Local Knowledge

Although our protocol types, as introduced so far, cannot express the same amount of detail on local knowledge as prior work [13, 56], they are however expressive enough to capture the underlying

principle that enables us to keep increased precision on the shared state between steps of a protocol. However, this local knowledge must be captured in a typestate, and thus fixed "a priori".

For this example, we use a simple two-states counter consisting of typestate N (encoding a set of natural numbers and zero) and P (encoding some positive natural number, that therefore does not include zero). The two types are defined as follows:

$$\mathsf{N} \triangleq \mathtt{Z\#}![] + \mathtt{NZ\#int} \qquad \mathsf{P} \triangleq \mathtt{NZ\#int} \qquad \text{(note that: "P <: N" by (ST:SUM))}$$

We now share an N cell in two asymmetric roles: `IncOnly`, that limits the actions of the alias to only increment the counter (in a protocol that can be split/shared repeatedly); and `Any`, an alias that relies on the restriction imposed by the previous protocol to be able to capture a stronger rely property in a step of its own protocol. Assuming an initial "**rw** p N" capability, this cell can be shared using the following two protocols:

$$\begin{aligned} \mathtt{IncOnly}[p] &\triangleq (\mathbf{rw}\ p\ \mathsf{N}) \Rightarrow (\mathbf{rw}\ p\ \mathsf{P})\ ;\ \mathtt{IncOnly}[p] \\ \mathtt{Any}[p] &\triangleq (\mathbf{rw}\ p\ \mathsf{N}) \Rightarrow (\mathbf{rw}\ p\ \mathsf{P})\ ;\ (\mathbf{rw}\ p\ \mathsf{P}) \Rightarrow (\mathbf{rw}\ p\ \mathsf{N})\ ;\ \mathtt{Any}[p] \end{aligned}$$

Thus, by constraining the actions of `IncOnly` we can rely on the assumption that `Any` remains positive on its second step, regardless of when other aliases may manipulate the shared state. Therefore, on the second step of `Any`, the case analysis can be sure that the value of the shared state must have remained with the `NZ` tag between focuses.

Client code can use the types above as follows:

```
1  open <v,value> = new Z#{} in
2    share (rw v Z#![]) as IncOnly[v] || Any[v];
3    unknown( <v, value:: IncOnly[v]> );                                  Δ = Any[v],  ...
4    focus N[v];                              Δ = rw v N,  ((rw v P ; (rw v P ⇒ rw v N ; Any[v])) ▷ ...)
5    case !value of // may or may not be "positive"
6      Z#_ → value := NZ#123
7    | NZ#n → value := NZ#456
8    end;                                     Δ = rw v P,  ((rw v P ; (rw v P ⇒ rw v N ; Any[v])) ▷ ...)
9    defocus;                                              Δ = (rw v P ⇒ rw v N ; Any[v]),  ...
10   ... // anything else may be executed many or none times
11   focus P[v];                                           Δ = rw v P,  ((rw v N ; Any[v]) ▷ ...)
12   case !value of // protocol enables type system to assume state remains nonzero!
13     NZ#n → value := Z#{}
14   end;                                                  Δ = rw v N,  ((rw v N ; Any[v]) ▷ ...)
15   defocus
```

We have the initial split:

$$\mathbf{rw}\ v\ \mathsf{N} \ \Rrightarrow\ \mathtt{IncOnly}[v] \parallel \mathtt{Any}[v]$$

that results in the following set of configurations:

$$\begin{aligned} \{\ &❶\langle\ \mathbf{rw}\ v\ \mathsf{N} \ \Rrightarrow\ \mathbf{rw}\ v\ \mathsf{N} \Rightarrow \mathbf{rw}\ v\ \mathsf{P}\ ;\ \mathtt{IncOnly}[v] \parallel \mathbf{rw}\ v\ \mathsf{N} \Rightarrow \mathbf{rw}\ v\ \mathsf{P}\ ;\ \mathbf{rw}\ v\ \mathsf{P} \Rightarrow \mathbf{rw}\ v\ \mathsf{N}\ ;\ \mathtt{Any}[v]\ \rangle, \\ &❷\langle\ \mathbf{rw}\ v\ \mathsf{P} \ \Rrightarrow\ \mathbf{rw}\ v\ \mathsf{N} \Rightarrow \mathbf{rw}\ v\ \mathsf{P}\ ;\ \mathtt{IncOnly}[v] \parallel \mathbf{rw}\ v\ \mathsf{N} \Rightarrow \mathbf{rw}\ v\ \mathsf{P}\ ;\ \mathbf{rw}\ v\ \mathsf{P} \Rightarrow \mathbf{rw}\ v\ \mathsf{N}\ ;\ \mathtt{Any}[v]\ \rangle, \\ &❸\langle\ \mathbf{rw}\ v\ \mathsf{P} \ \Rrightarrow\ \mathbf{rw}\ v\ \mathsf{N} \Rightarrow \mathbf{rw}\ v\ \mathsf{P}\ ;\ \mathtt{IncOnly}[v] \parallel \qquad\qquad \mathbf{rw}\ v\ \mathsf{P} \Rightarrow \mathbf{rw}\ v\ \mathsf{N}\ ;\ \mathtt{Any}[v]\ \rangle\ \} \end{aligned}$$

Naturally, the split "$\texttt{IncOnly}[v] \Rrightarrow \texttt{IncOnly}[v] \parallel \texttt{IncOnly}[v]$" also safely composes.

Note that we could also have started from the protocol:

$$\textbf{rec}\ X.(\ (\textbf{rw}\ v\ \mathsf{N})\ \Rightarrow\ (\textbf{rw}\ v\ \mathsf{P})\ ;\ \textbf{rec}\ Y.(\ (\ (\textbf{rw}\ v\ \mathsf{P})\ \Rightarrow\ \mathsf{N}\ ;\ X\ )\ \&\ (\ (\textbf{rw}\ v\ \mathsf{N})\ \Rightarrow\ (\textbf{rw}\ v\ \mathsf{P})\ ;\ Y\ )\ )\ )$$

And split this protocol into the `Any` and `IncOnly` protocols.

3.5.3 Iteratively Sharing State

Our technique is able to match an arbitrary number of aliases by splitting an existing protocol into new protocols. In previous examples, we saw how we can copy an existing protocol into identical copies. However, we can also extend the original protocol's uses of the shared state by *appending* additional steps, provided that those new uses will not cause unsafe interfere with the previously held assumptions. One way to obey this condition is to simply specialize existing steps (in this chapter, limited to subtyping specialization) or append new steps to an otherwise ownership recovery step, as we will be showing next.

This example shows step extensions by encoding a form of delegation through shared state that models a kind of "server-like process". Although single-threaded, such a system could be implemented using co-routines or collaborative multi-tasking. The overall computation is split between three individual workers (for instance by each using a private list containing cells with pending, shared, jobs) each with a specific task. A Receiver uses a `Free` job cell and stores some `Raw` element in it. A Compressor processes a `Raw` element into a `Done` state. Finally, the Storer removes the cells in order to store them elsewhere. In real implementations, each worker would be used by separate handlers/threads, triggered in unpredictable orders, to handle these jobs.

The example also shows how we can share multiple locations together, bundled using $*$. Each job is kept in a *c*ontainer cell while the *f*lag, used to communicate the information about the kind of content stored in the container, is kept in a separate cell. To simplify the presentation the raw value is typed with `V` and the processed value has type `P`, without giving concrete representations to these type abbreviations.

The types and protocols are:

$$\begin{array}{rcl}
\mathsf{F} & \triangleq & (\textbf{rw}\ f\ \texttt{Free\#}![])\ *\ (\textbf{rw}\ c\ ![]) \\
\mathsf{R} & \triangleq & (\textbf{rw}\ f\ \ \texttt{Raw\#}![])\ *\ (\textbf{rw}\ c\ !\texttt{V}) \\
\mathsf{D} & \triangleq & (\textbf{rw}\ f\ \texttt{Done\#}![])\ *\ (\textbf{rw}\ c\ !\texttt{P})
\end{array}$$

$$\begin{array}{rcl}
\mathsf{Receiver} & \triangleq & \mathsf{F} \Rightarrow \mathsf{R} \\
\mathsf{Compressor} & \triangleq & \textbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ (\mathsf{R} \Rightarrow \mathsf{D})\ ) \\
\mathsf{Storer} & \triangleq & \textbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ \textbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}; Y)\ \oplus\ (\mathsf{D} \Rightarrow \textbf{none})\ )\ )
\end{array}$$

The protocol for the Receiver is straightforward since it just processes a free cell by assigning it a raw value. Similarly, Compressor and Storer follow analogous ideas by using a kind of "waiting" steps until the cell is placed with the desired type for the actions that they are to take.

Note, however, that Storer keeps a more precise state when the cell is not F, even though it is not allowed to publicly modify the state of the cell in that situation.

To obtain these protocols through binary splits, we need an *intermediate* protocol that will be split to create the Compressor and Storer protocols, defined as follows:

$$\mathsf{Intermediate} \triangleq \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ (\mathsf{R} \Rightarrow \mathbf{none})\ )$$

The initial split (starting with F) is as follows:

$$\mathsf{F}\ \Rrightarrow\ \mathsf{Receiver}\ \|\ \mathsf{Intermediate}$$

Producing the following set of configurations:

$$\begin{array}{l}\{\ \langle\ \ \mathsf{F} \Rrightarrow \mathsf{F} \Rightarrow \mathsf{R}\ \|\ \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ (\mathsf{R} \Rightarrow \mathbf{none})\ )\ \rangle, \\ \ \ \langle\ \ \mathsf{R} \Rrightarrow \mathbf{none}\ \|\ \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ (\mathsf{R} \Rightarrow \mathbf{none})\ )\ \rangle, \\ \ \ \langle\ \mathbf{none} \Rrightarrow \mathbf{none}\ \|\ \mathbf{none}\ \rangle\ \}\end{array}$$

The Intermediate protocol is then further split to match the uses done through the Compressor and Storer protocols. However, before we can do that split, we must *extend* Intermediate to include additional steps that will match the work to be done by the two new protocols, thus creating the Intermediate2 protocol.

$$\mathsf{Intermediate2} \quad = \quad \mathsf{Intermediate} \bowtie \mathbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}; Y)\ \&\ (\mathsf{R} \Rightarrow \mathsf{D}; \mathsf{D} \Rightarrow \mathbf{none})\ )$$

Therefore such extension ($\bowtie$) will append the new uses to Intermediate's ownership recovery step, "R $\Rightarrow$ **none** ; **none**". This results in the protocol:

$$\mathsf{Intermediate2} \quad = \quad \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}\ ;\ X)\ \oplus\ \mathbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}\ ;\ Y)\ \&\ (\mathsf{R} \Rightarrow \mathsf{D}\ ;\ \mathsf{D} \Rightarrow \mathbf{none})\ )\ )$$

Now we can split Intermediate2 in the two Compressor and Storer protocols:

$$\mathsf{Intermediate2}\ \Rrightarrow\ \mathsf{Compressor}\ \|\ \mathsf{Storer}$$

We now show the resulting configurations:

$$\begin{array}{l}\{\ \textbf{❶}\ \langle\ \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}\ ;\ X)\ \oplus\ \mathbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}\ ;\ Y)\ \&\ (\mathsf{R} \Rightarrow \mathsf{D}\ ;\ \mathsf{D} \Rightarrow \mathbf{none})\ )\ ) \\ \qquad \Rrightarrow\ \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ (\mathsf{R} \Rightarrow \mathsf{D})\ )\ \|\ \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ \mathbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}; Y)\ \oplus\ (\mathsf{D} \Rightarrow \mathbf{none})\ )\ )\ \rangle, \\ \ \ \textbf{❷}\ \langle\ \mathbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}\ ;\ Y)\ \&\ (\mathsf{R} \Rightarrow \mathsf{D}\ ;\ \mathsf{D} \Rightarrow \mathbf{none})\ ) \\ \qquad \Rrightarrow\ \mathbf{rec}\ X.(\ (\mathsf{F} \Rightarrow \mathsf{F}; X)\ \oplus\ (\mathsf{R} \Rightarrow \mathsf{D})\ )\ \|\ \mathbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}; Y)\ \oplus\ (\mathsf{D} \Rightarrow \mathbf{none})\ )\ \rangle, \\ \ \ \textbf{❸}\ \langle\ \mathsf{D} \Rightarrow \mathbf{none}\ \Rrightarrow\ \mathbf{none}\ \|\ \mathbf{rec}\ Y.(\ (\mathsf{R} \Rightarrow \mathsf{R}; Y)\ \oplus\ (\mathsf{D} \Rightarrow \mathbf{none})\ )\ \rangle, \\ \ \ \textbf{❹}\ \langle\ \ \ \mathbf{none}\ \Rrightarrow\ \mathbf{none}\ \|\ \mathbf{none}\ \rangle\ \}\end{array}$$

The Receiver alias never needs to see how the other two aliases use the shared state. Although the second split is independent from the initial one, protocol composition ensures that it cannot cause unsafe interference. Thus, both Compressor and Storer protocols must match the interference that would be produced by Intermediate alone, although they can extend Intermediate's uses of the shared state by appending new uses to the terminating step.

3.6 Related Work

We now discuss related work, mainly focusing on the handling of interference.

Work on permissions [16, 17, 18, 19, 67] restricts the interference an alias may produce by constraining the actions that the alias can perform over that shared state, which is modeled by the kind of permission that the alias is assigned to. Thus, tracking the safe split and merging of permissions becomes crucial, as permissions need to match the different set of aliases to some shared resource. With *fractional permissions* [18], the full permission to use state is divided into fractions. To avoid interference, write accesses require the full permission (1, i.e. not aliased or unique) but many read-only accesses can co-exists by sharing fractions ($1/k$) of the full permission. This kind of tracking is both reversible and symmetric, as each fraction can be further divided on its own. However, tracking fractions can become cumbersome leading to the proposal of *counting permissions* [16] to instead provide asymmetric (but still reversible) tracking. In this situation, a central "authority" counts how many outstanding permissions exist and only restores write access when all copies have been accounted for.

We could model similar accounting mechanisms within our system, for instance through the following subtyping rules:

$$(\mathbf{rw}^{1}\ p\ A) <:> ((\mathbf{ro}^{\frac{1}{2}}\ p\ A)\ *\ (\mathbf{ro}^{\frac{1}{2}}\ p\ A)) \qquad (\mathbf{ro}^{\frac{1}{k}}\ p\ A) <:> ((\mathbf{ro}^{\frac{1}{2k}}\ p\ A)\ *\ (\mathbf{ro}^{\frac{1}{2k}}\ p\ A))$$

for fractional permissions (symmetric splitting), and:

$$(\mathbf{rw}\ p\ A) <:> ((\mathbf{ro}^{1}\ p\ A)\ *\ (\mathbf{ro}\ p\ A)) \qquad (\mathbf{ro}^{k}\ p\ A) <:> ((\mathbf{ro}^{k+1}\ p\ A)\ *\ (\mathbf{ro}\ p\ A))$$

for counting permissions (asymmetric splitting).

Since their addition is relatively straightforward, we choose to not add these mechanisms to our core language. For our intended sharing cases, neither technique would allow ownership recovery as both require knowing that the two locations to be merged are equal (the "p" in the rules above). Our sharing mechanisms aims to enable ownership recovery even when the type system loses track of which variables alias which state, essentially by reasoning about the causality of state changes within the shared cells. Both merging of fractional/counting permissions (or even of protocols, if considered) must rely on some information that the permissions are related to the same location. If that information is available, then it is possible to design a way to merge protocols just as fractions/counting permissions can be merged. Our sharing mechanism follows similar notions of splitting resources as introduced by these permission works. However, the richness in variety of the different protocols that can be safely composed means that protocol splitting can support both symmetric (such as invariant-based sharing) and asymmetric splitting (such as through iterative splitting that appends new uses), but incurs in consequence more verbosity than the fractions/counting permission mechanisms above.

Fähndrich and DeLine introduced the notion of adoption and focus [33, 39] to enable a "fictional" linearity over shared resources. Temporary linear uses must be invisible to other aliases, and they are required to restore the initially held invariant upon leaving the focused block. $\mathbf{L}^{3}$ [5] also includes a form of invariant-based sharing, similar to the adoption and focus mechanism, but

here called freeze/thaw/refreeze. Frozen capabilities can be freely duplicated (and even forgotten) where all copies obey a single type invariant (the frozen type). Thawing yields the regular capability contained in the frozen type. Any use of thaw-refreeze (similar to focus-defocus) is only valid if it guarantees that no other copy of the frozen location will see the intermediary, private changes that may disobey the (publicly) frozen type.

Through a combination of fractional permissions and by categorizing read-write uses into different permission kinds, *access permissions* [10, 11, 13, 15, 97] provide a fixed set of ways to control alias interference including support for private states. For instance, access permissions include the `unique` permission (linear type), `immutable` (where aliasing is free to occur without risking interference), and `share` (essentially, a form of invariant-based sharing). (Other works [24, 90] also employ simple invariant-based to handle interference over shared state.)

Our work uses a similar notion to focus (via the focus/defocus constructs) to delimit where inspection of shared resources can occur, as we enable shared resources to be temporarily used as a (non-shared) linear type. However, our protocols are free from the requirement of invariant-based sharing, as the type at defocus needs not to be the same as the focus type. From the permission perspective, our design omits the read-write distinction (we only support "read and write" capabilities) to focus exclusively on structuring alias interference using more fundamental protocol primitives.

Work on *Gradual Typestate* [97] approaches the problem of dynamic ownership recovery within the domain of a rich (access) permission system. Their system adds some run-time overhead (that tracks and counts aliases) to enable recovering uniqueness whenever an object is not actually aliased. However, this mechanism can fail at run-time if ownership recovery is attempted over an object that is still aliased.

In *Mezzo* [76], a dynamic mechanism related to adoption and focus is proposed as a way to circumvent a more restrictive aliasing policy of their core language. Dubbed *adoption and abandon*, it allows a permission to mutable state to be "adopted" (stored in a forest of references) and later extracted from it ("abandoned"). With this technique, groups of permissions are handled together by some name, enabling them to be shared across different program contexts. The type of those stored permissions are still the same (i.e. invariant), but it does allow ownership recovery through the dynamic abandon mechanism. As with *Gradual Typestate*, ownership recovery can fail at run-time and requires additional overhead to check its validity during execution.

Our protocols enable us to statically rule out run-time ownership recovery errors. Both approaches above resemble a kind of dynamic use of fractions/counting permissions to simplify tracking of aliases in the surface language at the expense of weaker safety guarantees. While it is unlikely that our protocols have sufficient expressiveness to capture all (safe) uses that can be done through those dynamic approaches, our static guarantees of safety eliminate the run-time overhead. The design trade-off appears to be a choice between requiring additional syntactical verbosity (in our case, a case analysis and tags) and a potentially less flexible language, or push some of that complexity to the operational semantics at the expense of a potentially less safe outcome (as is done in the two dynamic approaches above).

Gordon *et al.* propose *Rely-Guarantee References* [49], a type system where individual references carry three additional type components: a predicate (for local knowledge), a guarantee relation, and a rely relation. They handle an unknown number of aliases by constraining the writes to a cell to fit within the alias' declared guarantee, similarly to how rely-guarantee is used in program logics to handle thread-based interference but here used to handle alias interference within the sequential setting. Although they support a limited form of protocol (and their technique can generally be considered as a two-state protocol), their system effectively limits the actions allowed by each new alias to be strictly decreasing since their guarantee must fit within the original alias' guarantee. Since we support ownership recovery of shared state, a cell can be shared and return to non-shared without this restriction. Unlike ours, their work does not allow intermediate inconsistent states since all updates are publicly visible. In addition, their work requires proof obligations for, among other things, guarantee satisfaction while we use a more straightforward definition of protocol composition that is not dependent on theorem-proving. While their use of dependent refinement types adds expressiveness to the specification of interference, it also increases the challenges in automation as typechecking can require manual assistance in Coq. Our protocol design also favors more local reasoning, since each protocol characterizes an alias's own individual perspective on the shared state as opposed to carrying the two sets of actions (local and non-local) throughout the program.

In *Chalice* [58], programmer-supplied permissions and predicates are used to show that a program is free of data races and deadlocks. A limited form of rely-guarantee is used to reason about changes to the shared state that may occur between atomic sections. All changes from other threads must be expressed in auxiliary variables and be constrained to a two-state invariant that relates the *current* with the *previous* state, and where all rely and guarantee conditions are the same for all threads. However, their work approaches topics that we do not, such as deadlock prevention.

Monotonic [40, 74] based sharing enables unrestricted aliasing that cannot interfere since the changes converge to narrower, more precise, states. Our protocols are able to express monotonicity, for instance as shown in examples using monotonic counters. Note, however, that our formalization lacks the type expressiveness of [74] since we do not include support for type refinements. We believe this concern is orthogonal to our core sharing concepts, and is left as future work. Crucially, our mechanism is capable of expressing more than just monotonicity. For instance, due to ownership recovery, a cell can oscillate between shared and non-shared states during its lifetime, and allow each sharing phase to be completely unrelated to previous uses of that cell.

Our reasoning about interference was also influenced by Jones' initial proposal to handle interference via rely and guarantee conditions [55]. Although that line of research has followed a more strict approach to full functional verification, using program logics [51], there is some consilience with approaches that employ type-based verification. For instance, even linear logic [45] and more recent separation logic [78] share some underlying logical notions or resource handling. However, our own approach targets a middle-ground: less precise than full functional verification but more expressive than traditional type systems, with the goal of offering "lightweight" (in the sense of

fully automatic) and reasonably responsive verification.

Approaches that use advanced program logics [35, 37, 41, 71, 87, 92] employ rely-guarantee reasoning to verify inter-thread interference. Although our approach is type-based rather than program logic-based, there are several underlying similarities. *Concurrent abstract predicates* [35] extend the concept of *abstract predicates* [71] to express how state is manipulated, supporting internally aliased state through a *fiction of disjointness* (also present in [54, 56]) that is based on rely-guarantee principles and has similarities to our own abstractions. Their use of rely-guarantee also allows intermediate states within a critical section, which are immediately weakened (made stable) to account for possible interference when that critical section is left. While our use of rely-guarantee is tied to state (i.e. is seen as a resource), not threads, our protocols capture an identical notion of stability through a simpler constraint that ensures all visible states are considered during protocol composition. Likewise, our protocol format is designed to model interference from the isolated ("first-class") perspective of each alias, instead of having to carry around what other threads can or cannot [37] do. Because our protocols are just (linear) resources, we can use the standard frame-rule to hide other resources (instead of requiring extensions to support framing over rely-guarantee conditions as in [41]). In our work, we use protocol composition to check if the different interference perspectives are compatible and can compose safely together. Their designs employ this composition check (stability) at each point of interference, requiring precise information about what other interference may occur to be carried around the program. We offer a higher degree of isolation/locality, at the cost of requiring a kind of "pre-computed" stability check, protocol composition. Our composition mechanism also enables interference to change over time instead of requiring auxiliary variables to enable/disable which interference (in a fixed set of rely and guarantee conditions) may occur at that point in the program.

Per module notions of resource enable sharing by leveraging the concept of fictional separation [35, 54] into *partial commutative monoids* [35, 36, 37, 56] that split and merge access to modules, while sharing access to resources under that module layer. This generic sharing mechanism was shown to be capable of modeling rely-guarantee reasoning in [36]. Since thread-based interference is rooted in alias-related interference, these mechanisms are generally agnostic to whether sharing/interference occurs in a single-threaded or multi-threaded environment. Thus, they focus on how seemingly unrelated components interact, within a module, and under that layer of (fictional) separation.

We compare more closely to [56] due to our common use of $\mathbf{L}^3$ [5]. Krishnaswami *et al.* [56] define a generic sharing rule based on the use of frame-preserving operations over programmer-supplied commutative monoids, for safe sharing within a single-threaded environment. The core principle is centered on splitting the internal resources of a module such that all aliases obey an invariant that is shared, while also keeping some knowledge about the locally-owned shared state. By applying a frame condition over its specification, their shared resources ensure that any interference between clients is benign since it preserves the fiction of disjointness. Thus, local assumptions can interact with the shared state without being affected by the actions done through other aliases of that shared state.

Their work does not approach the issue of decidability of resource splitting, and requires wrap-

ping access to shared state in an module abstraction that serves as an intermediary to use shared state. Our work is essentially a custom commutative monoid that enables first-class sharing without (necessarily) needing a wrapping module abstraction. Although our protocol composition is a specialized monoid, we showed that this mechanism is relatively flexible, decidable and give an algorithmic implementation (see chapter 5). Other technical differences between our works abound such as their use of affine refinement types, our use of multi-threaded semantics and allowing inconsistent states (i.e. focused/locked cells) to be moved around as first-class, our design choice of keeping capabilities (and protocols) as typing artifacts, etc. Still, the richness of their specification language means that (although it might not always be an obvious, simple, or direct) encoding our use of protocol is likely encodable through the use of auxiliary variables. However, we believe that our use of a protocol paradigm presents a significant conceptual distinction since the protocols themselves can be used to guide the programmer on thinking about each alias's individual role on the interference to that shared state. Similarly, although both models allow ownership recovery, our protocols are typing artifacts which means that we do not need a module layer to enable this recovery and the state of that protocol can be switched to participate in completely unrelated protocols, later on. Additionally, after the initial split, our shared state may continue to be re-split in new ways. Lastly, in the sequential setting of this chapter, we use focus to statically forbid re-entrant uses of shared state while they use dynamic checks that diverge the execution when such operation is wrongly attempted.

Chapter 4

Polymorphic Protocol Composition and the Concurrent Setting

Chapter 3 introduced the core mechanisms for sharing, developed within the sequential setting. However, that system requires interference to be precisely described and fixed beforehand. The system introduced in this chapter aims to improve on that limitation by enabling protocols to be more generic and specify a wide range of allowed interference, thus enabling *polymorphic* protocol specifications. Since a protocol can now abstractly describe the changes that it does to the shared state, it can then work uniformly regardless of how those changes are later implemented by clients. To check this form of composition, we leverage existential and universal bounded quantification to ensure safe interference of those uses. Enabling protocols to range over a wider range of interference will enable a greater degree of separation between what a protocol relies on, and what it guarantees. This will improve the modularity, and locality, of our approach but will naturally require extensions to protocol composition so that composition accounts for such generic usages of the shared state. Furthermore, we develop this approach in the concurrent setting, with a language that supports the fork/join model of concurrency.

Note that the extensions to our sharing mechanism are not intimately tied to the concurrent setting. Instead, both concurrency and polymorphic protocols are generally separate issues up to ensuring mutual exclusion of the modifications to shared state. This means that while in a sequential setting run-time mutual exclusion is guaranteed by a single thread of execution, in the concurrent setting this mutual exclusion must be enforced via a dynamic mechanism (in our design, using locks).

Finally, while previous chapters introduced a more verbose core language, in here we remove all type annotations from our language, including pushing the use of quantifiers to be non-syntax directed. This change will enable, for instance, quantification over capabilities while retaining capabilities as typing artifacts, but will also result in some type system complexity (since it must now figure out on its own when to use quantifiers, etc.). Ensuring the decidability of the full type system will likely still require some annotations, but later chapters only discuss the decidability of the core protocol composition mechanism—which is decidable *without* needing extra annotations.

Thus, this chapter shows how to discipline the use of lock-based shared state through an enhanced and concurrent interpretation of our approach. At its core lays *polymorphic protocol com-*

position, a procedure that ensures safe interference among potentially abstracted protocols that share access to resources. Due to concurrency, proving soundness of the technique requires showing not only the absence of unsafe interference in correctly typed programs (and general *memory safety*) but also ensuring *data-race freedom*.

4.1 A Protocol for Modeling join

The language of this chapter supports the fork/join model of concurrency. However, we do not actually need a join construct since a join can be modeled using shared state, as will be shown below. Our technique is generally agnostic to what causes different alias interleavings, meaning that the alias interleaving caused by thread interleaving fits in similar ways to the interference that may occur within the sequential setting. Still, we review the meaning of our protocols to account for the use of locks that ensure mutual exclusion within the concurrent setting, thus replacing the previous use of focus/defocus.

There are two participants in this interaction: the `Main` thread and the `F`orked thread. The forked thread will compute some *result*. When the main thread joins the forked thread it will *wait* until the result becomes available, if it is not yet ready. Our core primitives to interact with shared state are reading, writing, and locking/unlocking. Because of this, our protocols must explicitly model the "wait for result" cycle of a join.[1] We do not consider the issue of designing a thread scheduler that would reduce, or even eliminate, unnecessary spinning caused by this "busy-wait" cycle.

We can model the two protocols as follows:

$$\begin{aligned} \texttt{F} &\triangleq \mathsf{Wait} \Rightarrow \mathsf{Result}\ ;\ \mathbf{none} \\ \texttt{M} &\triangleq (\ \mathsf{Wait} \Rightarrow \mathsf{Wait}\ ;\ \texttt{M}\ ) \oplus (\ \mathsf{Result} \Rightarrow \mathsf{Done}\ ;\ \mathsf{Done}\ ) \end{aligned}$$

Each protocol contains a sequence of steps that discipline the use of locks and the (type) assumptions on that locked state. Since locks hide all private usages, the protocols will only need to model the public changes done by an alias. The boundary of these changes are a single lock-unlock block which is mapped to a single rely⇒guarantee step in the protocol. When we lock a cell we will *assume* that the state is of some type and, when we eventually unlock that cell, we will *guarantee* that it changed to some other type. Multiple steps can be sequenced using ; to connect two steps.

A forked thread will be given the `F` protocol. This protocol initially assumes that the shared state is of type Wait, on locking. In order to unlock that cell, we must first fulfill the obligation to mutate the state to Result. Once that guarantee is obeyed the protocol continues as **none**, the empty type. This type models termination since the forked thread will never be able to access that shared state again. Note that since subsequent steps may be influenced by the guarantee of the current step, a protocol step is to be interpreted as "Wait ⇒ (Result; **none**)".

The `Main` protocol includes two alternative (⊕) steps that describe different uses of the shared state. If we find the shared cell containing the Wait type then the main thread must leave the state

[1] Each protocol must be aware of all valid states as an omission could leave room for unsafe interference. For instance, if a protocol only listed the states that it considers interesting, we could later apply (ST:ALTERNATIVE) to that protocol and add unsafe steps over the omitted states such that they cause unsafe interference with existing protocols.

with the same type, before later retrying `M`. Otherwise, if we find the cell as Result then we will mutate the cell to Done, before unlocking. Note that by the causality of the changes, the forked thread can no longer access the shared state when the cell reaches Done since `F` must have already terminated by then. This enables us to recover ownership of the shared state. The following step of `M` simply states that Done can return to be used as a regular linear capability, without needing to acquire a lock to access that state. Carefully note that this changes how ownership is recovered when compared to the protocol specification of the previous chapter. Due to the use of locks we must ensure that there is a matching unlock on that cell before recovering ownership. From the technical perspective, this condition ensures that the cell restores the lock which may be needed later, if that cell is later shared again.

We can now define Wait, Result, and Done as types of a single capability to location l, the location that is shared by the two threads, as follows:

$$\begin{aligned} \mathsf{Wait} &\triangleq \mathbf{rw}\ l\ \texttt{Wait\#![]} \\ \mathsf{Result} &\triangleq \mathbf{rw}\ l\ \texttt{Result\#}\mathbf{int} \\ \mathsf{Done} &\triangleq \mathbf{rw}\ l\ \texttt{![]} \end{aligned}$$

Wait is a capability to location l that contains a tagged value where `Wait` is the tag and "![]" (the empty, pure record) is the value. Result is also a capability for l but that contains an integer value tagged as `Result`. The two tags will enable us to check in which alternative the shared state is at by using standard case analysis. Done is simply the capability to location l containing the empty record.

A protocol describes an alias's local view on the evolution of the shared state. Thus, we can discuss the use of each protocol independently. A possible use of the `F` protocol follows. The forked thread will "consume" or "capture" the `F` protocol in its context, making it unavailable to others. The protocol is a linear resource and therefore is tracked by the linear typing environment (Δ) and either used by an expression or threaded through. Furthermore, the expression can use the enclosing lexical typing environment (Γ) because it only contains pure assumptions.

```
3  fork                                  Γ = c : ref l , l : loc , work : ![] ⊸ int  | Δ = F
4    let r = work {} in                  Γ = r : int , ...                           | Δ = Wait ⇒ (Result; none)
5      lock c;                           Γ = ...                                     | Δ = Wait , (Result; none)
6      c := Result#r;                    Γ = ...                                     | Δ = Result , (Result; none)
7      unlock c                          Γ = ...                                     | Δ = none
8    end                                 Γ = ...                                     | Δ = ·
```

We see that Γ contains a reference (`c`) to the location l (the location that is being shared), and a function with the `work` that the thread will do. Line 4 calls the function and stores its result in `r`. At this point we want to update the shared state to signal that the result is ready. Since we are accessing shared state in a multi-threaded environment we first lock the shared location that is being referenced by `c`. Upon locking, `F` is broken down into its two components: the rely type (Wait) and the guarantee type (Result; **none**). While Wait is just the linear capability that now becomes available to use, the guarantee type is an obligation that we promise to fulfill on unlocking. Indeed, line 7 is only valid because the shared state is with the promised type (Result). Once the guarantee is fulfilled, we can move on to the next step of the protocol, **none**. This type is the empty resource that can be automatically discarded leaving Δ empty ($\cdot$).

Given the uncertainty of when a protocol will be used, creating/composing protocols over some shared state must consider all possible ways in which the protocols may be interleaved. Protocol composition ensures full coverage of the different protocol positions that may occur as a result of this interleaving and that all uses of the shared state are safe. Recall that we only consider binary splits when checking safe composition, but since a protocol can be later re-split, there is no limit to how many protocols may share some same state.

Each of the two protocols above contains a finite number of different positions in that protocol. We call a *configuration* the combination of the positions of each protocol and the current type of the shared resources. Each configuration is of the form: $\langle \mathit{State} \Rrightarrow \mathit{Protocol} \parallel \mathit{Protocol} \rangle$. If we split a Wait cell into protocols `M` and `F`, we get the following set of configurations:

$$\{\ ❶ \langle\ \mathsf{Wait} \Rrightarrow \mathtt{M} \parallel \mathtt{F}\ \rangle\ ,\ ❷ \langle\ \mathsf{Result} \Rrightarrow \mathtt{M} \parallel \mathbf{none}\ \rangle\ ,$$
$$❸ \langle\ \mathsf{Done} \Rrightarrow \mathsf{Done} \parallel \mathbf{none}\ \rangle\ ,\ ❹ \langle\ \mathbf{none} \Rrightarrow \mathbf{none} \parallel \mathbf{none}\ \rangle\ \}$$

Stepping of configurations follows the same principles of the previous chapter. Configuration ❶ represents the initial split of Wait into `M` and `F`. Starting from that configuration, we will leave one of the protocols stationary while we simulate a use of the shared state (a *step*) with the remaining protocol. From ❶ if we step `M` we will stay in the same configuration. If instead `F` is stepped, we get to configuration ❷ that changed the state to Result and terminates the `F` protocol. By continuing to step `M` we have the two last configurations: ❸ where `M` is ready to recover ownership, and ❹ where the ownership of the shared state was recovered and all protocols have terminated.

Upon sharing, the ownership of the state belongs to all intervening protocols. Ownership recovery means that this ownership transitions to the terminating protocol. However, to be safe, we must be sure that this permanent ownership transfer only occurs on the *last* protocol to terminate, ensuring that no other protocol may attempt to see that state as shared. The ownership recovery in ❸ transfers Done from the shared state to the alias that uses the `M` protocol. We also see by ❹ that this stepping consumes both the shared state (leaving it as **none**) and that residual end-step of the `M` protocol (leaving that protocol position also as **none**).

All non-terminated protocols in the configurations above can take a step. In fact, even **none** can take a vacuous step that remains in the same configuration since **none** cannot change the shared state. Therefore, each protocol will always find an expected state in the shared cell regardless of how protocols are interleaved—i.e. all interference is safe since no configuration is stuck. A stuck configuration occurs when at least one of the protocols cannot take a step with the current type of the shared state. For instance, $\langle \mathsf{Result} \Rrightarrow \mathtt{M} \parallel \mathtt{F} \rangle$ cannot take a step with `F` since `F` does not assume Result in any of its current steps. If such stuck configurations were allowed to occur, then a program could fault due to unexpected values stored in shared cells or due to attempts to access cells that were destroyed using wrong assumptions of ownership recovery.

We now show the rest of the join encoding:

```
1 let newFork = λ(work).
2   let c = new Wait#{} in
3     fork ... // Lines 3 to 8 were shown above.
```

Γ = work : ![] ⊸ **int**	| Δ = ·
Γ = *l* : **loc** , c : **ref** *l* , ...	| Δ = **rw** *l* Wait#![]
Γ = ...	| Δ = M , F

To simplify the presentation, our term language is stripped of type annotations. The newFork function, that creates a join function based on the provided work function, has type:

$$!(\ \underbrace{!(\ ![] \multimap \mathbf{int}\ )}_{\text{the work}} \multimap \underbrace{(\ ![] \multimap \mathbf{int}\ )}_{\text{the join}}\ )$$

where the pure (!) function ($\multimap$) of the argument is the work to be done by the thread, as was shown above. Line 2 creates the cell that will be shared by the main and forked threads. This new cell, although typed as $\exists l.(\ !(\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ \texttt{Wait\#}![])\ )$, is automatically opened by the type system to allow direct access to the reference. Line 3 shares the cell (again, done automatically by the type system) using the split that was discussed above on the capability to location l:

$$(\mathbf{rw}\ l\ \texttt{Wait\#}![]) \Rrightarrow \texttt{M} \parallel \texttt{F}$$

This split results in the capability to location l being replaced by the two protocols, `M` and `F`, in Δ (line 3).

Finally, we show the join function that will "busy-wait" for the forked thread to produce a result. Its use of both **rec**ursion and case analysis should be straightforward as they follow identical usages from previous chapters, and so the text will focus on the less obvious details.

```
9  λ_.rec R.
10
11     lock c;
12     case !c of
13       Wait#x →
14         c := Wait#x;
15         unlock c;
16         R // retries
17     | Result#x →
18         unlock c;
19         delete c;
20         x
21     end
22   end
23 end
```

Δ = (Wait ⇒ (Wait; M)) ⊕ (Result ⇒ (Done; Done))

[a]Δ = Wait ⇒ (Wait; M) **[b]**Δ = Result ⇒ (Done; Done)

[a]Δ = Wait (**rw** l Wait#![]), (Wait; M) **[b]**Δ = Result (**rw** l Result#**int**), (Done; Done)

[a]Δ = **rw** l ![], (Wait; M) **[b]**Δ = **rw** l ![], (Done; Done)

[a]Δ = **rw** l ![], (Wait; M)

[a]Δ = Wait, (Wait; M)

[a]Δ = M

[b]Γ = x : **int**, ... | Δ = **rw** l ![], (Done; Done)

[b]Γ = x : **int**, ... | Δ = **rw** l ![]

[b]Γ = x : **int**, ... | Δ = ·

We omit Γ to center the discussion on the contents of Δ. The alternative type ($\oplus$) lists a union of types that may be valid at that point in the program. To use such a type, an expression must consider each alternative individually. The breakdown of $\oplus$ (line 10) is done automatically by the type system, by using (T:Alternative-Left) as discussed in previous chapters. Thus, the body of the recursion must be typed individually under each one of those alternatives, marked as **[a]** and **[b]**. Note how the tag in the sum type matches different branches in the case depending on the alternative. This enables different uses, that may require obeying incompatible guarantees, based solely on the tagged contents of the shared state. For instance the `Result` branch will recover ownership and destroy the shared cell (line 19), while the `Wait` branch must restore the linear value of that cell (that was removed by the linear dereference of line 12, that left "**rw** l ![]" in Δ) before retrying.

Finally, note that we do not address the issue of deadlock avoidance which means that a correctly typed program may still deadlock. This is a (somewhat) different design choice from what

was done in the previous chapter. Indeed, in chapter 3 we statically ruled out the presence of unsafe re-entrant uses of shared state. However, it is not enough to simply adapt such a rule to the concurrent setting to ensure deadlock-freedom. For instance, the conditions of chapter 3 do not ensure consistent "lock" acquisition order. Instead, we simply do not perform such checks in the system of this chapter which may leave a correctly typed program to deadlock at run-time.

4.2 Polymorphic Protocol Composition

Protocol composition, as introduced so far, forces both protocols to have the same (full) visibility on the contents of the shared resources. Ideally, each protocol should only depend on the types that are relevant to the operations done through that alias. For instance, the operation done through the `F` protocol of the previous Section does not need to know the precise type (Wait) that is initially stored in location l. Thus, we want to be able to abstract that type (as X) as follows:

$$\exists X.(\ \mathbf{rw}\ l\ X\ \Rightarrow\ (\ \mathbf{rw}\ l\ \texttt{Result\#}\mathbf{int}\ ;\ \mathbf{none}\ )\ )$$

Similarly, the wait step of the `M` protocol only depends on the tag of the shared cell enabling everything else to be abstracted as:

$$\exists X.(\ \mathbf{rw}\ l\ \texttt{Wait\#}X\ \Rightarrow\ (\ \mathbf{rw}\ l\ \texttt{Wait\#}X\ ;\ \texttt{M}\ )\ )\ \oplus\ \ldots$$

Enabling protocols to abstract part of their uses of the shared state improves modularity and increases flexibility, but also brings new challenges on ensuring safe protocol composition. We will focus the discussion on two new aliasing idioms that this kind of quantification enables: a) *existential-universal interaction*, how a universally quantified guarantee can interact with an existentially quantified rely; and b) *step extensions over abstractions*, how abstractions enable existing protocol steps to be re-split (i.e. nested protocol re-splitting) without the risk of introducing unsafe interference.

4.2.1 Existential-Universal Interaction

Enabling a step to existentially quantify the type initially stored in a cell will naturally decouple that protocol from the effects produced on the shared state by other protocols. Therefore, in this situation, a guarantee can also be more "generic" by allowing a greater range of changes to be done to that shared state. We now discuss the core ideas that enable safe existential-universal protocol splittings.

Consider the following protocols that share access to location p:

$$\begin{array}{lcl}\texttt{Nothing} & \triangleq & \exists X.(\ (\mathbf{rw}\ p\ X) \Rightarrow (\mathbf{rw}\ p\ X)\ ;\ \texttt{Nothing}\ ) \\ \texttt{Full}[Y] & \triangleq & (\mathbf{rw}\ p\ Y) \Rightarrow \forall Z.(\ (\mathbf{rw}\ p\ Z)\ ;\ \texttt{Full}[Z]\ )\end{array}$$

The `Nothing` protocol is defined using X to abstract the contents of the shared cell on a single step, guaranteeing that X is restored before repeating the protocol. Thus, `Nothing` is not allowed

to publicly modify the shared state, although it can undergo private changes. Conversely, `Full` is able to arbitrarily modify the shared state by allowing its clients to pick any type to apply to the guarantee. `Full` itself is parametric on the type that is currently stored in the shared cell, Y. Each step of `Full` can exploit the precise local information on how the state was modified, by remembering its own changes to cell p. Naturally, to enable this freedom, ensuring protocol composition requires checking the changes done by `Full` abstractly.

To illustrate how composition works in this case, consider the following split where p initially holds a value of type **`int`**:

$$(\mathbf{rw}\ p\ \mathtt{int}) \ \Rrightarrow\ \mathtt{Full}[\mathtt{int}] \parallel \mathtt{Nothing}$$

Protocol composition results in the following set of configurations:

$$\begin{array}{l}\{\ ❶\ \langle \quad p : \mathbf{loc} \vdash (\mathbf{rw}\ p\ \mathtt{int}) \Rrightarrow \mathtt{Full}[\mathtt{int}] \parallel \mathtt{Nothing}\ \rangle\ , \\ \quad ❷\ \langle\, Z : \mathbf{type}, p : \mathbf{loc} \vdash (\mathbf{rw}\ p\ Z) \ \Rrightarrow\ \mathtt{Full}[Z] \parallel \mathtt{Nothing}\ \rangle\ \}\end{array}$$

The use of abstractions will require composition to track type (and location) variables in configurations. For this reason, configurations must be extended to include a lexical typing environment, Γ, to track these variables on the left of $\vdash$. Configuration ❶ is the initial configuration given by the split above, which includes the assumption that p is a known **loc**ation.

To step `Nothing` from ❶, we must first find a representation type to open the existential. This type is found by unifying the current state of the shared state (**rw** p **`int`**) with the rely type of `Nothing`. Thus, we see that X is abstracting **`int`**. After we open the existential by exposing the **`int`** type, we see that the step will preserve **`int`** resulting in `Nothing` yielding the same ❶ configuration:

$$\dfrac{\dfrac{\dfrac{}{\langle\ldots\vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathcal{R}[\ (\ \mathbf{rw}\ p\ X \Rightarrow \mathbf{rw}\ p\ X\ ;\ \mathtt{Nothing}\ )\{\mathtt{int}/X\}\ ]\ \rangle \ \mapsto\ \langle\ldots\vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathcal{R}[\ \mathtt{Nothing}\ ]\ \rangle}}{\langle\ldots\vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathcal{R}[\ \exists X.(\ \mathbf{rw}\ p\ X \Rightarrow \mathbf{rw}\ p\ X\ ;\ \mathtt{Nothing}\ )\ ]\ \rangle \ \mapsto\ \langle\ldots\vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathcal{R}[\ \mathtt{Nothing}\ ]\ \rangle}}{❶\ \langle\ldots\vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathcal{R}[\ \mathtt{Nothing}\ ]\ \rangle \ \mapsto\ \langle\ldots\vdash \mathbf{rw}\ p\ Z \Rrightarrow \mathcal{R}[\ \mathtt{Nothing}\ ]\ \rangle\ ❶}$$

To step ❶ with `Full`[**`int`**], we must consider that its resulting guarantee is abstract. The new configuration, ❷, must consider a fresh type variable to represent that new type that a client can pick. Thus, stepping results in the following derivation:

$$\dfrac{\dfrac{}{\langle\ldots\vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathcal{R}[\ \mathbf{rw}\ p\ \mathtt{int} \Rightarrow \forall Z.(\ \mathbf{rw}\ p\ Z\ ;\ \mathtt{Full}[Z]\ )\ ]\ \rangle \ \mapsto\ \langle\ldots, Z : \mathbf{type} \vdash \mathbf{rw}\ p\ Z \Rrightarrow \mathcal{R}[\ \mathtt{Full}[Z]\ ]\ \rangle}{❶\ \langle\, p : \mathbf{loc} \vdash \mathbf{rw}\ p\ \mathtt{int} \Rrightarrow \mathcal{R}[\ \mathtt{Full}[\mathtt{int}]\ ]\ \rangle \ \mapsto\ \langle\ldots, Z : \mathbf{type} \vdash \mathbf{rw}\ p\ Z \Rrightarrow \mathcal{R}[\ \mathtt{Full}[Z]\ ]\ \rangle\ ❷}$$

In this case, we used Z to represent that new type. It is easy to see that if we were to step `Nothing` from ❷ we would remain in configuration ❷ following similar reasoning to that done for ❶. The surprising aspect is that further steps with `Full` will also yield configurations that are *equivalent* to ❷.

The typing environment plays a crucial role in enabling us to close the safe composition proof. Although each step of `Full` considers a fresh type, stepping results in configurations that are equivalent, up to renaming of variables and weakening of Γ. Weakening allows us to ignore variables

that no longer occur free in a configuration. This means that further steps with `Full` result in configurations that are equivalent to already seen configurations. Thus, although the set of different types that can be applied to `Full` is infinite, the number of *distinct interactions* that can legally occur through that shared state is finite if we model those interactions abstractly.

Bounded Quantification We use bounded quantification to provide more expressive abstractions that go beyond the fully opaque types used above (which are equivalent to a "<: **top**" bound).

By using appropriate bounds, we can give concrete roles to the `Nothing` and `Full` protocols. Consider that we want to share access to some data structure among several different threads. However, depending on how these threads dynamically use that data structure, it may become important to switch its representation (such as change from a linked list to a binary tree, etc.). Furthermore, we want one specialized thread (the C*ontroller*) to retain precise control over the data structure and to be allowed to monitor and change its representation. Concurrently, an arbitrary number of other threads (the W*orkers*) also have access to the data structure but are limited to only access its basic operations.

$$\begin{array}{lcl} \texttt{W} & \triangleq & \exists X <: \texttt{DS}.(\ (\textbf{rw}\ p\ X) \Rightarrow (\textbf{rw}\ p\ X)\ ;\ \texttt{W}\ ) \\ \texttt{C}[Y] & \triangleq & (\textbf{rw}\ p\ Y) \Rightarrow \forall Z <: \texttt{DS}.(\ (\textbf{rw}\ p\ Z)\ ;\ \texttt{C}[Z]\ ) \end{array}$$

As before, `W` is committed to preserve the representation type of X, but it now has sufficient permission to use that type as `DS`. `C` is now more constrained than before since it is forced to guarantee a type that is compatible with `DS`. However, `C` retains the possibility of both changing the representation type contained in the shared state, and also of "remembering" the precise (representation) type that was the result of its own local action. Finally, note that we can safely re-split `W` arbitrarily (i.e. $\texttt{W} \Rrightarrow \texttt{W} \parallel \texttt{W}$). Protocol composition yields a similar set of configurations, but with the bound assumption on Z.

Monotonic Counter Although our discussion here is focused on the core interference concepts, these basic mechanisms are at the core of more precise forms of interference. For instance, on interfering monotonic counters [49, 74] that share state symmetrically. However, for this example, we must consider an informal extension to specify the more precise refined states of the following protocol:

$$\texttt{MC} \triangleq \exists\{j : \textbf{int}\}.(\ (\textbf{rw}\ p\ j) \Rightarrow \forall\{i : \textbf{int} \mid i \geq j\}.(\ (\textbf{rw}\ p\ i)\ ;\ \texttt{MC}\ )\ )$$

The protocol models a monotonically increasing counter on location p. The step relies on location p initially containing some integer, j, and modifying the cell to store some other value, i, that is greater or equal than j. This interaction can be reduced to a core existential-universal interaction discussed above, and where the protocol can be re-split indefinitely:

$$\texttt{MC'} \triangleq \exists J.(\ (\textbf{rw}\ p\ J) \Rightarrow \forall I.(\ (\textbf{rw}\ p\ I)\ ;\ \texttt{MC'}\ )\ )$$

Adding support for dependent refinement types is beyond the scope of this work, and we believe an orthogonal issue to the core of interference control. The discussion above is meant to contextualize our claims on our focus on the core interference phenomenon.

4.2.2 Inner Step Extension

Specializing an existing protocol is possible, provided that its effects remain consistent with those of the original protocol. Namely we can append new steps to an otherwise ownership recovery step, or produce effects that are more precise than those of the original protocol. The first case allows us to connect two protocols together by that recovery step, in analogous ways to what was shown in the previous chapter. The latter case is more interesting: when combined with abstraction it allows specialization *within* an existing step. To illustrate the expressiveness gains, we revisit the join protocol of Section 4.1. However, instead of spawning a single thread to compute the work, we re-split the join protocol in two symmetric workers that share the workload. The last of the workers to complete merges the two results together and "signals" the waiting main thread.

First, we must rewrite the two protocols to enable abstraction on the `M` protocol, and add a choice (&) to the `F` protocol that enables `F` to use the state more than once until it provides a result.

$$\texttt{F}[X] \triangleq (\ (\textbf{rw}\ p\ \texttt{W\#}X) \Rightarrow \forall Y.(\ (\textbf{rw}\ p\ \texttt{W\#}Y)\ ;\ \texttt{F}[Y]\ )\ )\ \&\ (\ (\textbf{rw}\ p\ \texttt{W\#}X) \Rightarrow (\textbf{rw}\ p\ \texttt{R\#}\textbf{int})\ ;\ \textbf{none}\ )$$
$$\texttt{M} \quad \triangleq \exists Z.(\ (\textbf{rw}\ p\ \texttt{W\#}Z) \Rightarrow (\textbf{rw}\ p\ \texttt{W\#}Z)\ ;\ \texttt{M}\ )\ \oplus\ (\ (\textbf{rw}\ p\ \texttt{R\#}\textbf{int}) \Rightarrow (\textbf{rw}\ p\ \textbf{int})\ ;\ (\textbf{rw}\ p\ \textbf{int})\ )$$

As before, `M` will `Wait` until there is a `Result` in p. At that point, `M` will recover ownership of that cell. Unlike before, `M` no longer depends on the value tagged as `W` since it is abstracted as Z. The `F` protocol now holds two choices (&): the old step that transitions from `Wait` to `Result`, and a new step that changes the representation of the value tagged as `W` and used during the `Wait` phase. The `F` protocol of Section 4.1 is a specialization of this protocol since it includes only one of the choices. We now specialize `F` into two symmetric worKer protocols. To simplify the presentation, we assume that the worker thread will receive the work parameters through some other mean (such as a pure value shared among threads). Once a worker finishes its job, it will push the resulting **`int`** to the shared state. If it notices it is the last worker to finish, it will merge the two results and flag the state as ready, so that `Main` can proceed.

$$\texttt{K} \triangleq (\ (\textbf{rw}\ p\ \texttt{W\#([])}) \Rightarrow (\textbf{rw}\ p\ \texttt{W\#(R\#}\textbf{int}\texttt{)})\ ;\ \textbf{none}\ )\ \oplus\ (\ (\textbf{rw}\ p\ \texttt{W\#(R\#}\textbf{int}\texttt{)}) \Rightarrow (\textbf{rw}\ p\ \texttt{R\#}\textbf{int})\ ;\ \textbf{none}\ )$$

It is important to note that the new tags/values are nested *inside* the old `Wait` tag. This ensures that the new usages remain hidden from `M` and "look" just like the previous `F` usage. However, these inner tags are used by the two workers for coordination: the `W#Empty` tag means that neither thread has finished, and `W#Result` means that one of the threads has finished. We can then re-split `F` as follows (note the required initial type in `F`, `E#[]`, for this split to be valid):

$$\texttt{F[E\#[]]} \Rrightarrow \texttt{K} \parallel \texttt{K}$$

Protocol composition proceeds similarly to above, except that we are now simulating the steps of the new `K` protocols in the `F` protocol instead of stepping states:

$$\begin{array}{l}\{\ \textbf{❶}\ \langle\ \cdot \vdash \texttt{F[E\#[]]} \Rrightarrow \texttt{K} \parallel \texttt{K}\ \rangle\ , \\ \quad \textbf{❷}\ \langle\ \cdot \vdash \texttt{F[R\#}\textbf{int}\texttt{]} \Rrightarrow \texttt{K} \parallel \textbf{none}\ \rangle\ , \\ \quad \textbf{❸}\ \langle\ \cdot \vdash \texttt{F[R\#}\textbf{int}\texttt{]} \Rrightarrow \textbf{none} \parallel \texttt{K}\ \rangle\ , \\ \quad \textbf{❹}\ \langle\ \cdot \vdash \textbf{none} \Rrightarrow \textbf{none} \parallel \textbf{none}\ \rangle\ \}\end{array}$$

We only show the derivation on stepping from ❶ to ❷. For (some) conciseness, we omit "**rw** p" in the next derivation showing only the type contained in that capability.

$\langle \cdot \vdash$ W#(E#[]) $\Rightarrow$ (W#Y ; F[Y]){R#**int**/Y}	$\Rrightarrow \mathcal{R}[$ W#(E#[]) $\Rightarrow$ W#(R#**int**)] $\rangle$	$\mapsto \langle \cdot \vdash$ F[R#**int**] $\Rrightarrow \mathcal{R}[$ **none**] $\rangle$
$\langle \cdot \vdash$ W#(E#[]) $\Rightarrow \forall Y.($ W#Y ; F[Y])	$\Rrightarrow \mathcal{R}[$ W#(E#[]) $\Rightarrow$ W#(R#**int**)] $\rangle$	$\mapsto \langle \cdot \vdash$ F[R#**int**] $\Rrightarrow \mathcal{R}[$ **none**] $\rangle$
$\langle \cdot \vdash$ W#(E#[]) $\Rightarrow \forall Y.($ W#Y ; F[Y])	$\Rrightarrow \mathcal{R}[$ W#(E#[]) $\Rightarrow$ W#(R#**int**) $\oplus$ ...] $\rangle$	$\mapsto \langle \cdot \vdash$ F[R#**int**] $\Rrightarrow \mathcal{R}[$ **none**] $\rangle$
$\langle \cdot \vdash$ W#(E#[]) $\Rightarrow \forall Y.($ W#Y ; F[Y]) & ...	$\Rrightarrow \mathcal{R}[$ K] $\rangle$	$\mapsto \langle \cdot \vdash$ F[R#**int**] $\Rrightarrow \mathcal{R}[$ **none**] $\rangle$
❶ $\langle \cdot \vdash$ F[E#[]]	$\Rrightarrow \mathcal{R}[$ K] $\rangle$	$\mapsto \langle \cdot \vdash$ F[R#**int**] $\Rrightarrow \mathcal{R}[$ **none**] $\rangle$ ❷

Each simulation will match the rely and guarantee types of a step in F with a step in K, even if specializing a $\forall$ of F to a specific type in K. As before, K can choose which step to simulate when given a choice (&) of F steps. Similarly, at least one alternative ($\oplus$) of K must match a step in F. Therefore, the new K protocols work within the interference of the original F protocol, but specialize its steps.

4.3 Technical Development

For the purpose of detailing how the extensions to protocol composition work, we only need to consider the simplified types grammar and lexical typing environment of Figure 4.1. Most of these types were already introduced in the examples above and in previous chapters. We recall that $\oplus$ is our union type that models an internal choice, and & is an intersection type that offers a choice of uses to clients. **none** is the empty resource, which can also be interpreted as the "empty capability" or a "terminated protocol". Existential and universal quantification is modified to include a type bound on the right of "<:". As before, recursive types are equi-recursive such that the following unfolding principle holds:

$$(\textbf{rec}\ X(\overline{u}).A)[\overline{U}] = A\{(\textbf{rec}\ X(\overline{u}).A)/X\}\{\overline{U}/\overline{u}\} \qquad \text{(EQ:REC)}$$

Recursive types must be non-bottom and include a list of type/location parameters ($\overline{u}$) that are substituted by some type/location ($\overline{U}$) on unfold, besides unfolding the recursion variable (X).

We now discuss the technical details of checking safe protocol composition, within the concurrent setting. As before, a binary protocol split will generate an infinite binary tree representing all combinations of interleaved uses of the two new protocols. Each node of that tree has two children based on which protocol remains stationary while the other is stepped. Since this tree may be infinite, protocol composition must build a co-inductive proof of safe interference. The challenge is then on ensuring that such a check remains tractable and decidable, but still allows many interesting shared state interactions to occur. Besides extending the expressiveness of protocol composition, the changes to support concurrency are mostly related to how ownership recovery differs and the well-formed conditions that ensure an unlock matches the same set of locations that the respective lock listed.

$l \in$ Location Variables $\quad X \in$ Type Variables $\quad \rho \in$ Location Constants (Addresses)

$$p ::= \rho \mid l \qquad u ::= l \mid X \qquad U ::= p \mid A$$

A	::=	...	
	|	**none**	(empty resource)
	|	$A \Rightarrow A$	(rely)
	|	$A; A$	(guarantee)
	|	$A \oplus A$	(alternative)
	|	$A \,\&\, A$	(intersection)
	|	$X[\overline{U}]$	(type variable)
	|	(**rec** $X(\overline{u}).A)[\overline{U}]$	(recursive type)
	|	$\forall l.A$	(universal location quantification)
	|	$\exists l.A$	(existential location quantification)
	|	$\forall X <: A.A$	(bounded universal type quantification)
	|	$\exists X <: A.A$	(bounded existential type quantification)
Γ	::=	...	
	|	$\cdot$	(empty)
	|	$\Gamma,\ p$: **loc**	(location assertion)
	|	$\Gamma,\ X <: A$	(bound assertion)
	|	$\Gamma,\ X$: **type**	(type assertion)

Notes: $\overline{Z}$ is a potentially empty sequence of Z elements.

Figure 4.1: Relevant extensions to types grammar and typing environment.

4.3.1 Well-Formed Protocols

The grammar of *P*rotocols and shared *R*esources is now extended to enable abstraction on the protocols themselves (see Figure 4.2). This enables the kind of polymorphic protocol specification discussed above. However, we must also impose additional well-formedness conditions on the protocols; some conditions are enforced via protocol composition (but clarified here) and others via well-formed rules:

Protocol Structure The protocol syntax is restricted to limit where quantification may appear during the composition check. Existential quantification over the contents of protocols can only occur at the rely type level, such as "$\exists X.(... \Rightarrow ...)$", etc. Similarly, universal quantification may only occur at the guarantee type level, such as "$\forall X.(...; ...)$", etc. Other uses of quantification on the structure of protocols are forbidden since they may excessively abstract the kind of interference that a protocol can do. After the safe composition check, protocols can be abstracted just like any other type in the language.

Lock Set Intuitively, when we lock a set of locations we must ensure that it is the same set of locations that is later unlocked. This condition ensures that the run-time lock mechanism (a

$$
\begin{array}{rcl}
P, Q & ::= & (\textbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \\
& | & X[\overline{U_P}] \\
& | & P \oplus P \\
& | & P \,\&\, P \\
& | & R \Rightarrow P \\
& | & R\,;\ P \\
& | & \textbf{none} \\
& | & \exists l.P \\
& | & \forall l.P \\
& | & \exists X <: A.P \\
& | & \forall X <: A.P
\end{array}
\qquad
\begin{array}{rcl}
R & ::= & (\textbf{rec}\ X(\overline{u}).R)[\overline{U_R}] \\
& | & X[\overline{U_R}] \\
& | & R \oplus R \\
& | & R \,\&\, R \\
& | & A * A \\
& | & \textbf{rw}\ p\ A \\
& | & \textbf{none} \\
& | & \exists l.R \\
& | & \forall l.R \\
& | & \exists X <: A.R \\
& | & \forall X <: A.R
\end{array}
$$

$$
S \quad ::= \quad R \mid P
$$

Notes: As in previous chapters, recursive types are automatically unfolded, through (EQ:REC). Also note that other well-formed conditions apply, such as ensuring that the rely and guarantee types contain the same (non-empty) set of locations to lock/unlock. The stepping rules also constrain where $\exists$ and $\forall$ may appear on protocols.

Figure 4.2: Grammar restrictions for checking protocol composition.

token) is preserved. Thus, any unlock (guarantee type) must list the same set of (non-empty) locations that the respective lock (rely type) acquired. Basically:

$$\text{“}A_0 \Rightarrow A_1\ ;\ ...\text{”} \quad \text{is well-formed if} \quad \textbf{locs}(A_0) = \textbf{locs}(A_1) \neq \emptyset$$

Note that **locs** does not consider types “nested” inside some other state, so that $\textbf{locs}(\textbf{rw}\ p\ (\textbf{ref}\ q)) = \{p\}$. Likewise, **locs** also ignores protocols that may appear in A_0 so that their locks must be acquired separately from the locks of the current step.

The lock set invariant must also be obeyed when there are abstractions using $\exists$ or $\forall$. While abstracting locations will simply result in fresh names, abstracting types is more dangerous. For instance, a $\forall$ could be instantiated to a type that increases the set of locations of a step. To forbid this, we require that the type bound of a non-“nested” type variable to be a protocol. Thus, its lock set will not affect the lock set of the current step.

4.3.2 Protocol Composition Rules

Protocol composition follows the same definition as in the previous chapter. However, we must now include a lexical environment so that a configurations is now of the form:

$$\Gamma \ \vdash\ S \Rrightarrow P \parallel Q$$

Thus, the set of configurations C is defined as:

$$
\begin{array}{rcll}
C & ::= & \langle\ \Gamma \vdash S \Rrightarrow P \parallel Q\ \rangle & \text{(configuration)} \\
& | & C \cdot C & \text{(union)}
\end{array}
$$

$\boxed{C \mapsto C}$ **Composition, (C:*)**

$$\frac{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}_L[P] \,\rangle \mapsto C_0 \qquad \mathcal{R}_L[\square] = \square \parallel Q \quad \langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}_R[Q] \,\rangle \mapsto C_1 \qquad \mathcal{R}_R[\square] = P \parallel \square}{\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto C_0 \cdot C_1} \text{ (C:STEP)}$$

$$\frac{C_0 \mapsto C_2 \quad C_1 \mapsto C_3}{C_0 \cdot C_1 \mapsto C_2 \cdot C_3} \text{ (C:ALLSTEP)}$$

Composition — Reduction Step, (C-RS:*)

$$\frac{}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[\textbf{none}] \,\rangle \mapsto \langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[\textbf{none}] \,\rangle} \text{ (C-RS:NONE)}$$

$$\frac{\langle\, \Gamma_0 \vdash S \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, \Gamma_0, \Gamma_1 \vdash S \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C} \text{ (C-RS:WEAKENING)}$$

$$\frac{\langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \quad \langle\, \Gamma \vdash S_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_1}{\langle\, \Gamma \vdash S_0 \oplus S_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \cdot C_1} \text{ (C-RS:STATEALTERNATIVE)}$$

$$\frac{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P_0 \oplus P_1] \,\rangle \mapsto C} \text{ (C-RS:PROTOCOLALTERNATIVE)}$$

$$\frac{\langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S_0 \& S_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C} \text{ (C-RS:STATEINTERSECTION)}$$

$$\frac{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C_0 \quad \langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C_1}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P_0 \& P_1] \,\rangle \mapsto C_0 \cdot C_1} \text{ (C-RS:PROTOCOLINTERSECTION)}$$

Figure 4.3: Updated protocol stepping rules (1/3).

Composition — State Stepping, (c-ss:*)

(c-ss:Step)
$$\frac{}{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[R_0 \Rightarrow R_1; P] \,\rangle \mapsto \langle\, \Gamma \vdash R_1 \Rrightarrow \mathcal{R}[P] \,\rangle}$$

(c-ss:Recovery)
$$\frac{}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[R] \,\rangle \mapsto \langle\, \Gamma \vdash \mathbf{none} \Rrightarrow \mathcal{R}[\mathbf{none}] \,\rangle}$$

(c-ss:OpenLoc)
$$\frac{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P\{p/l\}] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[\exists l.P] \,\rangle \mapsto C}$$

(c-ss:OpenType)
$$\frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P\{A_1/X\}] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[\exists X <: A_0.P] \,\rangle \mapsto C}$$

(c-ss:ForallLoc)
$$\frac{\langle\, \Gamma, l : \mathbf{loc} \vdash R \Rrightarrow \mathcal{R}[R \Rightarrow P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[R \Rightarrow \forall l.P] \,\rangle \mapsto C}(l \text{ f.i.c.})$$

(c-ss:ForallType)
$$\frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash R \Rrightarrow \mathcal{R}[R \Rightarrow P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[R \Rightarrow \forall X <: A.P] \,\rangle \mapsto C}(X \text{ f.i.c.})$$

$P\{A/X\} \triangleq$ "substitution, in P, of X for A" f.i.c. $\triangleq$ "fresh in conclusion"

Figure 4.4: Protocol composition rules (2/3).

Composition — Protocol Stepping, (C-PS:*)

(C-PS:STEP)
$$\frac{}{\langle\, \Gamma \vdash (R_0 \Rightarrow R_1; Q) \Rrightarrow \mathcal{R}[R_0 \Rightarrow R_1; P] \,\rangle \mapsto \langle\, \Gamma \vdash Q \Rrightarrow \mathcal{R}[P] \,\rangle}$$

(C-PS:EXISTSTYPE)
$$\frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash P \Rrightarrow \mathcal{R}[Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash \exists X <: A.P \Rrightarrow \mathcal{R}[\exists X <: A.Q] \,\rangle \mapsto C}$$

(C-PS:EXISTSLOC)
$$\frac{\langle\, \Gamma, l : \mathbf{loc} \vdash P \Rrightarrow \mathcal{R}[Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash \exists l.P \Rrightarrow \mathcal{R}[\exists l.Q] \,\rangle \mapsto C}$$

(C-PS:FORALLTYPE)
$$\frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash R \Rightarrow P \Rrightarrow \mathcal{R}[R \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rightarrow \forall X <: A.P \Rrightarrow \mathcal{R}[R \Rightarrow \forall X <: A.Q] \,\rangle \mapsto C}(X \text{ f.i.c.})$$

(C-PS:FORALLLOC)
$$\frac{\langle\, \Gamma, l : \mathbf{loc} \vdash R \Rightarrow P \Rrightarrow \mathcal{R}[R \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[R \Rightarrow \forall l.Q] \,\rangle \mapsto C}(l \text{ f.i.c.})$$

(C-PS:LOCAPP)
$$\frac{\langle\, \Gamma \vdash R \Rightarrow P\{p/l\} \Rrightarrow \mathcal{R}[R \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[R \Rightarrow Q] \,\rangle \mapsto C}(l \text{ f.i.c.})$$

(C-PS:TYPEAPP)
$$\frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\, \Gamma \vdash R \Rightarrow P\{A_1/X\} \Rrightarrow \mathcal{R}[R \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rightarrow \forall X <: A_0.P \Rrightarrow \mathcal{R}[R \Rightarrow Q] \,\rangle \mapsto C}(X \text{ f.i.c.})$$

Figure 4.5: Protocol composition rules (2/3).

Which results in the following updated rule for (WF:SPLIT), while (WF:CONFIGURATION) remains unchanged but refers to the new configuration format:

$$\frac{\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle\!\uparrow}{\Gamma \vdash S \Rrightarrow P \parallel Q}\ (\text{WF:SPLIT}) \qquad \frac{C_0 \mapsto C_1 \qquad C_1\!\uparrow}{C_0\!\uparrow}\ (\text{WF:CONFIGURATION})$$

We now go over the rules of Figures 4.3, 4.4, and 4.5, highlighting the main changes to the definition of protocol composition used in the previous chapter. Recall that R denotes a type that is not a protocol. Any occurrence of a rely type can only occur in its structure if "nested". This difference enables us to precisely distinguish between splitting some state into two protocols and splitting an existing protocol into two new protocols.

We use three distinct label prefixes to group the stepping rules based on whether a rule is stepping over a protocol (C-PS:*), stepping over a non-protocol (C-SS:*), or is applicable in both cases (C-RS:*).

The main addition to the (C-RS:*) rules is step weakening. Weakening on a configuration, (C-RS:WEAKENING), is the crucial mechanism that enables us to close the co-inductive proof when using quantifiers. Thus, when we reach a configuration that is equivalent up to renaming of variables and weakening of Γ, we can close the proof. The remaining rules of that set remain essentially the same, up to all configurations now considering Γ.

The base rule for state stepping, (C-SS:STEP), remains the same as it transitions the step of the protocol and changes the state to reflect the guarantee state of the protocol. The remaining rules of that figure are new. The (C-SS:FORALL*) rules do similar stepping but considering an abstracted guarantee, which results in a typing environment with the opened abstraction. (C-SS:OPEN*) exposes the representation type/location (if it exists) before doing a regular step. (C-SS:RECOVERY) transfers the ownership of the resource to the terminating protocol that recovers ownership. Carefully note that this additional rule was not needed in the previous chapter since recovery was encoded differently. Protocol stepping rules, (C-PS:*), use the same base stepping case. (C-PS:FORALL*) and (C-PS:EXISTS*) open their respective abstraction before doing a regular simulation step. More interestingly, (C-PS:*APP) enables a simulated step to pick a particular type/location to apply before that regular simulation stepping.

Carefully note that the rules above enable the re-splitting of a protocol by extending an ownership recovery step (besides inner step specialization). In this situation, we have that the simulation of the original protocol will seamlessly switch from the protocol stepping rules to the state stepping rules.

The subtyping extensions are analogous to the one shown in the previous chapter, in Figure 3.4. Since the rules are modified to include Γ, we include the changed rules in Figure 4.6, which must also include the stepping rule to support the modified ownership recovery semantics.

We now look at the remaining aspects of the system. As before, the grammar is in let-normal form and is listed in Figure 4.7. The main changes to previous chapters is that our expressions no longer carry any type information. This means that quantification disappeared from the syntax, as we no longer have explicit constructs for packing/opening locations, etc. All these operations are now done automatically by the type system, instead of being syntax-directed, as we focus the

(C-RS:SUBSUMPTION)

$$\frac{\Gamma \vdash S_1 <: S_0 \qquad \langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C \qquad \Gamma \vdash P_0 <: P_1}{\langle\, \Gamma \vdash S_1 \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C}$$

(C-SS:RECOVERY)

$$\frac{\Gamma \vdash R_0 <: R_1}{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[R_1] \,\rangle \mapsto \langle\, \Gamma \vdash \textbf{none} \Rrightarrow \mathcal{R}[\textbf{none}] \,\rangle}$$

(C-SS:STEP)

$$\frac{\Gamma \vdash R_0 <: R_1}{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[R_1 \Rightarrow R_2; P] \,\rangle \mapsto \langle\, \Gamma \vdash R_2 \Rrightarrow \mathcal{R}[P] \,\rangle}$$

(C-PS:STEP)

$$\frac{\Gamma \vdash R_0 <: R_1 \qquad \Gamma \vdash R_3 <: R_2}{\langle\, \Gamma \vdash (R_0 \Rightarrow R_2; Q) \Rrightarrow \mathcal{R}[R_1 \Rightarrow R_3; P] \,\rangle \mapsto \langle\, \Gamma \vdash Q \Rrightarrow \mathcal{R}[P] \,\rangle}$$

Figure 4.6: Changes to protocol composition to consider subtyping.

discussion on the core protocol composition concerns and move away from the more low-level manipulation of typing concerns.

To simplify the specification of our operational semantics (and the specification of its formal properties later on), we define evaluation contexts ($\mathcal{E}$). Naturally, since the grammar is in let-normal form, these contexts are either the empty context ($\square$) or a let construct.

The types grammar (Figure 4.8) is identical to that of previous chapters, extended to include type bounds on type quantification and **top** for the top type.

4.3.3 Operational Semantics

We use a small step semantics with judgments of the form:

$$H\ ;\ T_0 \ \mapsto\ H_1\ ;\ T_1$$

where a program state, formed by a heap H_0 and a pool of threads T_0, steps to another program state of heap H_1 and thread pool T_1. A heap (H) binds addresses (ρ) to values (v) as before:

$$H \ ::= \ \emptyset \ \text{(empty)} \ \mid \ H,\, \rho \hookrightarrow v \ \text{(binding)}$$

A thread pool (T) is a multiset union of *e*xpressions, each representing a pending computation ("thread"), in arbitrary order. Repeated elements may occur when multiple threads are computing the same expression. We use the notation "$e \cdot T$" for the union of thread pool T and thread e. The semantics are analogous to that of previous chapters, and shown in Figure 4.9. However, note that for instance opening existentials and type application is no longer syntax-driven meaning that there are no reduction rules as those constructs are no longer present in our term grammar. Likewise, we must "wrap" previous reductions (such as memory allocation) with an evaluation context (instead of using only the empty context).

$\rho \in$ Location Constants (Addresses) $\quad \mathtt{t} \in$ Tags $\quad \mathtt{f} \in$ Fields

$x \in$ Variables $\quad l \in$ Location Variables $\quad X \in$ Type Variables

$$p ::= \rho \mid l \qquad u ::= l \mid X \qquad U ::= p \mid A$$

v	::=	ρ	(address)
	|	x	(variable)
	|	$\lambda x.e$	(function)
	|	$\{\overline{\mathtt{f} = v}\}$	(record)
	|	$\mathtt{t}\#v$	(tagged value)
e	::=	v	(value)
	|	$v.\mathtt{f}$	(field)
	|	$v\ v$	(application)
	|	let $x = e$ in e end	(let)
	|	new v	(cell creation)
	|	delete v	(cell deletion)
	|	$!v$	(dereference)
	|	$v := v$	(assign)
	|	case v of $\overline{\mathtt{t}\#x \to e}$ end	(case)
	|	lock $\overline{v}$	(lock locations)
	|	unlock $\overline{v}$	(unlock locations)
	|	fork e	(spawn thread)
$\mathcal{E}$	::=	$\square$ | let $x = \mathcal{E}$ in e	(evaluation contexts)

Figure 4.7: Values (v), expressions (e), and evaluation contexts ($\mathcal{E}$).

A	::=	$!A$	(pure/persistent)
	|	$A \multimap A$	(linear function)
	|	$A :: A$	(stacking)
	|	$A * A$	(separation)
	|	$[\overline{\mathtt{f} : A}]$	(record)
	|	$\forall l.A$	(universal location quantification)
	|	$\exists l.A$	(existential location quantification)
	|	**ref** p	(reference type)
	|	$X[\overline{U}]$	(type variable)
	|	(**rec** $X(\overline{u}).A)[\overline{U}]$	(recursive type)
	|	$\sum_i \mathtt{t}_i \# A_i$	(tagged sum)
	|	$A \oplus A$	(alternative)
	|	$A \,\&\, A$	(intersection)
	|	**rw** $p\ A$	(read-write capability to p)
	|	**none**	(empty resource)
	|	$A \Rightarrow A$	(rely)
	|	$A\ ;\ A$	(guarantee)
	|	**top**	(top)
	|	$\forall X <: A.A$	(bounded universal type quantification)
	|	$\exists X <: A.A$	(bounded existential type quantification)

Notes: $\sum_i \mathtt{t}_i \# A_i$ denotes a single tagged type or a sequence of tagged types separated by + (such as "$\mathtt{a}\#A + \mathtt{b}\#B + \mathtt{c}\#C$"); we simplify $X[]$ to X; we assume $\oplus$, $\&$, $*$, + are commutative and associative; $\overline{Z}$ is a potentially empty sequence of Z elements. Pairs, **rec**ursion, and other constructs are definable in the language [63].

Figure 4.8: Types (A).

$H_0 \,;\, T_0 \;\mapsto\; H_1 \,;\, T_1$ **Dynamics, (D:*)**

(D:New)
$$\frac{\rho \textbf{ fresh}}{H \,;\, \mathcal{E}[\mathsf{new}\ v] \;\mapsto\; H, \rho \hookrightarrow v \,;\, \mathcal{E}[\rho]}$$

(D:Delete)
$$\frac{}{H, \rho^{?} \hookrightarrow v \,;\, \mathcal{E}[\mathsf{delete}\ \rho] \;\mapsto\; H \,;\, \mathcal{E}[v]}$$

(D:Dereference)
$$\frac{}{H, \rho^{?} \hookrightarrow v \,;\, \mathcal{E}[!\rho] \;\mapsto\; H, \rho^{?} \hookrightarrow v \,;\, \mathcal{E}[v]}$$

(D:Assign)
$$\frac{}{H, \rho^{?} \hookrightarrow v_0 \,;\, \mathcal{E}[\rho := v_1] \;\mapsto\; H, \rho^{?} \hookrightarrow v_1 \,;\, \mathcal{E}[v_0]}$$

(D:Application)
$$\frac{}{H \,;\, \mathcal{E}[(\lambda x.e)\ v] \;\mapsto\; H \,;\, \mathcal{E}[e\{v/x\}]}$$

(D:Selection)
$$\frac{}{H \,;\, \mathcal{E}[\{\overline{\mathtt{f} = v}\}.\mathtt{f}_i] \;\mapsto\; H \,;\, \mathcal{E}[v_i]}$$

(D:Case)
$$\frac{}{H \,;\, \mathcal{E}[\mathsf{case}\ \mathtt{t}_i \# v_i\ \mathsf{of}\ \overline{\mathtt{t} \# x \to e}\ \mathsf{end}] \;\mapsto\; H \,;\, \mathcal{E}[e_i\{v_i/x_i\}]}$$

(D:Let)
$$\frac{}{H \,;\, \mathcal{E}[\mathsf{let}\ x = v\ \mathsf{in}\ e\ \mathsf{end}] \;\mapsto\; H \,;\, \mathcal{E}[e\{v/x\}]}$$

(D:Lock)
$$\frac{}{H, \overline{\rho \hookrightarrow v} \,;\, \mathcal{E}[\mathsf{lock}\ \overline{\rho}] \;\mapsto\; H, \overline{\rho^{\bullet} \hookrightarrow v} \,;\, \mathcal{E}[\{\}]}$$

(D:Unlock)
$$\frac{}{H, \overline{\rho^{\bullet} \hookrightarrow v} \,;\, \mathcal{E}[\mathsf{unlock}\ \overline{\rho}] \;\mapsto\; H, \overline{\rho \hookrightarrow v} \,;\, \mathcal{E}[\{\}]}$$

(D:Fork)
$$\frac{}{H \,;\, \mathcal{E}[\mathsf{fork}\ e] \;\mapsto\; H \,;\, \mathcal{E}[\{\}] \cdot e}$$

(D:Thread)
$$\frac{H_0 \,;\, \mathcal{E}[e_0] \;\mapsto\; H_1 \,;\, \mathcal{E}[e_1] \cdot T_1}{H_0 \,;\, \mathcal{E}[e_0] \cdot\ T_0 \;\mapsto\; H_1 \,;\, \mathcal{E}[e_1] \cdot T_1 \cdot\ T_0}$$

$\rho^{?}$ ranges over ρ (not locked) and $\rho^{\bullet}$ (locked) with lock token unchanged.

Figure 4.9: Operational semantics.

We now discuss the new reduction rules. (D:Thread) enables us to frame threads T_0 for a single reduction. Since e_0 may spawn an arbitrary number of threads, the resulting set of threads composes the framed threads (T_0) with those that may have been created (T_1). The remaining rules can be defined as working over a single thread, even if potentially spawning arbitrarily many new threads. (D:Fork) fork enables concurrent executions by spawning a new thread that will execute the expression e while sharing access to the heap, H. We use a mutual exclusion token ($\bullet$) to track whether some thread has acquired the lock for a cell, using (D:Lock). A lock is released with (D:Unlock). We generalize locking to an arbitrary number of locations to enable atomic ("all or nothing") simultaneous locking of all addresses listed by a lock. Thus, a thread may block until all those locks are available. This condition also allows those locations to have been deleted. Attempting to lock deleted state will effectively result in a "deadlock" as the thread will wait forever until the (destroyed) location becomes available. Thus, locking on deleted state is possible but results in a situation that is equivalent to deadlock. (Note that locking on deleted state should not occur in well-typed programs. A thread cannot leave residual non-linear resources, and deleting shared state early must result in an unfulfilled, linear guarantee. However, for proving preservation on expressions, it is convenient to state a weaker statement that allows such a situation to occur.)

4.3.4 Type System

Our typing judgments remains unchanged, using the form:

$$\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1$$

stating that with lexical environment Γ and linear resources Δ_0 we assign the expression e the type A and where the expression produces effects that result in the resources contained in Δ_1.

The typing environments are defined as follows, with changes to account for bounded quantification and no longer including a "defocus-guarantee" as in the previous chapter (since we are no longer concerned with avoiding re-entrant uses of shared state and just let the program deadlock if an erroneous use is done, such as locking the same location twice even if in the same thread):

Γ	::=	$\cdot$	(empty)
	|	$\Gamma,\ x : A$	(variable binding)
	|	$\Gamma,\ p : \mathbf{loc}$	(location assertion)
	|	$\Gamma,\ X <: A$	(bound assertion)
	|	$\Gamma,\ X : k$	(kind assertion)
Δ	::=	$\cdot$	(empty)
	|	$\Delta,\ x : A$	(linear binding)
	|	$\Delta,\ A$	(linear resource)
k	::=	$\mathbf{type} \mid \mathbf{type} \to k \mid \mathbf{loc} \to k$	

Figures 4.10 and 4.11 list the full set of typing rules, to be clearer how the changes to the syntax affect the complete type system. However, the most common rules remain generally analogous to those of previous chapters, although some typing rules were removed to reflect the changes in

$\boxed{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}$ **Typing rules, (T:*)**

$$\frac{}{\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\,\rho \dashv \cdot}\ (\text{T:Ref}) \qquad \frac{\Gamma \mid \cdot \vdash v : A \dashv \cdot}{\Gamma \mid \cdot \vdash v : !A \dashv \cdot}\ (\text{T:Pure}) \qquad \frac{}{\Gamma \mid \cdot \vdash v : ![] \dashv \cdot}\ (\text{T:Unit}) \qquad \frac{\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1}\ (\text{T:Pure-Elim})$$

$$\frac{}{\Gamma, x : A \mid \cdot \vdash x : !A \dashv \cdot}\ (\text{T:Pure-Read}) \qquad \frac{}{\Gamma \mid x : A \vdash x : A \dashv \cdot}\ (\text{T:Linear-Read}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v : [\overline{\mathtt{f} : A}] \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash v.\mathtt{f}_i : A_i \dashv \Delta_1}\ (\text{T:Selection}) \qquad \frac{\overline{\Gamma \mid \Delta \vdash v : A \dashv \cdot}}{\Gamma \mid \Delta \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot}\ (\text{T:Record})$$

$$\frac{\Gamma \mid \Delta_0 \vdash v : \exists l.((!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)) \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v : \exists l.A \dashv \Delta_1}\ (\text{T:Delete}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v : \exists l.((!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)) \dashv \Delta_1}\ (\text{T:New})$$

$$\frac{\Gamma \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1}{\Gamma \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_0}\ (\text{T:Assign})$$

$$\frac{\Gamma \mid \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta \vdash \lambda x.e : A_0 \multimap A_1 \dashv \cdot}\ (\text{T:Function}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2}\ (\text{T:Application})$$

$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash e : A_0 \dashv \Delta_2 \qquad \Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash \Delta_2 <: \Delta_3}{\Gamma \mid \Delta_0 \vdash e : A_1 \dashv \Delta_3}\ (\text{T:Subsumption})$$

$$\frac{\Gamma \mid \Delta \vdash v : A \dashv \cdot}{\Gamma \mid \Delta \vdash \mathtt{t}\#v : \mathtt{t}\#A \dashv \cdot}\ (\text{T:Tag}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v : \sum_i \mathtt{t}_i\#A_i \dashv \Delta_1 \qquad \overline{\Gamma \mid \Delta_1, x_i : A_i \vdash e_i : A \dashv \Delta_2} \qquad i \leq j}{\Gamma \mid \Delta_0 \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_2}\ (\text{T:Case})$$

Figure 4.10: Typing rules (1/2).

(T:DEREFERENCE-LINEAR)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A}{\Gamma \mid \Delta_0 \vdash\ !v : A \dashv \Delta_1, \mathbf{rw}\ p\ ![]}$$

(T:DEREFERENCE-PURE)
$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ !A}{\Gamma \mid \Delta_0 \vdash\ !v : !A \dashv \Delta_1, \mathbf{rw}\ p\ !A}$$

(T:INTERSECTION-RIGHT)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \quad \Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_2}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \& A_2}$$

(T:ALTERNATIVE-LEFT)
$$\frac{\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1 \quad \Gamma \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:FORK)
$$\frac{\Gamma \mid \Delta \vdash e : ![] \dashv \cdot}{\Gamma \mid \Delta \vdash \mathsf{fork}\ e : ![] \dashv \cdot}$$

(T:LET)
$$\frac{\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1, x : A_0 \vdash e_1 : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_2}$$

(T:FRAME)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0, \Delta_2 \vdash e : A \dashv \Delta_1, \Delta_2}$$

(T:CAP-ELIM)
$$\frac{\Gamma \mid \Delta_0, x : A_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : A_0 :: A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:CAP-STACK)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$$

(T:CAP-UNSTACK)
$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}$$

(T:FORALL-LOC-VAL)
$$\frac{\Gamma, l : \mathbf{loc} \mid \Delta_0 \vdash v : A_0 \dashv \cdot}{\Gamma \mid \Delta_0 \vdash v : \forall l.A_0 \dashv \cdot}$$

(T:FORALL-TYPE-VAL)
$$\frac{\Gamma, X : \mathbf{type}, X <: A_1 \mid \Delta_0 \vdash v : A_0 \dashv \cdot}{\Gamma \mid \Delta_0 \vdash v : \forall X <: A_1.A_0 \dashv \cdot}$$

(T:LOCK-RELY)
$$\frac{\overline{\Gamma \mid \cdot \vdash v : \mathbf{ref}\ p \dashv \cdot} \qquad \mathbf{locs}(A_0) = \overline{p}}{\Gamma \mid \Delta,\ A_0 \Rightarrow A_1 \vdash \mathsf{lock}\ \overline{v} : ![] \dashv \Delta, A_0, A_1}$$

(T:UNLOCK-GUARANTEE)
$$\frac{\overline{\Gamma \mid \cdot \vdash v : \mathbf{ref}\ p \dashv \cdot} \qquad \mathbf{locs}(A_0) = \overline{p}}{\Gamma \mid \Delta,\ A_0,\ A_0; A_1 \vdash \mathsf{unlock}\ \overline{v} : ![] \dashv \Delta, A_1}$$

(T:LOCOPENBIND)
$$\frac{\Gamma, l : \mathbf{loc} \mid \Delta_0, x : A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : \exists l.A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:LOCOPENCAP)
$$\frac{\Gamma, l : \mathbf{loc} \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, \exists l.A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:TYPEOPENBIND)
$$\frac{\Gamma, X : \mathbf{type}, X <: A_0 \mid \Delta_0, x : A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : \exists X <: A_0.A_1 \vdash e : A_2 \dashv \Delta_1}$$

(T:TYPEOPENCAP)
$$\frac{\Gamma, X : \mathbf{type}, X <: A_0 \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, \exists X <: A_0.A_1 \vdash e : A_2 \dashv \Delta_1}$$

Notes: bounded variables of a construct and type/location variables of quantifiers must be fresh in the rule's conclusion. **locs** is the non-empty set of locations as discussed in Section A.1.1.

Figure 4.11: Typing rules (2/2).

syntax. We now discuss the main differences, besides pushing quantification to no longer be syntax directed.

(T:FORK) types a fork expression by consuming the linear resources that the fork will use. This is somewhat similar to a closure, but where the result is unit because fork does not wait for the thread to finish its computation. Thus, a fork is executed for the effects it produces on the shared state. As such, and to not leak resources, the final residual resources of the forked expression must be empty as well as its resulting value of type "![]" (pure).

Uses of protocols are linked to the (T:LOCK-RELY) and (T:UNLOCK-GUARANTEE) rules that step a protocol by locking all the locations in the rely type and then checking that the guarantee was obeyed when unlocking, respectively. These two rules replace the previous (T:FOCUS-RELY) and (T:DEFOCUS-GUARANTEE) of chapter 3.

The prior (T:SHARE) is now replaced by a combination of (T:SUBSUMPTION) and (SD:SHARE), a rule for subtyping on environments shown in Figure 4.14, so that sharing is also left non-syntax directed instead of requiring the share construct.

Finally, we have the typing rules that handle quantification: (T:FORALL-*-VAL) and (T:TYPEOPEN*), that are also combined with the subtyping rules (ST:*APP) and (ST:PACK*) to handle quantification in a non-syntax directed way.

Our subtyping rules are listed in Figures 4.12, 4.13, and 4.14 and follow those of previous chapters and kernel $\mathbf{F}_{<:}$ [73] (for decidable bounded quantification). The main novelty is (SD:SHARE) that applies protocol composition to ensure that a resource A_0 can be safely split into resources A_1 and A_2. In prior chapters, this split was syntax-directed but we moved the split to subtyping rules to reduce the syntactical burden and ensure that types do not occur in the term language. Beyond those changes, we have subtyping rules to introduce existential packages, (ST:PACK*), replacing the pack constructs of prior chapters. And subtyping rules to eliminate universal quantification by applying a type/location to a $\forall$, in (ST:*APP). Finally, note the weakening subtyping rule, (ST:WEAKENING), that is necessary later to ensure decidability of the subtyping on types algorithm and close the co-inductive proof—as will be discussed in the next chapter. As before, subtyping transitivity remains admissible (if using the subtyping extension on protocol composition, as otherwise (SD:SHARE) will need subtyping premises on the shared/protocol types).

4.3.5 Technical Results

We now define our main safety theorems, progress and preservation, that are defined over valid program configurations such that:

$$\frac{\Gamma \mid \Delta_i \vdash e_i : ![] \dashv \cdot \qquad i \in \{0, ..., n\} \qquad n \geq 0}{\Gamma \mid \Delta_0,\ ...,\ \Delta_n \vdash (e_0 \ \cdot \ ... \ \cdot \ e_n)} \ (\text{WF:PROGRAM})$$

Stating that a thread pool "$e_0 \ \cdot \ ... \ \cdot \ e_n$" is well formed if each thread can be assigned a "piece" of the linear typing environment (containing resources), and if each individual thread/expression has type "![]" without leaving any residual resources ($\cdot$). Note that the conditions on each thread (e_i) are identical to those imposed by (T:FORK).

$\boxed{\Gamma \vdash A_0 <: A_1}$ **Subtyping on types, (ST:*)**

$$\frac{}{\Gamma \vdash A <: A}\ (\textsc{st:Symmetry}) \qquad \frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash\ !A_0 <: A_1}\ (\textsc{st:ToLinear}) \qquad \frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash\ !A_0 <: !A_1}\ (\textsc{st:Pure}) \qquad \frac{}{\Gamma \vdash\ !A <: ![]}\ (\textsc{st:PureTop}) \qquad \frac{\Gamma_0 \vdash A <: B}{\Gamma_0, \Gamma_1 \vdash A <: B}\ (\textsc{st:Weakening})$$

$$\frac{\Gamma \vdash A_1 <: A_3 \qquad \Gamma \vdash A_2 <: A_0}{\Gamma \vdash A_0 \multimap A_1 <: A_2 \multimap A_3}\ (\textsc{st:Function}) \qquad \frac{\Gamma \vdash A_0 <: A_2}{\Gamma \vdash A_0 <: A_2 \oplus A_1}\ (\textsc{st:Alternative}) \qquad \frac{\Gamma \vdash A_0 <: A_2}{\Gamma \vdash A_0 \& A_1 <: A_2}\ (\textsc{st:Intersection}) \qquad \frac{}{\Gamma \vdash A <: \mathbf{top}}\ (\textsc{st:Top})$$

$$\frac{\overline{\Gamma \vdash A_i <: B_i} \qquad n \leq m}{\Gamma \vdash \sum_i^n \mathtt{t}_i \# A_i <: \sum_i^m \mathtt{t}_i \# B_i}\ (\textsc{st:Sum}) \qquad \frac{X <: A_0 \in \Gamma \qquad \Gamma \vdash A_0 <: A_1}{\Gamma \vdash X <: A_1}\ (\textsc{st:TypeVar})$$

$$\frac{\Gamma \vdash [\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad A_i <: B_i}{\Gamma \vdash [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : B}\ ,\ \mathtt{f}_i : B_i]}\ (\textsc{st:Record}) \qquad \frac{\Gamma \vdash [\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad \overline{\mathtt{f} : A} \neq \emptyset}{\Gamma \vdash [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A_i] <: [\overline{\mathtt{f} : B}]}\ (\textsc{st:Discard})$$

$$\frac{\Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_0 :: A_2 <: A_1 :: A_3}\ (\textsc{st:Stack}) \qquad \frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash \mathbf{rw}\ p\ A_0\ \ <:\ \ \mathbf{rw}\ p\ A_1}\ (\textsc{st:Cap}) \qquad \frac{\Gamma \vdash A_0 <: A_2 \qquad \Gamma \vdash A_1 <: A_3}{\Gamma \vdash A_0 * A_1 <: A_2 * A_3}\ (\textsc{st:Star})$$

Figure 4.12: Subtyping on types (co-inductive) (1/2).

(ST:ALTERNATIVE-CONG)
$$\frac{\Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_0 \oplus A_2 <: A_1 \oplus A_3}$$

(ST:INTERSECTION-CONG)
$$\frac{\Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_0 \& A_2 <: A_1 \& A_3}$$

(ST:LOC-FORALL)
$$\frac{\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1}{\Gamma \vdash \forall l.A_0 <: \forall l.A_1}$$

(ST:PACKLOC)
$$\frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash A_0 <: \exists l.A_1\{l/p\}}$$

(ST:LOC-EXISTS)
$$\frac{\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1}{\Gamma \vdash \exists l.A_0 <: \exists l.A_1}$$

(ST:LOCAPP)
$$\frac{\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1}{\Gamma \vdash \forall l.A_0 <: A_1\{p/l\}}$$

(ST:PACKTYPE)
$$\frac{\Gamma \vdash A_2 <: A_1 \qquad \Gamma \vdash A_0 <: A_0'}{\Gamma \vdash A_0 <: \exists X <: A_1.A_0'\{X/A_2\}}$$

(ST:TYPEAPP)
$$\frac{\Gamma \vdash A_2 <: A_1 \qquad \Gamma, X : \mathbf{type}, X <: A_1 \vdash A_0 <: A_0'}{\Gamma \vdash \forall X <: A_1.A_0 <: A_0'\{A_2/X\}}$$

(ST:TYPE-EXISTS)
$$\frac{\Gamma, X : \mathbf{type}, X <: A_3 \vdash A_0 <: A_1}{\Gamma \vdash \exists X <: A_3.A_0 <: \exists X <: A_3.A_1}$$

(ST:TYPE-FORALL)
$$\frac{\Gamma, X : \mathbf{type}, X <: A_3 \vdash A_0 <: A_1}{\Gamma \vdash \forall X <: A_3.A_0 <: \forall X <: A_3.A_1}$$

Figure 4.13: Subtyping on types (co-inductive) (2/2).

$\boxed{\Delta_0 <: \Delta_1}$ **Subtyping on deltas, (SD:*)**

(SD:VAR)
$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Gamma \vdash \Delta_0, x : A_0 <: \Delta_1, x : A_1}$$

(SD:SYMMETRY)
$$\frac{}{\Gamma \vdash \Delta <: \Delta}$$

(SD:STAR-R)
$$\frac{\Gamma \vdash \Delta, A_0, A_1 <: \Delta, A_1, A_2}{\Gamma \vdash \Delta, A_0, A_1 <: \Delta, A_1 * A_2}$$

(SD:STAR-L)
$$\frac{\Gamma \vdash \Delta, A_0 * A_1 <: \Delta, A_1 * A_2}{\Gamma \vdash \Delta, A_0 * A_1 <: \Delta, A_1, A_2}$$

(SD:TYPE)
$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad \Gamma \vdash A_0 <: A_1}{\Gamma \vdash \Delta_0, A_0 <: \Delta_1, A_1}$$

(SD:NONE-R)
$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1}{\Gamma \vdash \Delta_0 <: \Delta_1, \mathbf{none}}$$

(SD:NONE-L)
$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1}{\Gamma \vdash \Delta_0, \mathbf{none} <: \Delta_1}$$

(SD:ALTERNATIVE-L)
$$\frac{\Gamma \vdash \Delta_0, A_0 <: \Delta_1 \qquad \Gamma \vdash \Delta_0, A_1 <: \Delta_1}{\Gamma \vdash \Delta_0, A_0 \oplus A_1 <: \Delta_1}$$

(SD:INTERSECTION-R)
$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1, A_1 \qquad \Gamma \vdash \Delta_0 <: \Delta_1, A_2}{\Gamma \vdash \Delta_0 <: \Delta_1, A_1 \& A_2}$$

(SD:SHARE)
$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1, A_0 \qquad \Gamma \vdash A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \vdash \Delta_0 <: \Delta_1, A_1, A_2}$$

Figure 4.14: Subtyping on linear typing environments.

For convenience and clarity, both safety theorems are supported by auxiliary theorems over a single expression, besides the main theorem over the complete thread pool. We now state progress over programs:

Theorem 5 (Progress - Program)**.** *If* $\Gamma \mid \Delta \vdash T_0$ *and* $\texttt{live}(T_0)$ *and if exists* H_0 *such that* $\Gamma \mid \Delta \vdash H_0$ *then* $H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$.

Where $\texttt{live}(T)$ means that the thread pool T contains at least one "live" thread such that the thread is neither a value (v) nor is waiting for a lock to be released (note that we include deadlocked configurations in this waiting set). "$\Gamma \mid \Delta \vdash H$" ensures that the heap is well-defined according to Γ and Δ. Changes to store typing are discussed further below.

We define $\texttt{Wait}(H, e)$ over a thread (e) and heap (H) such that the current evaluation context for e is reduced to evaluating the configuration:

$$H \; ; \; \mathcal{E}[\mathsf{lock}\, \rho, \overline{\rho'}] \; \cdot \; T \qquad \text{where} \qquad (\rho \hookrightarrow v) \notin H$$

which contains at least one location (ρ) that is either currently locked ($\rho^\bullet$) or was deleted and, therefore, the thread must block waiting (potentially indefinitely) for that lock to be released before continuing. Progress on expressions makes these program states explicit:

Theorem 6 (Progress - Expressions)**.** *If* $\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$ *then we have that either:*

- e_0 *is a value, or;*
- *if exists* H_0 *and* Δ *such that* $\Gamma \mid \Delta, \Delta_0 \vdash H_0$ *then either:*
  - *(steps)* $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$*, or;*
  - *(waits)* $\texttt{Wait}(H_0, e_0)$.

Note that if some locked shared state is deleted then a thread must leave a residual type, the guarantee that was not fulfilled. This use of the shared state would not type according to our (WF:PROGRAM) condition, since that residual guarantee type is linear. However, we must still account for that situations to be able to prove the theorem on preservation over expressions since its statement is weaker than progress over programs. (A similar situation occurs for the preservation theorems.)

Preservation ensures that a reduction step will preserve both the type and the effects of the expression that is being reduced (so that each thread's type, "![]", and effect, "·", remains unchanged). As above, we use a preservation theorem over programs that makes use of an auxiliary theorem on preservation over expressions:

Theorem 7 (Preservation - Programs)**.** *If we have* $\Gamma_0 \mid \Delta_0 \vdash H_0$ *and* $\Gamma_0 \mid \Delta_0 \vdash T_0$ *and* $H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$ *then, for some* Δ_1 *and* Γ_1*, we have:* $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash H_1$ *and* $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash T_1$.

So that a well-formed set of threads (T_0) remains well-formed after stepping one of these threads (resulting in T_1). Preservation over a single expression must still account for the resources (Δ_T) that may be consumed by a newly spawned thread (T):

Theorem 8 (Preservation - Expressions)**.** *If we have* $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$ *and* $\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0$ *and* $\Gamma_0 \mid \Delta_0 \vdash e_0 : A \dashv \Delta$ *then, we have that* $\Delta_0 = \Delta', \Delta_T$ *such that for some* Δ_1 *and* Γ_1*, we have:* $\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ *and* $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta$ *and* $\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$.

We complement our main results with the following "Error Freedom" corollary that shows that our system cannot type programs that allow data races and the dereference of destroyed memory

cells, i.e. that our system ensures memory safety and race freedom. We also include two properties related to the absence of erroneous uses of records and tagged sums, that the core system already ensured but that are only stated explicitly in here.

Corollary 1 (Error Freedom)**.** *The following program states cannot be typed:*

1. *(Data Race) When two threads read/modify the same location:*

$$H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[!\rho] \cdot T \qquad H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[\rho := v'] \cdot T$$

   (technically, we also ensure read-exclusive accesses)

2. *(Memory Fault) Accessing a non-existing/deleted location:*

$$H; \mathcal{E}[\rho := v] \cdot T \qquad H; \mathcal{E}[!\rho] \cdot T \qquad (\text{where } \rho \notin H)$$

3. *(Ownership Fault) Attempt to delete a non-existing location:*

$$H; \mathcal{E}[\mathsf{delete}\,\rho] \cdot T \qquad (\text{where } \rho \notin H)$$

4. *(Record Access Fault) Accessing a non-existing field:*

$$H; \mathcal{E}[\{\,\overline{\texttt{f} = v}\,\}.\texttt{f'}] \cdot T \qquad (\text{where } \texttt{f'} \notin \overline{\texttt{f}})$$

5. *(Unexpected Tag) Branches in case do not consider the provided value's tag:*

$$H; \mathcal{E}[\mathsf{case}\ \texttt{t'\#}v\ \mathsf{of}\ \overline{\texttt{t\#}x \to \ldots}\ \mathsf{end}] \cdot T \qquad (\text{where } \texttt{t'} \notin \overline{\texttt{t}})$$

The proof is straightforward due to our use of locks to ensure mutual exclusion and the fact that our protocols discipline the use of shared state—while the last two errors were already avoided by the initial core language of chapter 2. Thus, the errors above are avoided by protocol safe protocol composition and by the resource tracking of the core linear system.

Protocol composition obeys the same properties as in the previous chapter:

Lemma 7 (Composition Progress)**.** *If* $\Gamma \vdash S \Rrightarrow P \parallel Q$ *then* $\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto C$.

Lemma 8 (Composition Preservation)**.** *If* $\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash S' \Rrightarrow P' \parallel Q' \,\rangle \cdot C$ *and* $\Gamma \vdash S \Rrightarrow P \parallel Q$ *then* $\Gamma' \vdash S' \Rrightarrow P' \parallel Q'$.

And:

Lemma 9. $\Gamma \vdash S \Rrightarrow S \parallel \mathbf{none}$.

Lemma 10. *If* $\Gamma \vdash S \Rrightarrow P_0 \parallel P_1$ *then* $\Gamma \vdash S \Rrightarrow P_1 \parallel P_0$.

Lemma 11. *If we have* $\Gamma \vdash S \Rrightarrow P_0 \parallel P$ *and* $\Gamma \vdash P \Rrightarrow P_1 \parallel P_2$ *then exists* W *such that* $\Gamma \vdash S \Rrightarrow W \parallel P_2$ *and* $\Gamma \vdash W \Rrightarrow P_0 \parallel P_1$. *(i.e. if* $\Gamma \vdash S \Rrightarrow P_0 \parallel (P_1 \parallel P_2)$ *then* $\Gamma \vdash S \Rrightarrow (P_0 \parallel P_1) \parallel P_2$.*)*

Changes to store typing are shown in Figure 4.15. Store typing now includes (STR:SUBSUMPTION) to enable splitting resources and other uses of subtyping over store typed contexts. The main addition is (STR:LOCKED) and (STR:DEAD-LOCKED). Both rules state that a pending guarantee must be supported by a heap "$\overline{\rho \hookrightarrow v'}$" that is compatible with the expected guarantee type "A_1". Furthermore, the next step (A_2) of that protocol will enable the store typing of the rest of Δ. (Recall that due to our composition preservation theorem, we know that stepping a single protocol will still enable the full set of protocols of that state to safely compose.) The difference between (STR:LOCKED) and (STR:DEAD-LOCKED) resides only on whether the locations mentioned in the protocol are in the heap or not. If those locations still exist then the locations must be locked to rule out access to private state.

$\boxed{\Gamma \mid \Delta \vdash H}$ **Store typing, (STR:*)**

$$\frac{}{\cdot \mid \cdot \vdash \cdot}\ (\textsc{str:Empty}) \qquad \frac{\Gamma \mid \Delta \vdash H}{\Gamma, \rho : \mathbf{loc} \mid \Delta \vdash H}\ (\textsc{str:Loc}) \qquad \frac{\Gamma \mid \Delta_0 \vdash H \qquad \Gamma \vdash \Delta_0 <: \Delta_1}{\Gamma \mid \Delta_1 \vdash H}\ (\textsc{str:Subsumption})$$

$$\frac{\Gamma \mid \Delta, \Delta_v \vdash H \qquad \Gamma \mid \Delta_v \vdash v : A \dashv \cdot}{\Gamma \mid \Delta, \mathbf{rw}\, \rho\, A \vdash H, \rho \hookrightarrow v}\ (\textsc{str:Binding})$$

$$\frac{\mathbf{locs}(A_0) = \mathbf{locs}(A_1) = \overline{\rho} \qquad \Gamma \mid A_0 \vdash H', \overline{\rho \hookrightarrow v} \qquad \Gamma \mid A_1 \vdash H'', \overline{\rho \hookrightarrow v'} \qquad \Gamma \mid \Delta, A_2 \vdash H, H'', \overline{\rho \hookrightarrow v'}}{\Gamma \mid \Delta, A_0, (A_1; A_2) \vdash H, H', \overline{\rho^\bullet \hookrightarrow v}}\ (\textsc{str:Locked})$$

$$\frac{\mathbf{locs}(A_1) = \overline{\rho} \qquad \overline{\rho \hookrightarrow v} \notin H \qquad \Gamma \mid A_1 \vdash H'', \overline{\rho \hookrightarrow v'} \qquad \Gamma \mid \Delta, A_2 \vdash H, H'', \overline{\rho \hookrightarrow v'}}{\Gamma \mid \Delta, (A_1; A_2) \vdash H}\ (\textsc{str:Dead-Locked})$$

$$\frac{\Gamma, l : \mathbf{loc} \mid \Delta, A \vdash H}{\Gamma \mid \Delta, \forall l.A \vdash H}\ (\textsc{str:ForallLocs}) \qquad \frac{\Gamma, X : \mathbf{type}, X <: A_0 \mid \Delta, A_1 \vdash H}{\Gamma \mid \Delta, \forall X <: A_0.A_1 \vdash H}\ (\textsc{str:ForallTypes})$$

(where l, X are fresh in the conclusion)

Figure 4.15: Store Typing rules.

4.4 Additional Examples

We now discuss some additional examples. The last example revisits the pipe example of chapter 3 in order to illustrate the syntactical simplifications that occurred when we made some aspects of the language to no longer be syntax-directed. However, we include an explicit share construct for readability and clarity of which split we expect to occur at that point in the program—although the discussed language handles that split implicitly.

4.4.1 MVars

Our protocols can be used to model `MVars` [72] as employed in Haskell[2]. These structures are either empty or contain a value of some type. Naturally, since `MVars` can also be seen as modeling channels or a binary semaphore they can also be modeled in our system. We consider four operations: `newMVar` that creates a new `MVar`, `splitMVar` that splits an existing `MVar` sharing it further, `putMVar` which will either place a value into the shared state or wait until the shared state is again empty before inserting the designated value, and finally `takeMVar` which removes the value from the shared state (leaving it empty) and which may potentially need to wait for the state to become non-empty.

One possible way to model `MVars` is with the following protocol:

$$\begin{aligned}\texttt{MVar}[l] \triangleq \exists Y.(\ (\ \mathbf{rw}\ l\ \texttt{Empty\#}Y\ \Rightarrow\ \mathbf{rw}\ l\ \texttt{Empty\#}Y\ ;\ \texttt{MVar}[l]\ )\ \&\\ (\ \mathbf{rw}\ l\ \texttt{Empty\#}Y\ \Rightarrow\ \mathbf{rw}\ l\ \texttt{Full\#}\mathbf{int}\ ;\ \texttt{MVar}[l]\ )\ )\\ \oplus(\ (\ \mathbf{rw}\ l\ \texttt{Full\#}\mathbf{int}\ \Rightarrow\ \mathbf{rw}\ l\ \texttt{Empty\#}[]\ ;\ \texttt{MVar}[l]\ )\ \&\\ \exists Y.(\ \mathbf{rw}\ l\ \texttt{Full\#}Y \Rightarrow \mathbf{rw}\ l\ \texttt{Full\#}Y\ ;\ \texttt{MVar}[l]\ )\ )\end{aligned}$$

The protocol can be implemented in the following structure that creates a cell (at location `@m`) that will be used to model the `MVar`.

```
let newMVar = λ_.
  let m = new Empty#{} in
    share (rw @m Empty#![]) as MVar[@m] || none;
    {
    splitMVar = λ _. share (rw @m Empty#![]) as MVar[@m] || MVar[@m],

    putMVar = λ val .
      rec R.
        lock m;
        case !m of
          Empty#x →
            m := Full#val;
            unlock m
        | Full#value →
            m := Faull#value;
            unlock m;
            R // retries
        end
```

[2] http://hackage.haskell.org/package/base-4.7.0.2/docs/Control-Concurrent-MVar.html

```
    end,

  takeMVar = λ _ .
    rec R.
      lock m;
      case !m of
        Empty#x →
          m := Empty#x;
          unlock m;
          R // retries
      | Full#value →
          m := Empty#{};
          unlock m;
          value
      end
    end
}
end
end
```

The newMVar function can be assigned the following type, that abstracts the underlying location that was created by the function call (again omitting !'s for conciseness):

$$[] \multimap \exists t.([\ \texttt{splitMVar} : [] :: \texttt{MVar}[t] \multimap [] :: (\texttt{MVar}[t] * \texttt{MVar}[t]),\ \texttt{putMVar} : \textbf{int} :: \texttt{MVar}[t] \multimap [] :: \texttt{MVar}[t],\ \texttt{takeMVar} : [] :: \texttt{MVar}[t] \multimap \textbf{int} :: \texttt{MVar}[t]\ ] :: \texttt{MVar}[t])$$

Indeed, we may also assign a type that fully abstracts the type structure that is used for coordination by instead assigning the function the following type:

$$[] \multimap \exists M.([\ \texttt{splitMVar} : [] :: M \multimap [] :: (M * M),\ \texttt{putMVar} : \textbf{int} :: M \multimap [] :: M,\ \texttt{takeMVar} : [] :: M \multimap \textbf{int} :: M\ ] :: M)$$

4.4.2 Shared Pair

This example shows the case where neither alias sees the "full picture" of the contents of the shared state. Thus, each alias can have a different perspective on the contents of the shared state such that each alias abstracts different components of the pair.

Consider the following cell:

$$\textbf{rw}\ p\ [A, B]$$

Such a cell contains a pair type with the first component of type A and the second component of type B. The idea is to now share that same cell, including its contents, through two aliases. Each alias will not modify the other alias's pair component, although the state of the cell and of the alias' pair component may be changed. Due to the typing constraints enforced by a rely-guarantee protocol, each individual alias' actions are guaranteed to preserve the data of the other alias.

The crucial bit of this example is how protocol composition proceeds. Since part of the pair's type is abstracted, an alias will never see the complete type of the pair. However, since the other

alias will be allowed to change its component, safe composition will ensure that this interference is accounted for although it is abstracted. We write $\forall X$ to abbreviate "$\forall X <:$ **top**" and $\exists X$ for "$\exists X <:$ **top**":

$$\begin{array}{lcl} \mathtt{P}[A][B] & \triangleq & \mathbf{rw}\ p\ [A, B] \\ \mathtt{L}[A] & \triangleq & \exists X.(\ (\mathtt{P}[A][X])\ \Rightarrow\ \forall Y.(\ (\mathtt{P}[Y][X])\ ;\ (\mathtt{L}[Y])\ )\ ) \\ \mathtt{R}[A] & \triangleq & \exists X.(\ (\mathtt{P}[X][A])\ \Rightarrow\ \forall Y.(\ (\mathtt{P}[X][Y])\ ;\ (\mathtt{R}[Y])\ )\ ) \end{array}$$

$$\cdot\ \vdash\ (\mathtt{P}[X][Y])\ \Rrightarrow\ \mathtt{L}[X]\ \|\ \mathtt{R}[Y]$$

Note that the existential cannot last longer than the interval where no interference from the other alias may "appear" in the pair. Thus, on repeating the protocol, we must consider a fresh variable X to model the new type that may appear in the cell. After splitting, each alias will only precisely know the type of its own component. For the remaining component of the pair, the protocol will enforce that the alias must preserve the data stored there—although that alias has no precise type information on what its type may be. Therefore, this will allow a form of local hiding that yet is globally consistent.

Protocol composition results in the following set of configurations:

$$\begin{array}{lllllllll} \{\ \mathbf{1} & \langle\ \cdot & \vdash & \mathtt{P}[A][B] & \Rrightarrow & \mathtt{L}[A] & \| & \mathtt{R}[B] & \rangle\ , \\ \mathbf{2} & \langle\ X : \mathbf{type} & \vdash & \mathtt{P}[A][X] & \Rrightarrow & \mathtt{L}[A] & \| & \mathtt{R}[X] & \rangle\ , \\ \mathbf{3} & \langle\ Y : \mathbf{type} & \vdash & \mathtt{P}[Y][B] & \Rrightarrow & \mathtt{L}[Y] & \| & \mathtt{R}[B] & \rangle\ , \\ \mathbf{4} & \langle\ X : \mathbf{type}\ ,\ Y : \mathbf{type} & \vdash & \mathtt{P}[X][Y] & \Rrightarrow & \mathtt{L}[X] & \| & \mathtt{R}[Y] & \rangle\ \} \end{array}$$

4.4.3 Pipe Example, Revisited

We now revisit the pipe example of chapter 3. Recall that two aliases interact through shared state so that one alias waits for the other alias to insert an element into the shared cell. Due to the changes to support thread-based concurrency, the protocols are not exactly identical to those shown in chapter 3. Namely, our protocol types need to ensure that a lock-token invariant is preserved, which changes how protocols express ownership recovery. We use the following abbreviations for the states of the protocols:

$$\begin{array}{rcl} \mathsf{Node}[p] & \triangleq & \exists q.(\ (\mathbf{rw}\ p\ \texttt{!Node\#![element}\ :\ \mathbf{int}, \texttt{next}\ :\ (!\mathbf{ref}\ q)\texttt{]})\ \ *\ \ \mathsf{H}[q]\ ) \\ \mathsf{Close}[p] & \triangleq & \mathbf{rw}\ p\ \texttt{!Close\#![]} \\ \mathsf{Empty}[p] & \triangleq & \mathbf{rw}\ p\ \texttt{!Empty\#![]} \end{array}$$

Note that in Node, we can avoid nesting the protocol to the next node inside the capability to location p (as was done in chapter 3). Instead, we see that $\mathsf{H}[q]$ is at "the same level" as that capability and combined together using $*$. This allows using the capability to p independently of H, and since p's content is pure it also avoids the need to re-assign linear values to p, as happened in section 3.3.5.

Next, we redefine the protocols for the head and tail aliases:

$$\begin{array}{l} \mathsf{T}[p]\ \triangleq\ \mathsf{Empty}[p] \Rightarrow (\ \mathsf{Close}[p]\ \oplus\ \mathsf{Node}[p]\ )\ ;\ \mathbf{none} \\ \mathsf{H}[p]\ \triangleq\ (\ \mathsf{Node}[p]\ \Rightarrow\ \mathsf{Node}[p]\ ;\ \mathsf{Node}[p]\ )\ \oplus\ (\ \mathsf{Close}[p]\ \Rightarrow\ \mathsf{Close}[p]\ ;\ \mathsf{Close}[p]\ ) \\ \qquad\qquad \oplus\ (\ \mathsf{Empty}[p] \Rightarrow \mathsf{Empty}[p]\ ;\ \mathsf{H}[p]\ ) \end{array}$$

As in other examples, although the shared cell can be freely used in the `Empty` case those changes are only allowed privately. This means that the contents of that cell must forcefully be returned to its original state when the shared cell is unlocked.

Protocol composition proceeds similarly to before by checking that the following split of location p shares the state safely:

$$\cdot \vdash \mathsf{Empty}[p] \;\Rrightarrow\; \mathsf{T}[p] \parallel \mathsf{H}[p]$$

And yields the following set of configurations:

$$\begin{array}{llrcll}
\{ & \langle \cdot \vdash & \mathsf{Empty}[p] & \Rrightarrow & \mathsf{T}[p] & \parallel \mathsf{H}[p] \rangle, \\
& \langle \cdot \vdash & \mathsf{Close}[p] \oplus \mathsf{Node}[p] & \Rrightarrow & \textbf{none} & \parallel \mathsf{H}[p] \rangle, \\
& \langle \cdot \vdash & \mathsf{Close}[p] & \Rrightarrow & \textbf{none} & \parallel \mathsf{Close}[p] \rangle, \\
& \langle \cdot \vdash & \mathsf{Node}[p] & \Rrightarrow & \textbf{none} & \parallel \mathsf{Node}[p] \rangle, \\
& \langle \cdot \vdash & \textbf{none} & \Rrightarrow & \textbf{none} & \parallel \textbf{none} \rangle \; \}
\end{array}$$

Implementation Finally, we show a possible implementation of the pipe within the language defined in this chapter. Since existential types are automatically opened, we use the `@` prefix on a variable to denote its location variable that was automatically opened by the type system. Therefore, if `t` is a "**ref** a" then `@t` denotes location a. Note that due to redefinition of Node, we no longer need to re-assign its contents on inspection, as we are just reading a pure value on `tryTake` in the `Empty` case.

```
1  let newPipe = λ(_).
2    let node = new Empty#{} in
3      // we use explicit sharing annotation to make it clear how state is split
4      share (rw @node !Empty#![]) as H[@node] || T[@node];
5      let h = new node in
6       let t = new node in
7       {
8       put = λ e.let last = new Empty#{} in
9                   let old = !t in
10                    lock old;
11                    share (rw @last !Empty#![]) as H[@last] || T[@last];
12                    old := Node#{ element = e , next = last };
13                    unlock old;
14                    t := last;
15                  end
16                end,
17
18      close = λ _.let last = !t in
19                    delete t;
20                    lock last;
21                    last := Closed#{};
22                    unlock last
23                  end,
24
25      tryTake = λ _.let first = = !h in
```

```
                    lock first;
                    case !first of
                      Empty#_ →
                        unlock first;
                        NoResult#{}
                    | Closed#_ →
                        unlock first;
                        delete first;
                        delete h;
                        Depleted#{}
                    | Node#n →
                        h := n.next;
                        unlock first;
                        delete first;
                        Result#n.element
                    end
                  end
  }
  end
  end
end
in
// ...
```

The types remains identical to the previous chapter, but using the abbreviations given above.

4.5 Encoding Typeful Message-Passing Concurrency

We now discuss how our protocols can be used to encode message-passing style of concurrency, via shared memory cells. However, our encoding only works in a non-distributed setting since our core language does not account for actual network interactions—although that can also be seen as interacting with the network card's buffer. Interestingly, this enables values to be shared between threads without copying. For instance, an auxiliary cell can be used to store values so that only the reference to that location (and its capability, a type) will need to move between threads via the channel. Thus, the need to copy arbitrary sized values is eliminated.

Due to the underlying protocol types and its shared memory underpinnings, our encoding can send "messages to self" and is naturally asynchronous. This flexibility also allows for non-deterministic interactions, when different alternatives may be picked depending on a particular thread scheduling. Technically, instead of sending or receiving a message, the interaction occurs through relying and guaranteeing that the shared cell contains values of a certain type. The novelty is that our protocols can encode both shared-memory and message-passing styles of interactions in a single, unified protocol framework.

4.5.1 Brief Overview

Section 4.5.3 includes the complete "`Buyer-Seller-Shipper`" example (the canonical and simple example frequently used in session-based concurrency works) while in here we only take a closer look at the main aspects of the `Buyer`'s interaction to illustrate the core principle of our encoding.

We model a *c*hannel using a capability to location c. For conciseness, we omit "**rw** c" from "**rw** c A" since all changes occur over the same location. The `Buyer`'s type uses standard π-calculus [66] notations where ! means sending (and ? means receiving) a value. These actions are directly mapped to the rely type (for receiving) and the guarantee type (for sending) of a protocol.

$$\underbrace{\texttt{buy!}(prod)}_{\texttt{idle0\#[]} \Rightarrow \texttt{buy\#}prod} \;;\; \underbrace{\texttt{price?}(p)}_{\texttt{price\#}p \Rightarrow \texttt{idle2\#[]}} \;;\; \underbrace{\texttt{details?}(d)}_{\texttt{details\#}d \Rightarrow \texttt{[]}}$$

`Buyer` starts by sending a request to `buy` some *prod*uct, then waits for the *p*rice, and finally receives the *d*etails of that product. Under that interaction protocol, we simply map sends to a guarantee type of a step, and receives the a rely type of a step.

Our protocol interactions are both non-deterministic and may contain an arbitrary number of simultaneous participants. To ensure that the desired participant (`Buyer`) is the only one allowed to received (take) the price, we must mark the contents with a specific tag so that only `Buyer` has permission to change that state. To handle the non-deterministic interleaving of protocols, we must introduce explicit "wait states" that allow a participant to check if the communication has reached the desired point to that participant or if it should continue waiting. We abstract these steps as WaitSteps (although we properly define them further below) as they essentially recur on that same step of the protocol (i.e. "busy-wait").

$$\begin{array}{l}(\ \texttt{idle0\#[]} \Rightarrow \texttt{buy\#}prod\ )\ ; \\ \quad (\ \textsf{WaitSteps} \oplus (\ \texttt{price\#}p \Rightarrow \texttt{idle2\#[]}\ )\ )\ ; \\ \quad\quad (\ \textsf{WaitSteps} \oplus (\ \texttt{details\#}d \Rightarrow \texttt{[]}\ )\ )\end{array}$$

The richness of our shared state interactions means that we can immediately support fairly complex session-based mechanisms (such as delegation, asynchronous communication, "messages to self", multiparty interactions, internal/external choices, etc.) within our small protocol framework. However, this additional flexibility comes at the cost of requiring a more complex protocol composition mechanism. Our protocol composition accounts for both the non-deterministic nature of protocol interleaving and the arbitrary number of simultaneous participants, features which are usually absent from strictly choreographed session-based concurrency (which greatly simplifies splitting).

Naturally, more complex examples are possible. In here our focus is on showing the core insights that enable us to relate these two techniques: 1) mapping send/receive to our rely/guarantee types; 2) adding explicit waiting states to account for non-deterministic protocol interleaving; and 3) tag the content of a cell in order to ensure that only the right participant will be able to mutate the state at that point in the interaction (even if others must be able to non-destructively inspect it).

4.5.2 Encoding `send` and `receive`

We now discuss the high-level concepts of our encoding of message-passing:

Channels as memory locations. To create a new channel we must create a new cell that will model the interaction that occurs through that channel. Similarly, closing a channel is equivalent of deleting that memory location, which effectively prohibits further uses of the cell/channel. Since the communication occurs through the shared location, the channel name is meaningless as long as the specific location is shared by the two endpoints, even if using different local names for those channels / location variables.

Sending as (little more than) writing to shared state. Our encoded `send` is non-blocking which means that the thread does not need to wait for the other party to `receive` the value. This also means that a thread can `send` a message to itself, if the programmer so wishes. To avoid overwriting non-received values, the exact state of the channel/cell must be marked with a specific tag. This is akin to sending or waiting for an acknowledge that the value has been properly received, instead of potentially flooding the receiver's buffer. The extra complexity can be hidden from the programmer by using the idioms that we discuss below.

Receiving as (little more than) reading from shared state. We can encode `receive` in similar ways to sending, so that we may need to mark the shared state with a specific tag so that the receiver knows that the cell was read (and thus that the cell is available to write to).

Multiparty. Our channels are shared cells where the kind of communication that can be done through those shared cells is only constrained by the protocol type. Thus, whenever the protocols compose safely, that interaction is ruled valid. As we saw in previous examples, protocol composition allows arbitrary aliasing and therefore multiparty communication (including delegation) is naturally supported by our scheme.

We now discuss the encoding of send/receive shown in Figure 4.16. The underlying principle of these idioms is to hide the waiting states of the channel. Client code will then look identical to what you would normally see in traditional message-passing systems. Therefore, a send/receive will potentially have to wait if the cell is not available with the desired tag that will enable "sending" or if it has no new value to "receive". Creating a new channel (`new`) and closing a channel (`close`) are straightforward uses of memory allocation (followed by an explicit share construct to split the state into the desired protocols for that channel) and memory deletion, respectively. Therefore, we will only look into more detail on the encoding of `receive` and `send`.

The crucial aspect is that the protocol must model the buffer's changing state as the communication progresses, enabling the encoding to seamlessly wait for a particular phase of the interaction.

Consider the following code:

```
1 let x = receive(c) in ...
```

To receive a value sent to channel c, we need to wait for a specific state to be stored in that cell. This waiting is based on the specific protocol type of the channel. This means that c should have a type of the kind:

$$\mathsf{RetryOn}[A, B, ...] \oplus \ (\ \mathbf{rw}\ \mathtt{c}\ \mathtt{ReadyToReceive\#}V \Rightarrow \mathbf{rw}\ \mathtt{c}\ \mathtt{idle\#}[]\ ;\ \ldots\ )$$

```
receive(c) ≜
  rec X. // recursion point
    lock c; // locks location of 'c'
    case !c of
      // 1. waiting states, just unlocks and retries:
      A#n → ... // analogous to case below
    | B#n →
        c := B#n; // restore any linear type
        unlock c; // unlock cell
        X // retry
    // 2. desired (receive) state:
    | ReadyToReceive#v →
        c := idle#{}; // marks as received
        unlock c; // unlock cell
        v // result of receiving from channel 'c'
    end // end case
  end // end recursion
  // at this point c has type P
```

```
send(c,v) ≜
  rec X. // recursion point
    lock c;
    case !c of
      // 1. waiting states, just unlocks and retries.
      A#n → ... // analogous to case below
    | B#n →
        c := B#n; // restore any linear type
        unlock c; // unlock cell
        X // retry
    // 2. desired (receive) state:
    | idle#_ →
        c := ReadyToReceive#v; // signal sent
        unlock c; // unlock cell
        {} // result of send is unit
    end // end case
  end // end recursion
  // at this point c has type P
```

Figure 4.16: Possible encoding of `send` and `receive` functions.

where RetryOn are just ("busy-wait") cycles in the protocol that retry that same step of the protocol. The alternative step on the right advances the protocol when the right value, tagged with `ReadyToReceive`, is found in c.

Sending is analogous since it must also wait for the "channel" to be free (for instance by being tagged with `idle`) which leads to the recursion that spins waiting for the appropriate tag to be present. Similarly to receive, we have:

$$\mathsf{RetryOn}[A, B, ...] \oplus \ (\ \mathbf{rw}\ \mathtt{c}\ \mathtt{idle\#}[] \Rightarrow \mathbf{rw}\ \mathtt{c}\ \mathtt{ReadyToReceive\#}V\ ;\ \ldots\ )$$

Open Problems Technically, our messaging mechanism is a "queue" of a single element. The communication can occur asynchronously and without any guarantee of global progress, making the communication vulnerable to live-locks or even deadlocks if the use of this encoding is mixed with locks. While ideas from prior works (such as [47]) could perhaps be adapted/extended to ensure deadlock freedom, the liveness problem appears to be rooted in the operational semantics we use. We lack the kind of "lockstep" scheduling that occurs in strictly choreographed systems. For instance, in traditional π-calculus, a send and a receive to/from a channel will synchronize the two processes. In our system, the scheduler does not guarantee this strong synchronization which enables a single thread to "monopolize" the system. Thus, the scheduler does not guarantee fairness in the use of the computing resources, which may block global progress. Consequently, providing some form of liveness guarantee will likely require changing the operational semantics to provide stronger scheduling properties.

In our system, to ensure safe and local re-splits, we must explicitly list all the waiting phases of the communication. While in traditional message-passing system such waiting phases are hidden from the programmer, our receive and send may cause the current thread to "busy-wait" when an

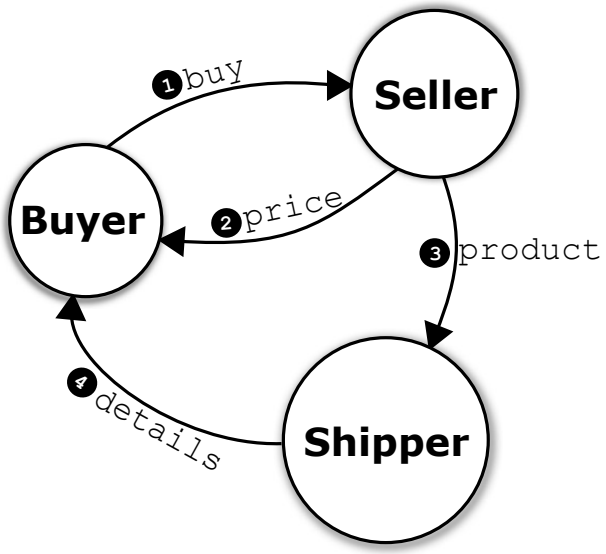

Figure 4.17: Buyer-Seller-Shipper message-passing example.

old value is still in the shared cell. (Although one could write different code to allow overwrites at the cost of non-determinism.) This waiting is akin to blocking the thread when there is no actual, higher-level, mechanism to wait on some event. Technically the implementation could employ a simple optimization that registers which thread should wake up on a particular state change of the cell/channel, but we do not consider the problem of finding better ways to schedule threads to reduce excessive spinning.

4.5.3 Buyer-Seller-Shipper Example

The example of Figure 4.17 (adapted from [25]) shows a multiparty communication between a *Buyer*, a *Seller*, and a *Shipper*. The buyer sends a request to buy some product to the seller, which then replies with the price and delegates shipping to the shipper. The shipper, upon receiving the request for some product, then replies by sending the shipping details back to the buyer.

We follow the original example in using "session-delegation" instead of opening a private communication channel to talk to the shipper, although our system can also model that. Similarly, to simplify the presentation, we do not abstract a channel's internal states even though such an abstraction would produce a more modular protocol specification.

We begin by modeling the communication by explicitly labeling the messages sent through the channel using the π-calculus notation of "!" for sending and "?" for receiving, and extending the notation to use "$\triangleleft$" and "$\triangleright$" to connect or delegate the communication.

$$\begin{array}{rcl} \text{Buyer} & : & \text{Seller} \triangleleft \texttt{buy}!(prod)\ ;\ \texttt{price}?(p)\ ;\ \texttt{details}?(d) \\ \text{Seller} & \triangleright & \texttt{buy}?(prod)\ ;\ \texttt{price}!(p)\ ;\ \text{Shipper} \triangleleft \texttt{product}!(prod) \\ \text{Shipper} & \triangleright & \texttt{product}?(prod)\ ;\ \texttt{details}!(d) \end{array}$$

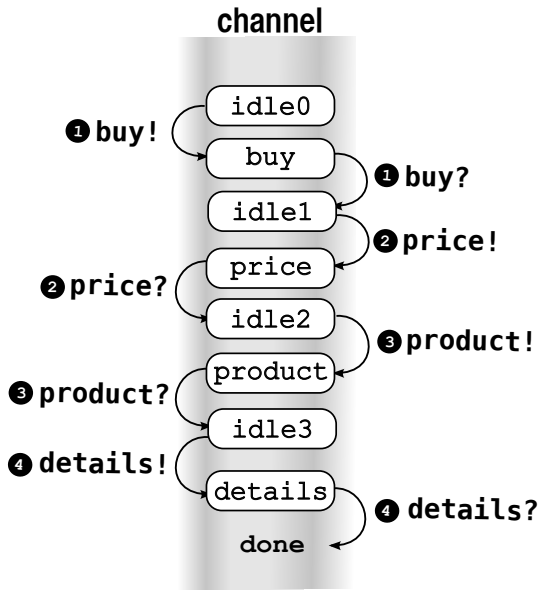

Figure 4.18: Buyer-Seller-Shipper shared channel's changing session state.

The types above define a Buyer type that initializes a communication (◁) with the Seller server. From the perspective of the Buyer, the buyer sends a buy message (with the desired *prod*uct) and waits for two replies in that channel: one with the `price` and another with the `details` of the requested product. From the Seller perspective, on each new request (▷), the seller reads the product that is to be bought, sends back the price, and then connects to the Shipper to send the *prod*uct information while delegating to Shipper the remainder of the communication. Finally, the Shipper, on each new request, receives the product and sends back the shipping `details` over that channel.

To model this interaction, we will "merge" all communications into a single (coordinated) shared cell. Threads will wait on the shared cell for specific tags to appear in the shared state. Similarly, we must model the internal states of the "channel" (i.e. ready to receive, ready to send, etc.) explicitly. Although our idiom hides these temporary states from the programmer, they are needed to encode the interaction in our rely-guarantee protocols.

We begin by directly translating the types above into our protocols, ignoring missing "busy-waiting" states and instead focusing only on the useful transitions and temporary (`idle`) states (see Figure 4.18 for a schematic of how the different channel's states occur within the communication

channel).

$$\begin{array}{rl}
\texttt{Buyer}: & \mathbf{rw}\ c\ \texttt{idle0\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{buy\#}\mathit{prod}\ ; \\
& \mathbf{rw}\ c\ \texttt{price\#}p \Rightarrow \mathbf{rw}\ c\ \texttt{idle2\#}[]\ ; \\
& \mathbf{rw}\ c\ \texttt{details\#}d \Rightarrow \mathbf{rw}\ c\ []\ ; \\
& \mathbf{rw}\ c\ [] \\
\\
\texttt{Seller}: & \mathbf{rw}\ c\ \texttt{buy\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{idle1\#}[]\ ; \\
& \mathbf{rw}\ c\ \texttt{idle1\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{price\#}p\ ; \\
& \mathbf{rw}\ c\ \texttt{idle2\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{product\#}\mathit{prod}\ ; \\
& \mathbf{none} \\
\\
\texttt{Shipper}: & \mathbf{rw}\ c\ \texttt{product\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{idle3\#}[]\ ; \\
& \mathbf{rw}\ c\ \texttt{idle3\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{details\#}d\ ; \\
& \mathbf{none}
\end{array}$$

Where the initial state is of the cell is "**rw** c `idle0#`[]".

We now add the necessary waiting states, but use the following idiom to reduce the syntactic burden and only list the tags that must be waited on.

$$\mathbf{wait}\ \overline{A}\ \mathbf{else}\ P\ \triangleq\ \mathbf{rec}\ X.(\ (A_0 \Rightarrow A_0; X)\ \oplus\ ...\ \oplus\ (A_n \Rightarrow A_n; X)\ \oplus\ P\ )$$

We use the abbreviation above to simplify the syntax of listing each "wait" state, although our $\overline{A}$ will be limited to listing tags and not the full type. Note that instead of manually inserting **wait**...**else**, we could instead use/adapt our protocol composition mechanism to detect any missing state/steps and introduce waiting on the required steps to ensure safe composition for this particular kind of message-passing usage. Still, for the purposes of this example we list the missing states explicitly but gray out those components of the protocol to preserve (some) clarity:

$$\begin{array}{rl}
\texttt{Buyer}: & \mathbf{rw}\ c\ \texttt{idle0\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{buy\#}\mathit{prod}\ ; \\
& \mathbf{wait}\ \texttt{buy}, \texttt{idle1}\ \mathbf{else}\quad \mathbf{rw}\ c\ \texttt{price\#}p \Rightarrow \mathbf{rw}\ c\ \texttt{idle2\#}[]\ ; \\
& \mathbf{wait}\ \texttt{idle2}, \texttt{idle3}, \texttt{product}\ \mathbf{else}\quad \mathbf{rw}\ c\ \texttt{details\#}d \Rightarrow \mathbf{rw}\ c\ []\ ; \\
& \mathbf{rw}\ c\ [] \\
\\
\texttt{Seller}: & \mathbf{wait}\ \texttt{idle0}\ \mathbf{else}\quad \mathbf{rw}\ c\ \texttt{buy\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{idle1\#}[]\ ; \\
& \mathbf{rw}\ c\ \texttt{idle1\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{price\#}p\ ; \\
& \mathbf{wait}\ \texttt{price}\ \mathbf{else}\quad \mathbf{rw}\ c\ \texttt{idle2\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{product\#}\mathit{prod}\ ; \\
& \mathbf{none} \\
\\
\texttt{Shipper}: & \mathbf{wait}\ \texttt{idle0}, \texttt{idle1}, \texttt{idle2}, \texttt{buy}, \texttt{price}\ \mathbf{else} \\
& \mathbf{rw}\ c\ \texttt{product\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{idle3\#}[]\ ; \\
& \mathbf{rw}\ c\ \texttt{idle3\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{details\#}d\ ; \\
& \mathbf{none}
\end{array}$$

In our scheme, all participants must be aware of *all* (public) states that may appear on the

```
fork // Seller server: buy?(prod) ; price!(p) ; Shipper ◃ product!(prod)
  rec L. // recursively waits for new connections
    let c = listenSeller() in
      fork // worker thread to handle this connection
        // the product that is to be bought
        (let product = receive(c) in
          // do something with product to find price
          send( c, FETCH_PRICE(product) );
          // splits the usage of the protocol
          connectShipper(c);
          send( c, product )
        end)
    end;
    L
```

Figure 4.19: Seller code.

shared state. Therefore, each party must explicitly know the state it is to wait on (and not change) because all states are visible and thread/alias interleaving can be non-deterministic.

Implementation Finally, we show a possible implementation of this interaction. To begin the communication, each server must have its own message queue to receive new requests. For instance, by means of a pipe. It is this pipe that stores the channel/cell that connects the two endpoints to establish the desired communication.

Assume that the pipe was already created and shared by assigning to each endpoint a different role in the use of the pipe (consumer or producer). Therefore, the consumer of the pipe will listen for new requests to be pushed into the pipe, while the producer will insert new requests onto the pipe. We then proceed by making `listenShipper` correspond to the consumer of the pipe for the Shipper server (that will do successive `tryTake`s), while `connectShipper` is the corresponding producer that will push new values into the pipe. Analogously, we have `listenSeller` and `connectSeller` for similar uses but for the Seller server.

Do note that `connectShipper` is slightly more complex than just the use of a pipe. Indeed, to create a new channel, `connectShipper` will also have to create the new cell and all the corresponding protocols of the interaction. In the case of this example, this corresponds to splitting the usage of the cell in three protocols: one for the buyer, one for the seller and another for the shipper. On `listenSeller` the seller will receive two protocols one for itself and another for shipper. Through `connectShipper` seller will send shipper the part of the protocol that was delegated to shipper.

Figures 4.21, 4.19, and 4.20 show possible implementations of the three communicating processes using the `receive` and `send` functions discussed above.

```
fork // Shipper server: product?(prod) ; details!(d)
  rec L.
    let c = listenShipper() in
      // get product info
      let product = receive(c) in
        // send shipping details
        send( c, FETCH_SHIPPING_INFO(product) )
      end
    end;
    L
```

Figure 4.20: Shipper code.

```
// Buyer: Seller ◃ buy!(prod) ; price?(p) ; details?(d)
let c = connectSeller() in
  send(c, GET_USER_PRODUCT() );
  let price = receive(c) in
    let details = receive(c) in
      close(c)
    end
  end
end
```

Figure 4.21: Buyer code.

4.6 Related Work

Most of the related work discussion of chapter 3 is still relevant to the work of this chapter. Therefore, we only revisit the aspects that changed from the discussion done in Section 3.6, or discuss aspects that are clarified by the extensions developed in the work of this chapter.

Alms [90] supports, by using a module system as intermediary (and using custom type constructs to "decorate" capabilities as necessary) lock-based sharing, fractional permission, and message-based communication at that module level using relevant language extensions. While they show that the underlying ideas of linear/affine resource management in all those uses fit well at the module level, each extension requires its own special type constructs and language support (that do not appear to have been formalized). The core distinction to our work lies in our low-level handling of mutable state with direct support for lock and fork within the formal system. Likewise, by following $\mathbf{L}^3$ we do not require an intermediary module to enable the separation of reference and capability—besides our more precise tracking of sharing via our protocol types.

Work on access permissions includes support for concurrency such as by using atomic blocks [10, 11] implemented using software transactional memory. Indeed, by exploiting restrictions on mutation, more efficient parallel executions can be exploited [11, 48]. From the perspective of permission expressiveness, our design omits the read-write distinction to focus exclusively on structuring alias interference using more fundamental protocol primitives. Interestingly, although we only

model write-exclusive uses, our types can enforce "effectively" read-exclusive semantics by ensuring that any private change in a cell will be reverted to its original public value (as shown in examples above using existential types over steps). However, this simpler form of read-only cannot capture access permissions' multiple, simultaneous readers semantics. Still, by modeling interference in a more fundamental way, we gain additional expressiveness that enables us to specify ownership recovery and model uses that exceed invariant-based sharing—going beyond their most permissive `share` permission.

In [24], Caires and Seco pointed out that their "behavioral" interpretation of stateful usages resembles imposing temporal constraints by analogy with the kind of temporal constraints imposed by the typing of π-calculus processes. While we also observed the temporal nature of our protocols (also discussed in chapter 3), our types are not related to behavioral types even if we share some relation to the π-calculus (as we discussed in Section 4.5). Our protocols' temporal constraints arise from the need to stabilize the type information when faced with potential interference caused by non-deterministic alias/thread interleaving, making it necessary to check that the type is valid regardless of *when* it may be used. The notion of a behavioral type used in [24] fits instead in a different syntactic interpretation of the typing conditions shown in chapter 2. As we discussed in that chapter, they focus on full state encapsulation (so that the typestate becomes hidden) and where the non-deterministic alias/thread interleaving does not play a role at the level of their types.

By converting our focus/defocus into concrete lock/unlock constructs we also come closer to the work of Krishnaswami *et al.* [56], as we also forfeit reasoning about re-entrant uses of shared state and just let the execution deadlock (while their work simply diverges the execution).

Many others have pointed out links between message-passing and shared memory interference. Jones in [55] made the distinction between "tightly" coupled and "loosely" coupled interference, with focus on the "tightly" coupled case (shared memory). Alms [90] provides some support for message-passing through a special-purpose library and type constructs, based on session-types [53], that fit rather well within their typing framework. Even $\mathbf{L}^3$ [5] noted the "strong update"-like notions of typing type-changing channels/session, and indeed others have shown the connection of session types to linear logic [23] (the same logical principles that guide the design of $\mathbf{L}^3$).

Our protocols can be interpreted as modeling the causality of state changes. For instance, "only after seeing state X will the current alias mutate the state to Y". This naturally gives rise to the possibility of modeling choreography principles of message-passing systems, such as the π-calculus [66], but in a non-distributed shared-memory protocol framework. In this scenario, a channel can be precisely encoded as a shared cell that models the channel's changing session properties. Thus, any interaction between threads is modeled indirectly via inspection of the contents of the shared cell—akin to the channel's buffer. Instead of sending/receiving a value through a channel, the exchanged value is stored/read from a shared cell. Besides storing the value, the cell can also coordinate the use of the state by using tags to track the evolution of the session's type-/properties. The richness of our shared state interactions means that we can immediately support fairly complex session-based mechanisms within our protocol framework. However, this flexibility comes at the cost of requiring a more complex composition mechanism (protocol composition)

than the composition mechanisms normally used in strictly choreographed session-based concurrency. Also note that, unlike some session-based works, we did not consider the issue of guaranteeing deadlock-freedom. We provide a simple example of encoding non-distributed message-passing in Section 4.5, to clarify these claims.

Perhaps the most important distinction between our work and others resides in our protocol design mapping low-level uses of shared state to steps in the protocol type. While prior work captures high-level per-module notions of interference, our design models interference directly in the types in ways that are mapped directly to individual uses of lock-unlock/focus-defocus that access shared state. This means that locked/focused cells can move around module boundaries and we can also abstract components of a protocol by using standard notions of type abstraction.

By using steps to model how interference evolves over time, we can very precisely define the local uses of the shared state without requiring auxiliary variables. For instance, a function could depend on the protocol: $\forall X.(\ \mathbf{rw}\ p\ A \Rightarrow \mathbf{rw}\ p\ B\ ;\ X\ )$ that only requires a single step to be available and is polymorphic on what other steps may exist after that single use. Therefore, our design is meant to guide the programmer on how to reason locally about (safe) interference by expressing the uses of the shared state directly in a protocol type. Through protocol composition, we ensure that the complete use of those shared resources compose safely regardless of how such uses may be interleaved later at run-time. Thus, the interference expected by a single local protocol conforms to the global combined interference that may be produced by any protocol of that state at that point in time. Protocol composition ensures that these individual pieces of the complete interference "cake" are globally consistent throughout the full lifetime of that shared state.

Chapter 5

Composition and Subtyping Algorithms

We now discuss the main algorithmic aspects of our approach. Our focus is on the properties of the two core algorithms that check protocol composition and subtyping on types. These two algorithms are to work automatically, without the need for manual intervention from the programmer once the required input is provided.

It should be noted that the rest of the language may be potentially undecidable if additional type annotations are not used. For instance, the search space for finding a set of protocols that compose safely is potentially infinite, which is why we used the share keyword in chapter 3. With the aim of improving readability and the simplicity of this presentation, we omitted potentially relevant (for algorithmic type checking purposes) type annotations in chapter 4. Therefore, we discuss how to address this issue by potentially adding the necessary (extra) annotations. Intuitively, an implementation can place an upper bound on the number of types to be considered during type checking, and require explicit annotations to consider "larger" sets of types (such as when splitting resources). Naturally, such a typing should still be sound but may no longer be complete w.r.t. the formal type system discussed in previous chapters.

This chapter presents the algorithmic implementations of subtyping (on types) and protocol composition, the core procedures of our technique, and briefly discusses the path to a full implementation. We discuss the relevant properties of our two algorithms, namely soundness and completeness (w.r.t. the formal definition shown in previous chapters), and show that these algorithms terminate when given well-formed inputs. To enable this discussion, we begin by presenting well-formedness conditions that restrict the kind of types that our system allows, ensuring that the structure of any type is regular. These conditions were applied to all types in previous chapters, but are only discussed in this chapter.

Note that our algorithms assume the existence of type equality and type unification procedures [73]. Both procedures are generally straightforward to implement. Equality simply checks the structural equality of two types, and unification follows similar lines but attempts to unify a variable with the structure of the remaining type. The prototype implementation uses these simple procedures, but we do not show the details of those algorithms in here.

5.1 Ensuring Regular Type Structure

Both subtype and protocol composition algorithms may have to recur on the inner sub-terms of a type. For instance, checking that $\texttt{int} \multimap \texttt{bool} <: \texttt{string} \multimap \texttt{double}$ is not valid requires checking $\texttt{string} <: \texttt{int}$ and $\texttt{bool} <: \texttt{double}$, c.f. (ST:FUNCTION). Consequently, it becomes important to ensure that the set of sub-terms of a type is finite so that we can be sure both algorithms will converge to a result after an arbitrary, but finite, number of recursive steps. The main problem is in finding a bound on the unfolding of recursive types. More specifically, we must ensure that our use of recursive types with parameters still produces types that are regular in their structure, regardless of the arguments used.

As we have seen in previous chapters, both subtype and protocol composition will build a co-inductive proof. The corresponding algorithm will only terminate if the proof reaches a "loop" in its derivation that closes the derivation. Consequently, we must show that any well-formed type will enable such a loop to be reached. The fundamental property is to ensure that the structure of the derivation is regular. In our system, because we have both parametric polymorphism and recursive types with parameters, ensuring this regularity is non-trivial and requires more than strict derivation equality.

For our purposes here, our equivalence relation will only consider renaming of type/location variables and weakening of assumptions. Consequently, we will impose well-formedness conditions on our types to ensure that any well-formed type will produce derivations that are equivalent up to those two conditions. We leave as future work devising other, perhaps less restrictive, well-formedness conditions and equivalences since our focus here is to discuss the implementation of protocol composition in the clearest possible terms.

To illustrate an irregular unfold, consider for instance the following recursive type, R, that uses a single (type) parameter, X:

$$(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))[\texttt{int}] \tag{Ex.1}$$

Unfolding this type, and its resulting sub-terms, produces the following sequence of types (we underline the "fixed" part of the recursive type to highlight how that same unfold produces irregular types over its argument):

$$\begin{aligned} & \underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [\texttt{int}] \\ \texttt{int} \multimap\ & \underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [\texttt{int} \multimap \texttt{int}] \\ \texttt{int} \multimap \texttt{int} \multimap\ & \underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [(\texttt{int} \multimap \texttt{int}) \multimap (\texttt{int} \multimap \texttt{int})] \\ & \ldots \end{aligned}$$

for clarity, we replace the underlined type with just R to highlight the irregular set of sub-terms:

$$\begin{aligned} & R[\texttt{int}] \\ \texttt{int} \multimap\ & R[\texttt{int} \multimap \texttt{int}] \\ \texttt{int} \multimap \texttt{int} \multimap\ & R[(\texttt{int} \multimap \texttt{int}) \multimap (\texttt{int} \multimap \texttt{int})] \\ & \ldots \end{aligned}$$

We see that the type that results from unfolding is not regular, as the use of the recursive variable R in "**rec** $R(X).($ **int** $\multimap R[X \multimap X]$ $)$" produces a type that is non-repeating. Consequently, if such a use were allowed, it would make it impossible for an algorithm that traverses all sub-terms of a type to terminate since the type above does not present a finite, regular structure due to its ever growing argument that is applied to the recursive type R.

To forbid uses such as the one above, we limit the kind of arguments that may be applied to a recursive type variable (such as R above) via well-formedness rules (for the full set of rules, see appendix A.1.1). We restrict the arguments that can be applied to a recursive type variable to be limited to location variables or type variables, and exclude recursive type variables:

$$\frac{(X : k_0 \to ... \to k_n \to \textbf{type}) \in \Gamma \qquad (U_i : k_i) \in \Gamma \qquad i \in \{0, ..., n\}}{\Gamma \vdash X[\overline{U}]\ \textbf{type}}$$

The rule states that for a given recursive type variable X (recursive type variables have a "... $\to$ **type**" kind), its arguments ($\overline{U}$) must each be an assumption of compatible kind ($(U_i : k_i) \in \Gamma$). Since we are considering each individual k_i of X, these can only be either a **loc** or a **type** (and never of the form "... $\to$ **type**") which effectively enforces that only location variables or (non-recursive) type variables can be used in this context. Thus, applications of the form $R[R]$ are forbidden since R is a recursive type variable, and $R[X \multimap X]$ is also forbidden since the argument is of a function type (not a type/location variable).

Note, however, that the argument applied to the recursive type is *not* restricted to just type/location variables and instead is only required to be of the desired kind:

$$\frac{\begin{array}{c} u_0 : k_0, ..., u_n : k_n, X : k_0 \to ... \to k_n \to \textbf{type} \vdash A\ \textbf{type} \\ \Gamma \vdash U_i\ k_i \qquad k_i = \texttt{kind}(u_i) \qquad i \in \{0, ..., n\} \end{array}}{\Gamma \vdash (\textbf{rec}\ X(\overline{u}).A)[\overline{U}]\ \textbf{type}}$$

where: $\texttt{kind}(l) = \textbf{loc}$ and $\texttt{kind}(X) = \textbf{type}$. Thus, using **int** as argument in (**rec** $R(X).A$)[**int**] is legal. However, because we only allow each parameter of a recursive type to be either of kind **type** or kind **loc**, recursive type variables cannot appear as arguments (in $\overline{U}$) even in this situation. To preserve the well-formedness condition on uses of $X[\overline{U'}]$ we must also avoid situations where substitution from other **rec**s may replace some argument in $\overline{U'}$ with a non-variable type before X is unfolded. Therefore, the body of **rec**, i.e. A, must ignore all other variables that are outside the top-level **rec**, so that substitution of any element in $\overline{U'}$ will only occur as the **rec** is unfolded.

However, only using the restrictions above is still not sufficient to ensure that the algorithms will terminate, since the resulting set of sub-terms may still be irregular. Consider the following type:

$$(\textbf{rec}\ V(Z).(\forall X <: (A \multimap Z).V[X]))[\textbf{int}] \tag{Ex.2}$$

If we traverse the sub-terms of this type, we see that the *typing context* of the "$V[X]$" sub-term is irregular, although the type structure of $V[...]$ itself remains regular.

To illustrate this, we are going to look into multiple unfolds of V but only show the premise that is used to check that $V[...]$ is well-formed. To highlight the renaming on each unfold, each

new use of X is indexed with ever growing integers. Although we are traversing the **rec**'s sub-terms (automatically unfolding V to continue such traversal), we omit all "**: type**" assertions to focus instead on the typing context that is used to check that "$\Gamma \vdash V[...]$ **type**" (i.e. that $V[...]$ is well-formed):

$$\begin{array}{rcl} \cdot & \vdash & V[\mathtt{int}] \\ X_0 <: A \multimap \mathtt{int} & \vdash & V[X_0] \\ X_1 <: A \multimap X_0,\ X_0 <: A \multimap \mathtt{int} & \vdash & V[X_1] \\ X_2 <: A \multimap X_1,\ X_1 <: A \multimap X_0,\ X_0 <: A \multimap \mathtt{int} & \vdash & V[X_2] \\ & \ldots & \end{array}$$

Consequently, for a recursive type to be well-formed we must also ensure that the enclosing context of future unfolds is regular since it is not enough to only look at the type's structure alone.

We restrict the type of the bound of a $\forall$ or $\exists$ such that the bound must be well-formed in the empty context "$\cdot \vdash A$ **type**" in any "$\exists X <: A.B$" or "$\forall X <: A.B$" types via the following well-formdness conditions (c.f. appendix A.1.1):

$$\frac{\cdot \vdash A_0 \textbf{ type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1 \textbf{ type}}{\Gamma \vdash \forall X <: A_0.A_1 \textbf{ type}} \qquad \frac{\cdot \vdash A_0 \textbf{ type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1 \textbf{ type}}{\Gamma \vdash \exists X <: A_0.A_1 \textbf{ type}}$$

These conditions naturally ensure that the typing contexts in a type must be regular since the typing context is essentially fixed and cannot change on each unfold. We leave as future work relaxing this condition, but for our discussion here, this well-formedness restriction is enough to type our examples and provides an interesting domain for checking safe protocol composition.

Still, our constraints enable some flexibility such as the case of the following type, that can be considered regular by considering renaming of variables and weakening:

$$(\textbf{rec } M(Y).(Y \multimap \forall X <: \textbf{top}.M[X]))[\mathtt{int}] \tag{Ex.3}$$

As above, we illustrate the case via successive unfolds but only show the typing context used to check that $\Gamma \vdash M[...]$ **type** (i.e. that $M[...]$ is well-formed):

$$\begin{array}{rcl} \cdot & \vdash & M[\mathtt{int}] \\ X_0 <: \textbf{top} & \vdash & M[X_0] \\ X_1 <: \textbf{top}, X_0 <: \textbf{top} & \vdash & M[X_1] \\ X_2 <: \textbf{top}, X_1 <: \textbf{top}, X_0 <: \textbf{top} & \vdash & M[X_2] \\ & \ldots & \end{array}$$

By inversion on weakening of assumptions, we can consider the last context to only really require the "$X_2 <:$ **top**" assumption (since the other variables do not occur in $M[X_2]$). By renaming X_0 and X_2 to some fresh variable, both the first and third types can be deduced equivalent ($\equiv$)—which would enable to close a co-inductive proof that traverses M's sub-terms.

$$\frac{\begin{array}{c} Z <: \textbf{top} \vdash M[Z] \quad \equiv \quad Z <: \textbf{top} \vdash M[Z] \\ \hline (X_2 <: \textbf{top})\{Z/X_2\} \vdash (M[X_2])\{Z/X_2\} \quad \equiv \quad (X_0 <: \textbf{top})\{Z/X_0\} \vdash (M[X_0])\{Z/X_0\} \end{array}}{X_2 <: \textbf{top}, X_1 <: \textbf{top}, X_0 <: \textbf{top} \vdash M[X_2] \quad \equiv \quad X_0 <: \textbf{top} \vdash M[X_0]}$$

Thus, we consider equivalence up to renaming of variables and weakening of assumptions—besides the other restrictions discussed above.

5.1.1 Finite Sub-terms

$$
\begin{array}{lcl}
\textbf{sts}(\ \Gamma \vdash \textbf{none}\ ) & = & \{\ \Gamma \vdash \textbf{none}\ \} \\
\textbf{sts}(\ \Gamma \vdash \textbf{top}\ ) & = & \{\ \Gamma \vdash \textbf{top}\ \} \\
\textbf{sts}(\ \Gamma \vdash \textbf{ref}\ p\ ) & = & \{\ \Gamma \vdash \textbf{ref}\ p\ \} \\
\textbf{sts}(\ \Gamma \vdash \textbf{rw}\ p\ A\ ) & = & \{\ \Gamma \vdash \textbf{rw}\ p\ A\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \\
\textbf{sts}(\ \Gamma \vdash\ !A\ ) & = & \{\ \Gamma \vdash\ !A\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \\
\textbf{sts}(\ \Gamma \vdash A \multimap B\ ) & = & \{\ \Gamma \vdash A \multimap B\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \cup \textbf{sts}(\ \Gamma \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash A :: B\ ) & = & \{\ \Gamma \vdash A :: B\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \cup \textbf{sts}(\ \Gamma \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash A * B\ ) & = & \{\ \Gamma \vdash A * B\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \cup \textbf{sts}(\ \Gamma \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash A \oplus B\ ) & = & \{\ \Gamma \vdash A \oplus B\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \cup \textbf{sts}(\ \Gamma \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash A\ \&\ B\ ) & = & \{\ \Gamma \vdash A\&B\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \cup \textbf{sts}(\ \Gamma \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash A \Rightarrow B\ ) & = & \{\ \Gamma \vdash A \Rightarrow B\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \cup \textbf{sts}(\ \Gamma \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash A\ ;\ B\ ) & = & \{\ \Gamma \vdash A\ ;\ B\ \} \cup \textbf{sts}(\ \Gamma \vdash A\ ) \cup \textbf{sts}(\ \Gamma \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash [\overline{\mathtt{f} : A}]\ ) & = & \{\ \Gamma \vdash [\overline{\mathtt{f} : A}]\ \} \cup \bigcup_i \textbf{sts}(\ \Gamma \vdash A_i\ ) \\
\textbf{sts}(\ \Gamma \vdash \sum_i \mathtt{t}_i \# A_i\ ) & = & \{\ \Gamma \vdash \sum_i \mathtt{t}_i \# A_i\ \} \cup \bigcup_i \textbf{sts}(\ \Gamma \vdash A_i\ ) \\
\textbf{sts}(\ \Gamma \vdash \forall l.A\ ) & = & \{\ \Gamma \vdash \forall l.A\ \} \cup \textbf{sts}(\ \Gamma, l : \textbf{loc} \vdash A\ ) \\
\textbf{sts}(\ \Gamma \vdash \exists l.A\ ) & = & \{\ \Gamma \vdash \exists l.A\ \} \cup \textbf{sts}(\ \Gamma, l : \textbf{loc} \vdash A\ ) \\
\textbf{sts}(\ \Gamma \vdash \forall X <: A.B\ ) & = & \{\ \Gamma \vdash \forall X <: A.B\ \} \cup \textbf{sts}(\ \cdot \vdash A\ ) \cup \textbf{sts}(\ \Gamma, X : \textbf{type}, X <: A \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash \exists X <: A.B\ ) & = & \{\ \Gamma \vdash \exists X <: A.B\ \} \cup \textbf{sts}(\ \cdot \vdash A\ ) \cup \textbf{sts}(\ \Gamma, X : \textbf{type}, X <: A \vdash B\ ) \\
\textbf{sts}(\ \Gamma \vdash X[\overline{U}]\ ) & = & \{\ \Gamma \vdash X[\overline{U}]\ \} \\
\textbf{sts}(\ \Gamma \vdash (\textbf{rec}\ X(\overline{u}).A)[\overline{U}]\ ) & = & \{\ \Gamma \vdash (\textbf{rec}\ X(\overline{u}).A)[\overline{U}]\ \} \cup \textbf{sts}(\ \Gamma \vdash A\{(\textbf{rec}\ X(\overline{u}).A)/X\}\{\overline{U}/\overline{u}\}\ )
\end{array}
$$

Note: we omit **type** from $\Gamma \vdash A$ **type** for conciseness.

Figure 5.1: Computing the (infinite) set of sub-terms of a type (**sts**).

We now discuss the regularity of the structure of our well-formed types. These lemmas are essential to show that there is a bound in the number of members of the set of types that an algorithm will recur on. Thus, when we recur on some type's sub-term, the domain of possible sub-terms of that type must necessarily be monotonically shrinking since its set of distinct sub-terms is bounded by a finite number. To simplify the discussion, instead of counting the exact size of the set of distinct sub-terms of a type, our proofs will often resort to simpler *overapproximations* of that set. Since even the dimension of that overapproximation is finite, then the set of distinct sub-terms of any well-formed type will also necessarily be finite.

Figure 5.1 shows a straightforward way to compute the infinite set of sub-terms of a type. The most important case to note is that of **rec**, which unfolds the recursive type and then continues the analysis over the unfolded type. Consequently, **sts**'s definition may not terminate if we do not define a way to identify repeating sub-terms and stop further (unnecessary) recursive calls to **sts**.

$$
\begin{array}{lcl}
\mathbf{st}(\,\Gamma \vdash A\,) & = & \mathbf{st}(\,\Gamma \vdash A, \emptyset\,) \\
\mathbf{st}(\,\Gamma \vdash A, v\,) & = & v \ \mathtt{if}\ (\Gamma' \vdash A') \in v \ \mathbf{and}\ (\Gamma \vdash A) \equiv (\Gamma' \vdash A') \\
\mathbf{st}(\,\Gamma \vdash \mathbf{none}, v\,) & = & v \cup \{\,\Gamma \vdash \mathbf{none}\,\} \\
\mathbf{st}(\,\Gamma \vdash \mathbf{top}, v\,) & = & v \cup \{\,\Gamma \vdash \mathbf{top}\,\} \\
\mathbf{st}(\,\Gamma \vdash \mathbf{ref}\ p, v\,) & = & v \cup \{\,\Gamma \vdash \mathbf{ref}\ p\,\} \\
\mathbf{st}(\,\Gamma \vdash \mathbf{rw}\ p\ A, v\,) & = & v \cup \{\,\Gamma \vdash \mathbf{rw}\ p\ A\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \\
\mathbf{st}(\,\Gamma \vdash\ !A, v\,) & = & v \cup \{\,\Gamma \vdash\ !A\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \\
\mathbf{st}(\,\Gamma \vdash A \multimap B, v\,) & = & v \cup \{\,\Gamma \vdash A \multimap B\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \cup \mathbf{st}(\,\Gamma \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash A :: B, v\,) & = & v \cup \{\,\Gamma \vdash A :: B\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \cup \mathbf{st}(\,\Gamma \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash A * B, v\,) & = & v \cup \{\,\Gamma \vdash A * B\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \cup \mathbf{st}(\,\Gamma \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash A \oplus B, v\,) & = & v \cup \{\,\Gamma \vdash A \oplus B\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \cup \mathbf{st}(\,\Gamma \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash A \,\&\, B, v\,) & = & v \cup \{\,\Gamma \vdash A \& B\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \cup \mathbf{st}(\,\Gamma \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash A \Rightarrow B, v\,) & = & v \cup \{\,\Gamma \vdash A \Rightarrow B\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \cup \mathbf{st}(\,\Gamma \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash A\ ;\ B, v\,) & = & v \cup \{\,\Gamma \vdash A\ ;\ B\,\} \cup \mathbf{st}(\,\Gamma \vdash A, v\,) \cup \mathbf{st}(\,\Gamma \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash [\overline{\mathtt{f} : A}, v]\,) & = & v \cup \{\,\Gamma \vdash [\overline{\mathtt{f} : A}]\,\} \cup \bigcup_i \mathbf{st}(\,\Gamma \vdash A_i, v\,) \\
\mathbf{st}(\,\Gamma \vdash \sum_i \mathtt{t}_i \# A_i, v\,) & = & v \cup \{\,\Gamma \vdash \sum_i \mathtt{t}_i \# A_i\,\} \cup \bigcup_i \mathbf{st}(\,\Gamma \vdash A_i, v\,) \\
\mathbf{st}(\,\Gamma \vdash \forall l.A, v\,) & = & v \cup \{\,\Gamma \vdash \forall l.A\,\} \cup \mathbf{st}(\,\Gamma, l : \mathbf{loc} \vdash A, v\,) \\
\mathbf{st}(\,\Gamma \vdash \exists l.A, v\,) & = & v \cup \{\,\Gamma \vdash \exists l.A\,\} \cup \mathbf{st}(\,\Gamma, l : \mathbf{loc} \vdash A, v\,) \\
\mathbf{st}(\,\Gamma \vdash \forall X <: A.B, v\,) & = & v \cup \{\,\Gamma \vdash \forall X <: A.B\,\} \cup \mathbf{st}(\,\cdot \vdash A, v\,) \cup \mathbf{st}(\,\Gamma, X : \mathbf{type}, X <: A \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash \exists X <: A.B, v\,) & = & v \cup \{\,\Gamma \vdash \exists X <: A.B\,\} \cup \mathbf{st}(\,\cdot \vdash A, v\,) \cup \mathbf{st}(\,\Gamma, X : \mathbf{type}, X <: A \vdash B, v\,) \\
\mathbf{st}(\,\Gamma \vdash X[\overline{U}], v\,) & = & v \cup \{\,\Gamma \vdash X[\overline{U}]\,\} \\
\mathbf{st}(\,\Gamma \vdash (\mathbf{rec}\ X(\overline{u}).A)[\overline{U}], v\,) & = & \mathbf{st}(\,\Gamma \vdash A\{(\mathbf{rec}\ X(\overline{u}).A)/X\}\{\overline{U}/\overline{u}\}, v \cup \{\,\Gamma \vdash (\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]\,\}\,)
\end{array}
$$

Where v is a set of "$\Gamma \vdash A$" elements. Note that the rules are ordered so that we try to apply the $\equiv$ case (and detect cycles) before using any of the remaining rules.

Figure 5.2: Computing the set of sub-terms of a type, up to equivalence ($\equiv$).

As discussed above, we consider equivalence ($\equiv$) up to renaming of variables and weakening of assumptions defined as follows (for any two well-formed types, and such that any premise must also obey type well-formedness):

$$
\begin{array}{lcll}
\Gamma \vdash A & \equiv & \Gamma \vdash A & & \text{(equality)} \\
\Gamma_0, \Gamma_1 \vdash A_0 & \equiv & \Gamma_2, \Gamma_3 \vdash A_1 & \mathtt{if}\ \Gamma_1 \vdash A_0 \ \equiv\ \Gamma_3 \vdash A_1 & \text{(weakening)} \\
\Gamma_0 \vdash A_0 & \equiv & \Gamma_1 \vdash A_1 & \mathtt{if}\ \Gamma_0\{Z/X\} \vdash A_0\{Z/X\} \ \equiv\ \Gamma_1\{Z/Y\} \vdash A_1\{Z/Y\}\ \mathbf{and}\ Z\ \mathbf{fresh} & \text{(renaming type)} \\
\Gamma_0 \vdash A_0 & \equiv & \Gamma_1 \vdash A_1 & \mathtt{if}\ \Gamma_0\{l/t'\} \vdash A_0\{l/t'\} \ \equiv\ \Gamma_1\{l/t\} \vdash A_1\{l/t\}\ \mathbf{and}\ l\ \mathbf{fresh} & \text{(renaming loc)}
\end{array}
$$

With the conditions above, we can approximate the infinite set of sub-terms computed by **sts** with one that computes an equivalent (up to renaming and weakening) but finite set since we are just collapsing equivalent members of that set (i.e. each member now represents the class of types defined up to renaming and weakening). Thus, the algorithm stopping condition would simply have to carry a set of visited sub-terms and not recur on equivalent sub-terms that it already visited.

Lemma 12 (Finite Uses)**.** *Given a well-formed recursive type* $(\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]$ *the number of possible uses of* X *in* A *such that* $\Gamma \vdash X[\overline{U'}]$ ***type*** *is bounded.*

Proof. Our well-formedness restrictions enforce that any well-formed $X[\overline{U'}]$ can only contain either location or type variables in the $\overline{U'}$ set. For those variables to be themselves well-formed they must be present in Γ. Since Γ is necessarily a finite set of assumptions it must contain a finite number of different location/type variables.

We have that for m location/type variables in Γ and if the set $\overline{u}$ has size n, then there exists at most m^n tuples of n variables (repetition is allowed since types are pure), each containing different type/location applications that can be used in a well-formed $X[\overline{U'}]$ type. (Note the "at most" since we are ignoring the type/location variable distinction and just counting both kinds together.) □

Lemma 13 (Finite Unfolds)**.** *Unfolding a well-formed recursive type* (***rec*** $X(\overline{u}).A)[\overline{U}]$ *produces a finite set of variants of that original recursive type that (at most) contains: permutations of* $\overline{U}$*, or a set of mixtures of* $\overline{U}$ *with some type/location variables representing a class of equivalent (≡) types.*

Proof. By the well-formedness conditions on $X[\overline{U'}]$, we have that $\overline{U'}$ will list a set of type/location variables, which include the recursive type's parameters ($\overline{u}$). Thus, for any use of the recursive type variable X that may occur in **rec**, we have two cases:

- Either $\overline{U'}$ only contains uses of the recursive type's parameters (i.e. $\overline{u}$).
  Then, this means that an unfold will produce a (**rec** $X(\overline{u})A)[\overline{U'}]$ which, at most, only differs in the order of the elements in $\overline{U'}$. (Recall that since all variables in A cannot be bound to elements outside the top-level **rec**, A remains invariant over unfold.) Consequently, for n elements in $\overline{U}$ there can be n^n different orderings in $\overline{U'}$ since repetition is allowed and both $\overline{U}$ and $\overline{U'}$ must have the same number of elements.
- Or $\overline{U'}$ contains a mixture of type/location variables (that must have been declared by a ∀ or ∃ inside the **rec**) combined with some elements of $\overline{u}$.
  Lets consider a $\forall Y <: B.X[Y]$ use inside of A. When this type is unfolded, we obtain a (**rec** $X(...).(...))[Y]$ type where Y is now the argument of that **rec** (rather than previously provided top-level arguments that were used in the top-level **rec**). Furthermore, our well-formedness conditions on ∀ and ∃ ensure that the bound inside the unfolded **rec** will remain invariant over unfold, so that future unfolds will produce equivalent uses of Y's (potentially renamed). Because by well-formedness conditions the bound is isolated from any variable (i.e. typed in the empty context), this ensures that future unfolds will type Y in a Γ that is equivalent up to weakening (by ignoring past uses of Y). (Note that the order of the bounds in Γ is not important, as the assumptions in Γ form an unordered set.)
  Thus, by (Finite Uses) we have that each use of X is bounded by a finite number of different mixtures/permutations of location/type variables in Γ. Consequently, the set of types represented by all the different unfolds produces a finite set of recursive types representing the infinite set of different unfolds, that yet is equivalent up to renaming and weakening to that finite set.
  Lets assume that a top-level (**rec** $X(...).A)[...]$ contains x number of different uses of X in A. As done in the proof of (Finite Uses) we know that each variable will have m^n tuples of n variables for a Γ context containing m type/location variables and where X expects n arguments. Combining those variables with the u possible number of different concrete types provided in the top-level **rec** yields $(m+u)^n$ possible different uses on each X. Consequently,

for x uses of X, we can estimate $x * (m + u)^n$ different elements in the set if we are overapproximating by considering that all uses of X have the largest set of Γ that may appear in any use of X.

We conclude by combining the two finite sets.

□

Lemma 14 (Finite Sub-Terms)**.** *Given a well-formed type A, such that* $\Gamma \vdash A$ ***type****, the set of sub-terms of A is finite up to renaming of variables and weakening of* Γ.

Proof. The proof is reduced to showing that the definition of **st** (Figure 5.2) terminates and that **st** produces a set that is equivalent to the set produced by **sts** (Figure 5.1), up to $\equiv$ types. Equivalence is immediate since the two definitions only differ in the tracking of v, the set of visited sub-terms which enables **st** to stop when it finds a repeating sub-term while **sts** continues indefinitely producing types that are actually $\equiv$.

Termination is straightforward as it follows by the previous lemmas. There is a finite number of different types that any recursive type can generate and all other type constructs produce a finite number of sub-terms, thus traversing this set of types must eventually stop. Each case is simply an application of inversion on the specific well-formedness condition, followed by the application of the induction hypothesis which then enables us to conclude that the combination of two terminating recursive calls to **st** will also have to terminate. Termination on the recursive type case follows immediately by the eventual exhaustion of the finitely many different sub-terms that its unfold may produce, shown in (Finite Unfolds), and that **st** will have to visit.

□

5.2 Protocol Composition Algorithm

The algorithm to check safe protocol composition (Figures 5.3 and 5.4) is a straightforward interpretation of the protocol composition rules shown in chapter 4 (Figures 4.3, 4.4, and 4.5). The algorithm simply goes from conclusion to premise and makes the necessary distinction between stepping on protocols and on non-protocol resources, via the syntactical structure of S. Once all distinct configurations (up to renaming and weakening of Γ) are visited, the algorithm stops since it exhausted the domain of possible different protocol positions. Figures 5.3 and 5.4 show the initial, simpler version of protocol composition and leave for the next sub-section the implementation changes to employ subtyping at the protocol composition level. As stated above, we require well-formed recursive types to be non-bottom. Thus, unfolding a well-formed recursive type will eventually reach a non-recursive type whose type structure can be analyzed.

The algorithm, c, shown obeys the following properties:

Theorem 9 (c soundness)**.** *If* $\mathsf{c}(\Gamma, S, A, B)$ *then* $\Gamma \vdash S \Rrightarrow A \parallel B$.

So that the algorithm's result is consistent with the axiomatic definition.

Theorem 10 (c completeness)**.** *If* $\Gamma \vdash S \Rrightarrow A \parallel B$ *then* $\mathsf{c}(\Gamma, S, A, B)$.

So that any conclusion taken by the axiomatic definition can be matched by a result of the algorithm, given well-formed inputs.

$\boxed{\mathtt{c}(\Gamma,\, S,\, P,\, Q)}$ **Composition, (c)**

(1) $\mathtt{c}(\Gamma,\, S,\, P,\, Q) \triangleq \mathtt{c}(\Gamma,\, S,\, P,\, Q,\, \emptyset)$

(2) $\mathtt{c}(\Gamma,\, S,\, P,\, Q,\, v) \triangleq$
$\forall(\Gamma' \vdash S' \Rrightarrow P' \parallel Q') \in (\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P],\, v) \cup \mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[Q],\, v)).$
$(\, \mathtt{c}(\Gamma',\, S',\, P',\, Q',\, v \cup \langle \Gamma \vdash S \Rrightarrow P \parallel Q \rangle)\,)$ /* (c:Step) and (c:AllStep) */

$\boxed{\mathtt{stp}(\Gamma,\, S,\, P,\, v)}$ **Step, (stp)**

(3) $\mathtt{stp}((\Gamma', \Gamma),\, S,\, \mathcal{R}[P],\, v) \triangleq$ /* (c-rs:Weakening) */
$\emptyset$ **if** $\langle \Gamma \vdash S \Rrightarrow \mathcal{R}[P] \rangle \in v$ /* considering equality up to (eq:Rec) */

(4) $\mathtt{stp}(\Gamma,\, (\mathbf{rec}\ X(\overline{u}).S)[\overline{U}],\, \mathcal{R}[P],\, v) \triangleq$
$\mathtt{stp}(\Gamma,\, S\{\mathbf{rec}\ X(\overline{u}).S/X\}\{\overline{U}/\overline{u}\},\, \mathcal{R}[P],\, v)$ /* (eq:Rec) */

(5) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[(\mathbf{rec}\ X(\overline{u}).P)[\overline{U}]],\, v) \triangleq$
$\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P\{\mathbf{rec}\ X(\overline{u}).P/X\}\{\overline{U}/\overline{u}\}],\, v)$ /* (eq:Rec) */

(6) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[\mathbf{none}],\, v) \triangleq \{\Gamma \vdash S \Rrightarrow \mathcal{R}[\mathbf{none}]\}$ /* (c-rs:None) */

(7) $\mathtt{stp}(\Gamma,\, R_0,\, \mathcal{R}[R_0 \Rightarrow R_1; P],\, v) \triangleq \{\Gamma \vdash R_1 \Rrightarrow \mathcal{R}[P]\}$ /* (c-ss:Step) */

(8) $\mathtt{stp}(\Gamma,\, (R_0 \Rightarrow R_1; Q),\, \mathcal{R}[R_0 \Rightarrow R_1; P],\, v) \triangleq \{\Gamma \vdash Q \Rrightarrow \mathcal{R}[P]\}$ /* (c-ps:Step) */

(9) $\mathtt{stp}(\Gamma,\, S_0 \oplus S_1,\, \mathcal{R}[P],\, v) \triangleq$
$\mathtt{stp}(\Gamma,\, S_0,\, \mathcal{R}[P],\, v) \cup \mathtt{stp}(\Gamma,\, S_1,\, \mathcal{R}[P],\, v)$ /* (c-rs:StateAlternative) */

(10) $\mathtt{stp}(\Gamma,\, S_0 \& S_1,\, \mathcal{R}[P],\, v) \triangleq \mathtt{stp}(\Gamma,\, S_0,\, \mathcal{R}[P],\, v)$ /* (c-rs:StateIntersection) */
(11) $\mathtt{stp}(\Gamma,\, S_0 \& S_1,\, \mathcal{R}[P],\, v) \triangleq \mathtt{stp}(\Gamma,\, S_1,\, \mathcal{R}[P],\, v)$

(12) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P_0 \oplus P_1],\, v) \triangleq \mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P_0],\, v)$ /* (c-rs:ProtocolAlternative) */
(13) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P_0 \oplus P_1],\, v) \triangleq \mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P_1],\, v)$

(14) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P_0 \& P_1],\, v) \triangleq$
$\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P_0],\, v) \cup \mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[P_1],\, v)$ /* (c-rs:ProtocolIntersection) */

Figure 5.3: Protocol Composition Algorithm (1/2).

(15) $\texttt{stp}(\Gamma,\ R,\ \mathcal{R}[R],\ v) \triangleq \{\Gamma \vdash \textbf{none} \Rrightarrow \mathcal{R}[\textbf{none}]\}$ /* (C-SS:RECOVERY) */

(16) $\texttt{stp}(\Gamma,\ R,\ \mathcal{R}[\exists l.P],\ v) \triangleq \texttt{stp}(\Gamma,\ R,\ \mathcal{R}[P\{p/l\}],\ v)$ /* (C-SS:OPENLOC) */

(17) $\texttt{stp}(\Gamma,\ R,\ \mathcal{R}[\exists X <: A_0.P],\ v) \triangleq$ /* (C-SS:OPENTYPE) */
$\quad \texttt{stp}(\Gamma,\ R,\ \mathcal{R}[P\{A_1/X\}],\ v)\ \texttt{if}\ \texttt{sbt}(\Gamma,\ A_1,\ A_0)$

(18) $\texttt{stp}(\Gamma,\ R,\ \mathcal{R}[R \Rightarrow \forall l.P],\ v) \triangleq \texttt{stp}((\Gamma, l : \textbf{loc}),\ R,\ \mathcal{R}[R \Rightarrow P],\ v)$ /* (C-SS:FORALLLOC) */

(19) $\texttt{stp}(\Gamma,\ R,\ \mathcal{R}[R \Rightarrow \forall X <: A.P],\ v) \triangleq$
$\quad \texttt{stp}((\Gamma, X : \textbf{type}, X <: A),\ R,\ \mathcal{R}[R \Rightarrow P],\ v)$ /* (C-SS:FORALTYPE) */

(20) $\texttt{stp}(\Gamma,\ \exists l.P,\ \mathcal{R}[\exists l.Q],\ v) \triangleq \texttt{stp}((\Gamma, l : \textbf{loc}),\ P,\ \mathcal{R}[Q],\ v)$ /* (C-PS:EXISTSLOC) */

(21) $\texttt{stp}(\Gamma,\ \exists X <: A.P,\ \mathcal{R}[\exists X <: A.Q],\ v) \triangleq$ /* (C-PS:EXISTSTYPE) */
$\quad \texttt{stp}((\Gamma, X : \textbf{type}, X <: A),\ P,\ \mathcal{R}[Q],\ v)$

(22) $\texttt{stp}(\Gamma,\ R \Rightarrow \forall l.P,\ \mathcal{R}[R \Rightarrow \forall l.Q],\ v) \triangleq$ /* (C-PS:FORALLLOC) */
$\quad \texttt{stp}((\Gamma, l : \textbf{loc}),\ R \Rightarrow P,\ \mathcal{R}[R \Rightarrow Q],\ v)$

(23) $\texttt{stp}(\Gamma,\ R \Rightarrow \forall X <: A.P,\ \mathcal{R}[R \Rightarrow \forall X <: A.Q],\ v) \triangleq$ /* (C-PS:FORALLTYPE) */
$\quad \texttt{stp}((\Gamma, X : \textbf{type}, X <: A),\ R \Rightarrow P,\ \mathcal{R}[R \Rightarrow Q],\ v)$

(24) $\texttt{stp}(\Gamma,\ R \Rightarrow \forall l.P,\ \mathcal{R}[R \Rightarrow Q],\ v) \triangleq \texttt{stp}(\Gamma,\ R \Rightarrow P\{l/p\},\ \mathcal{R}[R \Rightarrow Q],\ v)$ /* (C-PS:LOCAPP) */

(25) $\texttt{stp}(\Gamma,\ R \Rightarrow \forall X <: A_0.P,\ \mathcal{R}[R \Rightarrow Q],\ v) \triangleq$ /* (C-PS:TYPEAPP) */
$\quad \texttt{stp}(\Gamma,\ R \Rightarrow P\{A_1/X\},\ \mathcal{R}[R \Rightarrow Q],\ v)\ \texttt{if}\ \texttt{sbt}(\Gamma,\ A_1,\ A_0)$

Figure 5.4: Protocol Composition Algorithm (2/2).

(3) $\mathtt{stp}((\Gamma', \Gamma), S, \mathcal{R}[P], v) \triangleq$ /* (c-rs:Weakening) */
$\emptyset$ **if** $\langle \Gamma \vdash S' \Rrightarrow \mathcal{R}[P'] \rangle \in v \wedge \mathtt{sbt}(\Gamma, S, S') \wedge \mathtt{sbt}(\Gamma, P', P)$

(7) $\mathtt{stp}(\Gamma, R_0, \mathcal{R}[R_1 \Rightarrow R_2; P], v) \triangleq \{\Gamma \vdash R_2 \Rrightarrow \mathcal{R}[P]\}$ **if** $\mathtt{sbt}(\Gamma, R_0, R_1)$ /* (c-ss:Step) */

(8) $\mathtt{stp}(\Gamma, (R_0 \Rightarrow R_2; Q), \mathcal{R}[R_1 \Rightarrow R_3; P], v) \triangleq$
$\{\Gamma \vdash Q \Rrightarrow \mathcal{R}[P]\}$ **if** $\mathtt{sbt}(\Gamma, R_0, R_1) \wedge \mathtt{sbt}(\Gamma, R_3, R_2)$ /* (c-ps:Step) */

(15) $\mathtt{stp}(\Gamma, R_0, \mathcal{R}[R_1], v) \triangleq \{\Gamma \vdash \mathbf{none} \Rrightarrow \mathcal{R}[\mathbf{none}]\}$ **if** $\mathtt{sbt}(\Gamma, R_0, R_1)$ /* (c-ss:Recovery) */

Figure 5.5: Subtyping Extension.

Both proofs are straightforward by induction on the structure of the algorithm or the protocol composition derivation. We include a comment on each of the algorithm's rules to indicate which axiomatic rule it refers to.

Theorem 11 (c decidability)**.** *Given well-formed types and a well-formed environment* $\mathsf{c}(\Gamma, S, A, B)$ *terminates.*

Consequently, there is an upper bound on the number of configurations that will be visited by the algorithm. The crucial lemma that supports this theorem was shown above. We showed that any well-formed type essentially contains a finite set of sub-terms, up to renaming and weakening of assumptions. Therefore, even if we consider the product of both sets of sub-terms of A and B, that resulting set must still be finite ensuring that the protocol composition check will eventually exhaust the set of different protocol/type combinations.

5.2.1 Subtyping Extension

The next section will show that the subtyping (on types) algorithm is sound, complete, and decidable and consequently it should be straightforward to show that even when we add the subtyping extensions (Figure 5.5, based on Figure 4.6) protocol composition should remain at least sound and decidable. The problem is whether protocol composition remains complete when we consider the (c-rs:Subsumption) stepping rule of Figure 4.6 since that rule's conclusion does not direct the types to be used in its premise.

It is at this point that the order in subtyping the components of a configuration is important. We see that a protocol that obeys (c-rs:Subsumption) would actually also safely compose without that rule being present, i.e. (c-rs:Subsumption) is admissible. Instead, (c-rs:Subsumption) is only important to close the co-inductive proof "earlier" by reducing the number of configurations that we may need to be visited to check safe composition. Thus, from the algorithmic side, we can simply "embed" the effects that the (c-rs:Subsumption) rule has into the modified (c-rs:Weakening) with subtyping and argue that c remains complete. Even if the algorithm may need to consider more configurations than the axiomatic definition that uses (c-rs:Subsumption), our theorems are only concerned in showing that protocol composition remains valid (not that it will visit the exact same set of configurations).

Theorem 12. *If* $\Gamma \vdash S_1 <: S_0$ *and* $\langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C$ *and* $\Gamma \vdash P_0 <: P_1$ *then* $\langle\, \Gamma \vdash S_1 \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C$ *(without using* (c-rs:Subsumption)*).*

$\boxed{\texttt{sbt}(\,\Gamma,\,A,\,B\,)}$ (i.e.: $\texttt{sbt}(\,\Gamma \vdash A <: B\,)$)

(1) $\texttt{sbt}(\,\Gamma,\,A,\,B\,) \triangleq \texttt{sbt}(\,\Gamma,\,A,\,B,\,\emptyset\,)$

(2) $\texttt{sbt}(\,\Gamma,\,A,\,A,\,t\,) \triangleq \textbf{true}$ /* (ST:SYMMETRY) */

(3) $\texttt{sbt}(\,\Gamma,\,!A,\,B,\,t\,) \triangleq \texttt{sbt}(\,\Gamma,\,A,\,B,\,t\,)$ /* (ST:TOLINEAR) */

(4) $\texttt{sbt}(\,\Gamma,\,!A,\,![],\,t\,) \triangleq \textbf{true}$ /* (ST:PURETOP) */

(5) $\texttt{sbt}(\,\Gamma,\,!A,\,!B,\,t\,) \triangleq \texttt{sbt}(\,\Gamma,\,A,\,B,\,t\,)$ /* (ST:PURE) */

(6) $\texttt{sbt}(\,\Gamma,\,A,\,\textbf{top},\,t\,) \triangleq \textbf{true}$ /* (ST:TOP) */

(7) $\texttt{sbt}(\,\Gamma,\,X,\,B,\,t\,) \triangleq ((X <: A) \in \Gamma) \wedge \texttt{sbt}(\,\Gamma,\,A,\,B,\,t\,)$ /* (ST:TYPEVAR) */

(8) $\texttt{sbt}(\,(\Gamma',\Gamma),\,A,\,B,\,t\,) \triangleq \texttt{sbt}(\,\Gamma,\,A,\,B,\,t\,)$ /* (ST:WEAKENING) */

(9) $\texttt{sbt}(\,\Gamma,\,(A \multimap B),\,(C \multimap D),\,t\,) \triangleq \texttt{sbt}(\,\Gamma,\,C,\,A,\,t\,) \wedge \texttt{sbt}(\,\Gamma,\,B,\,D,\,t\,)$ /* (ST:FUNCTION) */

(10) $\texttt{sbt}(\,\Gamma,\,(A :: B),\,(C :: D),\,t\,) \triangleq \texttt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \texttt{sbt}(\,\Gamma,\,B,\,D,\,t\,)$ /* (ST:STACK) */

(11) $\texttt{sbt}(\,\Gamma,\,(\textbf{rw}\ l\ A),\,(\textbf{rw}\ l\ B),\,t\,) \triangleq \texttt{sbt}(\,\Gamma,\,A,\,B,\,t\,)$ /* (ST:CAP) */

(12) $\texttt{sbt}(\,\Gamma,\,(A * B),\,(C * D),\,t\,) \triangleq$
$(\texttt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \texttt{sbt}(\,\Gamma,\,B,\,D,\,t\,)) \vee$
$(\texttt{sbt}(\,\Gamma,\,A,\,D,\,t\,) \wedge \texttt{sbt}(\,\Gamma,\,B,\,C,\,t\,))$ /* (ST:STAR) */

(13) $\texttt{sbt}(\,\Gamma,\,\exists l.A,\,\exists l.B,\,t\,) \triangleq \texttt{sbt}(\,(\Gamma, l : \textbf{loc}),\,A,\,B,\,t\,)$ /* (ST:LOC-EXISTS) */

(14) $\texttt{sbt}(\,\Gamma,\,\forall l.A,\,\forall l.B,\,t\,) \triangleq \texttt{sbt}(\,(\Gamma, l : \textbf{loc}),\,A,\,B,\,t\,)$ /* (ST:LOC-FORALL) */

(15) $\texttt{sbt}(\,\Gamma,\,\exists X <: A.B,\,\exists X <: A.C,\,t\,) \triangleq \texttt{sbt}(\,(\Gamma, X <: A, X : \textbf{type}),\,B,\,C,\,t\,)$ /* (ST:TYPE-EXISTS) */

(16) $\texttt{sbt}(\,\Gamma,\,\forall X <: A.B,\,\forall X <: A.C,\,t\,) \triangleq \texttt{sbt}(\,(\Gamma, X <: A, X : \textbf{type}),\,B,\,C,\,t\,)$ /* (ST:TYPE-FORALL) */

(17) $\texttt{sbt}(\,(\Gamma',\Gamma),\,A,\,B,\,t\,) \triangleq ((\Gamma \vdash A <: B) \in t)$ /* (EQ:REC) */

Figure 5.6: Subtyping Algorithm (1/2).

The proof is straightforward as protocol composition enforces breaking the $\oplus$ and & cases, and the changes to the algorithm's (c-ss:STEP) and (c-ps:STEP) rules shown in Figure 5.5 ensure that subtyping over a state or protocol is still derivable. Note that there are no subtyping rules for applying subtyping over the inner structure of a protocol.

5.3 Subtyping Algorithm

Our subtyping algorithm (shown in Figures 5.6 and 5.7, based on the axiomatic definition of Figures 4.12 and 4.13) follows the approach of Amadio *et al.* [8]. Therefore, we include a *t*rail set that tracks the set of visited subtyping pairs, enabling the algorithm to close the co-inductive proof when it finds an already visited pair of sub-terms. Recall that unfolding recursive types is done

(18) $\mathtt{sbt}(\,\Gamma,\,(\mathbf{rec}\;X(\overline{u}).A)[\overline{U}],\,(\mathbf{rec}\;Y(\overline{u'}).B)[\overline{U'}],\,t\,) \triangleq$
$\mathtt{sbt}(\,\Gamma,\,A\{\mathbf{rec}\;X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\},\,B\{\mathbf{rec}\;Y(\overline{u'}).B/Y\}\{\overline{U'}/\overline{u'}\},$
$(t \cup (\Gamma \vdash (\mathbf{rec}\;X(\overline{u}).A)[\overline{U}] <: (\mathbf{rec}\;Y(\overline{u'}).B)[\overline{U'}]))\,)$ /* (EQ:REC) */

(19) $\mathtt{sbt}(\,\Gamma,\,(\mathbf{rec}\;X(\overline{u}).A)[\overline{U}],\,B,\,t\,) \triangleq$
$\mathtt{sbt}(\,\Gamma,\,A\{\mathbf{rec}\;X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\},\,B,\,(t \cup (\Gamma \vdash (\mathbf{rec}\;X(\overline{u}).A)[\overline{U}] <: B))\,)$ /* (EQ:REC) */

(20) $\mathtt{sbt}(\,\Gamma,\,A,\,(\mathbf{rec}\;Y(\overline{u'}).B)[\overline{U'}],\,t\,) \triangleq$
$\mathtt{sbt}(\,\Gamma,\,A,\,B\{\mathbf{rec}\;Y(\overline{u'}).B/Y\}\{\overline{U'}/\overline{u'}\},\,(t \cup (\Gamma \vdash A <: (\mathbf{rec}\;Y(\overline{u'}).B)[\overline{U'}]))\,)$ /* (EQ:REC) */

(21) $\mathtt{sbt}(\,\Gamma,\,A,\,B \oplus C,\,t\,) \triangleq \mathtt{sbt}(\,\Gamma,\,A,\,B,\,t\,) \vee \mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,)$ /* (ST:ALTERNATIVE) */

(22) $\mathtt{sbt}(\,\Gamma,\,A \oplus B,\,C \oplus D,\,t\,) \triangleq$
$(\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,)) \vee$
$(\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,))$ /* (ST:ALTERNATIVE-CONG) */

(23) $\mathtt{sbt}(\,\Gamma,\,A\&B,\,C,\,t\,) \triangleq \mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \vee \mathtt{sbt}(\,\Gamma,\,B,\,C,\,t\,)$ /* (ST:INTERSECTION) */

(24) $\mathtt{sbt}(\,\Gamma,\,A\&B,\,C\&D,\,t\,) \triangleq$
$(\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,)) \vee$
$(\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,))$ /* (ST:INTERSECTION-CONG) */

(25) $\mathtt{sbt}(\,\Gamma,\,\sum_i^n \mathtt{t}_i\#A_i,\,\sum_i^m \mathtt{t}_i\#B_i,\,t\,) \triangleq n \leq m \wedge \bigwedge_j^n \mathtt{sbt}(\,\Gamma,\,A_j,\,B_j,\,t\,)$ /* (ST:SUM) */

(26) $\mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A},\mathtt{f}' : A'],\,[\overline{\mathtt{f} : B}],\,t\,) \triangleq$
$[\overline{\mathtt{f} : A}] \neq \emptyset \wedge \mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A}],\,[\overline{\mathtt{f} : B}],\,t\,)$ /* (ST:DISCARD) */

(27) $\mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A},\mathtt{f}' : A'],\,[\overline{\mathtt{f} : B},\mathtt{f}' : B'],\,t\,) \triangleq$
$\mathtt{sbt}(\,\Gamma,\,A',\,B',\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A}],\,[\overline{\mathtt{f} : B}],\,t\,)$ /* (ST:RECORD) */

(28) $\mathtt{sbt}(\,\Gamma,\,A,\,\exists l.B,\,t\,) \triangleq \mathtt{sbt}(\,\Gamma,\,A,\,B\{p/l\},\,t\,)$ /* (ST:PACKLOC) */

(29) $\mathtt{sbt}(\,\Gamma,\,\forall l.A,\,B,\,t\,) \triangleq \mathtt{sbt}(\,(\Gamma, l : \mathbf{loc}),\,A\{p/l\},\,B,\,t\,)$ /* (ST:LOCAPP) */

(30) $\mathtt{sbt}(\,\Gamma,\,A,\,\exists X <: B.C,\,t\,) \triangleq \mathtt{sbt}(\,\Gamma,\,A,\,C\{D/X\},\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,D,\,B,\,t\,)$ /* (ST:PACKTYPE) */

(31) $\mathtt{sbt}(\,\Gamma,\,\forall X <: A.B,\,C,\,t\,) \triangleq$
$\mathtt{sbt}(\,(\Gamma, X <: A, X : \mathbf{type}),\,B\{D/X\},\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,D,\,A,\,t\,)$ /* (ST:TYPEAPP) */

Figure 5.7: Subtyping Algorithm (2/2).

through (EQ:REC). The remaining rules are essentially directed by the structure of the type, which enables our algorithm to simply interpret the subtyping rules from conclusion to premise, without needing changes to the rules shown in the formal system of prior chapters. Our use of weakening to close the co-inductive proof, useful to close subtyping of types quantified with bounds, follows ideas proposed in [81] but extended to include recursive types with parameters.

We state that `sbt` follows analogous properties to the protocol composition algorithm:

Theorem 13 (`sbt` soundness)**.** *If* `sbt`(Γ, A, B) *then* $\Gamma \vdash A <: B$.

Theorem 14 (`sbt` completeness)**.** *If* $\Gamma \vdash A <: B$ *then* `sbt`(Γ, A, B).

Theorem 15 (`sbt` decidability)**.** *Given well-formed types, A and B, and a well-formed environment,* Γ*,* `sbt`(Γ, A, B) *terminates.*

5.4 Notes on the Full Language Implementation

We implemented the full systems of chapter 2[1] and chapter 3[2], and both algorithms of this chapter[3]. Although our two initial prototypes implement the full language (but without any supporting proofs of any of its algorithms), our focus here is only on discussing the central components of our interference-control system. We believe that these two components, protocol composition and subtyping, constitute the main algorithmic challenges of a full implementation. The remaining aspects can be made decidable (even if not complete) through additional type annotations that are consistent with the rest of the language—adding annotations that make the system practically syntax-directly, similar to what was done with the two full language prototypes. Finally, we assume type well-formedness although a complete implementation of the typechecker would (easily) verify this assumption as necessary.

Perhaps a more interesting approach would be to exploit the employed local reasoning, and use that locality to infer types by extracting these types from their local usages. Otherwise, avoiding the extra type annotations needed would require a "search" algorithm that attempts different typing derivations when typing fails, while bounding the domain of types that can be attempted to ensure decidability even if invalidating completeness. However, we believe it would be more interesting to attempt to extract types, including protocols, from a program by exploiting the way in which protocols directly express the local usages of shared state. In that situation, the language would preserve most of its succinctness of chapter 4, while still relying on the two algorithms shown here. We leave as future work the exploration of this kind of type inference and implementation.

[1] http://www.cs.cmu.edu/~foliveir/dead-parrot.html
[2] http://www.cs.cmu.edu/~foliveir/deaf-parrot.html
[3] http://www.cs.cmu.edu/~foliveir/protocol-composition.html

Chapter 6

Conclusions

We conclude this dissertation by discussing possible future work, and briefly summarizing the presented work. Section 6.1 discusses possible future work aimed at addressing open problems of our design, and discusses other interesting relevant directions that can be taken towards more practical, precise, and powerful verification of shared mutable state. Finally, Section 6.2 revisits the thesis statement and contributions, by summarizing the work shown in previous chapters.

6.1 Future Work

We now discuss some open problems of the proposed approach and possible future directions.

Type Expressiveness: Several improvements related to the expressiveness of our types and state abstractions can be pursued in future work.

For instance, as done in [57, 69, 71], we can offer more precise state abstractions that go beyond typestate-like abstractions. This will enable, in some situations, more precise tracking of state changes, reducing somewhat conservative reasoning and syntactical overhead that our system currently requires in those situations.

As shown in Section 2.4.1, our stack object is limited to only two typestates, `Empty` and `NonEmpty`. Although we can insert three elements in succession, sequentially removing those three elements will still require a conservative `isEmpty` check to rule-out the `Empty` typestate when using `pop`. If instead the state abstraction is able to capture some finer typing, such as "`NonEmpty`$(1, 2, 3)$" (were 1, 2, and 3 were the inserted elements), checking such a program would not require the conservative check (even if the precise values contained may need to be abstracted in some cases). See Figure 6.1 for an illustration of the impact such a change would have on this particular example. From the usability stand point, such a change is quite import in reducing the perception that programmers may get of "the type system getting in the way" of their coding/reasoning, and the potential need for unsafe workarounds.

Other possible future directions include support for read-only capabilities, and the full range of access permissions (such as enabling type annotations for "deep" immutability over the full structure of a type rather than just our current "shallow" immutability, etc.); object-

Current:

```
let s = newStack {} in                    Empty
  s.push(1);                              NonEmpty
  s.push(2);                              NonEmpty
  s.push(3);                              NonEmpty
  s.pop();                        NonEmpty ⊕ Empty
  case s.isEmpty() of
    Empty#_ → {}
  | NonEmpty#_ →                          NonEmpty
    s.pop();                      NonEmpty ⊕ Empty
    ...
  end
```

Using more refined abstractions:

```
let s = newStack {} in                    Empty
  s.push(1);                              NonEmpty(1)
  s.push(2);                              NonEmpty(1,2)
  s.push(3);                              NonEmpty(1,2,3)
  s.pop();                                NonEmpty(1,2)
  s.pop();                                NonEmpty(1)
  s.pop()                                 Empty
```

Figure 6.1: Possible comparison to a system with more precise typing.

orientation features (dynamic dispatch, inheritance, etc.); and perhaps more dynamic ownership recovery methods (such as supporting features similar to "automatic reference counting", gradual approaches [97], or via idioms on top of a more precise type system with support for type refinements [42]).

Flexible Avoidance of Re-Entrant/Deadlocking Uses: Because shared state exploits the notion of fictional disjointness/separation, this means that whenever we have a type such as "$A * B$" we are not able to know precisely if A and B do or not overlap. Thus, $A * B$ means both that A and B are separate, and that A and B may be treated as separate but actually share state. Therefore, if A is undergoing private uses, then uses of B may or may not need to be concealed to avoid inspection of private states. In chapter 3, we saw a "simple" mechanism to avoid such re-entrancy. However, chapter 4 does not use such restriction since avoiding re-entrancy is not sufficient to ensure a strong global progress property, as inconsistent lock acquisition may cause deadlocks.

Perhaps more interestingly, the challenge of this reasoning appears to arise from the loss of knowledge caused by (non-strictly contained) uses of fictional separation. Instead if we enable types to express which other states may overlap, we may be able to offer a "region-like" semantics that will enable the type system to precisely know which types may or may not refer the same overlapping part of the heap—even if potentially conservative in some situations.

Somewhat sketchy reasoning follows: if we split A into B and C, we would get a type $B \circledast C$ to mean that B and C are not separate, i.e. $\neg(B * C)$. Focus (or locking) over a $\circledast$ type would make the other type unavailable until unfocus (or unlock), analogously to the frame rule but not lexically-scoped. Notions of fictional separation are simply a coercion of the $*$ type, so that we have:

$$A * B <: A \circledast B$$

To mean that whenever we know that A and B are separate, then they can also be safely used as if they were sharing state. More interesting is the use of both $*$ and $\circledast$, so that a type

such as: $A * (B \circledast C)$ would mean that A is separate from B and C, but B and C may overlap with each other. Future work may perhaps address issues of distributivity of such typing, how case analysis may proceed to distinguish between the different kinds of separation, and whether such a verification can be implemented in a decidable system—besides considering how to extend such a reasoning to avoid deadlocks when the concurrent setting is considered.

Perhaps such a feature could be useful to better model circular data structures. Although our use of "alias type"-like types enables pointers to refer to themselves, thus creating circular data structures, there are limitations on how those data structures can be typed.

For instance, consider a circular linked list. Conceptually, the list makes a complete cycle when iteration reaches a memory location that was already visited. However, our types and type system are not capable of expressing and exploiting equality between locations, meaning that at most they must rely on some form of tag to imply that we reached the "pivot" of the circular list.

Still, this use of a tagged value limits how the data structure may be interpreted. While without any explicit tagging we could reinterpret the structure by picking any of its elements as the initial entry (i.e. the pivot), with an explicit tag this is no longer possible as the pivot is fixed as the same to all aliases. This limitation impacts the expressiveness on some uses of circular data structures, such as circular linked lists and doubly linked lists.

Low-level, Safe Modeling of other Concurrency Frameworks: Our work focused on the use of mutual exclusion through lock-based constructs. Locks enable mutual exclusion to be held over an arbitrary length of time, with the danger of causing deadlocks. Alternatively, by constraining the length of mutual exclusion to a single "compare-and-swap" primitive, the danger of deadlock disappears since it is not possible to "lock"' other cells within the "compare-and-swap" primitive. However, both approaches may live-lock if the system reaches a state where it never does useful progress.

It remains an open problem to model both kinds of mutual exclusion within a single, unified typing framework. This unification would perhaps enable compilers to pick which mutual exclusion primitive is better suited to a particular use, rather than rely on the programmer's manual decision. Or, similarly, enable the compiler to decide on whether the state is shared but only used locally by a single thread and forfeit the need for run-time mutual exclusion mechanisms. Finally, providing some guarantees towards live-lock avoidance would also be an interesting future direction.

Interestingly, such a system would also enable the safe, low-level composition of primitive mutual-exclusion blocks enabling the safe construction of more complex concurrency frameworks (such as *software transactional memory* [82] or *reagents* [91]) as either idioms or type abstractions on top of that core system, somewhat similarly to how we showed that message-passing can be handled within our protocol framework. This would enable the safe interaction of a wide-range of programmer supplied concurrency abstractions, in order to let the programmer pick the one which better suits their needs—while ensuring safety regardless of their choice or how they are combined. Indeed, since most higher-level concurrency

frameworks/constructs are implemented through the use of either locks or CAS primitives, there should be a way to reason uniformly about such systems through their low-level composition of locking/CAS primitives, while providing safe guarantees of progress and memory safety.

Thus, possible future work would look into how to model the unifying, underlying concepts of composing such primitives in a safe way, enabling the construction of larger abstractions that yet interact safely with other concurrency frameworks.

User-level features, type inference, user study: We did not consider concerns on how to address more user-level language features. For instance, defining type inference algorithms to reduce the syntactic overhead of the language; or do a large case analysis to study the suitability of our abstractions and perhaps find expressiveness gaps.

Finally, we did not explore the problem of providing an efficient compiler-level implementation (such as efficient memory layout, instruction generation, etc.). We leave as future work exploring these compiler-level issues, perhaps by following the work of more recent languages such as Rust [2] that provides a simpler form of memory safety using linear types and borrowing, or Go [1] that encourages sharing memory via message-passing communication over channels that exchange a fixed/invariant type. However, by using our (more flexible) model we should be able to provide stronger (and proven) safety guarantees together with more solid theoretical foundations.

6.2 Summary

This thesis proposes *rely-guarantee protocols* as a modular, composable, expressive, and automatically verifiable mechanism to control the interference resulting from the interaction of non-local aliases that share access to overlapping mutable state.

In chapter 2, we introduce our core language that was used as basis to build our protocol framework. Both chapters 3 and 4 discuss how our sharing mechanism works in the sequential and concurrent setting, respectively. Each of those chapters discusses the three core mechanisms of our technique: how protocols can be specified, used, and composed. The algorithmic aspects of safe protocol composition, are at the core of chapter 5 and the main, central component of the system's decidability claim. While the rest of the language can require annotations consistent with the intended language use case, we showed that protocol composition is decidable on its own.

Bibliography

[1] Go language. https://golang.org/. 6.1

[2] Rust language. https://www.rust-lang.org/. 6.1

[3] Fatal radiation dose in therapy attributed to computer mistake. http://www.nytimes.com/1986/06/21/us/fatal-radiation-dose-in-therapy-attributed-to-computer-mistake.html, June 21, 1986. 1

[4] Toyota unintended acceleration and the big bowl of 'spaghetti' code. http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-%E2%80%9Cspaghetti%E2%80%9D-code, November 7, 2013. 1

[5] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, December 2007. ISSN 0169-2968. URL http://dl.acm.org/citation.cfm?id=1365997.1366003. 1.1, 2, 2.2, 2.2.2, 2.6, 3.6, 4.6

[6] Jonathan Aldrich. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 101–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2514738. URL http://doi.acm.org/10.1145/2509578.2514738. 1.1

[7] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640073. URL http://doi.acm.org/10.1145/1639950.1640073. 2.6

[8] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 104–118, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. doi: 10.1145/99583.99600. URL http://doi.acm.org/10.1145/99583.99600. 5.3

[9] Ron Avitzur. The graphing calculator story. https://www.youtube.com/watch?v=GMyg5ohTsVY#t=2520, August 1, 2006. 1

[10] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-*

oriented Programming Systems Languages and Applications, OOPSLA '08, pages 227–244, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449783. URL http://doi.acm.org/10.1145/1449764.1449783. 2.4, 2.6, 3.6, 4.6

[11] Nels E. Beckman, Yoon Phil Kim, Sven Stork, and Jonathan Aldrich. Reducing stm overhead with access permissions. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, IWACO '09, pages 2:1–2:10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-546-8. doi: 10.1145/1562154.1562156. URL http://doi.acm.org/10.1145/1562154.1562156. 2.6, 3.6, 4.6

[12] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 2–26, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0. URL http://dl.acm.org/citation.cfm?id=2032497.2032501. 2, 2.4

[13] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297050. URL http://doi.acm.org/10.1145/1297027.1297050. 2, 2.4, 2.4.2, 2.4.2, 2.6, 3.5.2, 3.6

[14] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical api protocol checking with access permissions. In Sophia Drossopoulou, editor, *ECOOP 2009 Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 195–219. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_10. URL http://dx.doi.org/10.1007/978-3-642-03013-0_10. 2.4

[15] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Checking concurrent typestate with access permissions in plural: A retrospective. In Peri L. Tarr and Alexander L. Wolf, editors, *Engineering of Software*, pages 35–48. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19822-9. doi: 10.1007/978-3-642-19823-6_4. URL http://dx.doi.org/10.1007/978-3-642-19823-6_4. 2.6, 3.6

[16] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040327. URL http://doi.acm.org/10.1145/1040305.1040327. 2.6, 3.6

[17] John Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, May 2001. ISSN 0038-0644. doi: 10.1002/spe.370. URL http://dx.doi.org/10.1002/spe.370. 2.4.2, 2.6, 3.6

[18] John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40325-6. URL http://dl.acm.org/citation.cfm?id=1760267.1760273. 2.2, 2.6, 3.6

[19] John Boyland, James Noble, and William Retert. Capabilities for sharing. In JrgenLind-

skov Knudsen, editor, *ECOOP 2001 Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42206-8. doi: 10.1007/3-540-45337-7_2. URL http://dx.doi.org/10.1007/3-540-45337-7_2. 2.6, 3.6

[20] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 283–295, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040329. URL http://doi.acm.org/10.1145/1040305.1040329. 2.6

[21] Luís Caires. Spatial-behavioral types, distributed services, and resources. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *Trustworthy Global Computing*, volume 4661 of *Lecture Notes in Computer Science*, pages 98–115. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75333-9. doi: 10.1007/978-3-540-75336-0_7. URL http://dx.doi.org/10.1007/978-3-540-75336-0_7. 2.5, 2.5.1, 2.6

[22] Luís Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008. doi: 10.1016/j.tcs.2008.04.030. URL http://dx.doi.org/10.1016/j.tcs.2008.04.030. 2.5, 2.5.1, 2.6

[23] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and Franois Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15374-7. doi: 10.1007/978-3-642-15375-4_16. URL http://dx.doi.org/10.1007/978-3-642-15375-4_16. 4.6

[24] Luís Caires and João C. Seco. The type discipline of behavioral separation. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 275–286, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429103. URL http://doi.acm.org/10.1145/2429069.2429103. 2, 2.5, 2.5.1, 2.5.1, 2.5.1, 2.5.1, 2.6, 3.6, 4.6

[25] Luís Caires and Hugo Torres Vieira. Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440, December 2010. ISSN 0304-3975. doi: 10.1016/j.tcs.2010.09.010. URL http://dx.doi.org/10.1016/j.tcs.2010.09.010. 4.5.3

[26] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2908-9. doi: 10.1109/LICS.2007.30. URL http://dx.doi.org/10.1109/LICS.2007.30. 2.1.3, 2.4.2

[27] John Carmack. In-depth: Functional programming in c++. http://gamasutra.com/view/news/169296/Indepth_Functional_programming_in_C.php, April 30, 2012. 1.1

[28] Elias Castegren and Tobias Wrigstad. Capable: Capabilities for scalability. In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*,

IWACO '14, 2014. URL http://www.ownership-types.org/iwaco14/accepted_files/Paper2.pdf. 2.6

[29] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 213–224, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411235. URL http://doi.acm.org/10.1145/1411204.1411235. 2.2.4

[30] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 262–275, New York, NY, USA, 1999. ACM. ISBN 1-58113-095-3. doi: 10.1145/292540.292564. URL http://doi.acm.org/10.1145/292540.292564. 2, 2.6

[31] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 198–208, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351259. URL http://doi.acm.org/10.1145/351240.351259. 2.4.1

[32] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 59–69, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: 10.1145/378795.378811. URL http://doi.acm.org/10.1145/378795.378811. 2.4, 2.6

[33] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22159-3. doi: 10.1007/978-3-540-24851-4_21. URL http://dx.doi.org/10.1007/978-3-540-24851-4_21. 2.4, 2.6, 3.6

[34] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL http://dl.acm.org/citation.cfm?id=1883978.1884012. 1.2, 4, 3.2

[35] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, MatthewJ. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo DHondt, editor, *ECOOP 2010 Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_24. URL http://dx.doi.org/10.1007/978-3-642-14107-2_24. 3.6

[36] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, POPL '13, pages 287–300, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429104. URL http://doi.acm.org/10.1145/2429069.2429104. 1.2, 3.6

[37] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3. doi: 10.1007/978-3-642-00590-9_26. URL http://dx.doi.org/10.1007/978-3-642-00590-9_26. 3.6

[38] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In AndrewD. Gordon, editor, *Foundations of Software Science and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 250–266. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-00897-2. doi: 10.1007/3-540-36576-1_16. URL http://dx.doi.org/10.1007/3-540-36576-1_16. 2.4.1

[39] Manuel Fähndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512532. URL http://doi.acm.org/10.1145/512529.512532. 2.6, 3.3, 3.6

[40] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestate. In *Proceedings of the first international workshop on alias confinement and ownership*, IWACO 03, July 2003. 2.4, 3.5.1, 3.6

[41] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480922. URL http://doi.acm.org/10.1145/1480881.1480922. 3.6

[42] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113468. URL http://doi.acm.org/10.1145/113445.113468. 6.1

[43] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 299–312, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706335. URL http://doi.acm.org/10.1145/1706299.1706335. 2, 2.5.1, 2.6

[44] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319848. URL http://doi.acm.org/10.1145/319838.319848. 2.1.3

[45] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. 1.1, 2.1.2, 2.6, 3.6

[46] James Gleick. Little bug big bang. http://www.nytimes.com/1996/12/01/magazine/little-bug-big-bang.html, December 1, 1996. 1

[47] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Static lock capabilities for deadlock freedom. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 67–78, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103796. URL http://doi.acm.org/10.1145/2103786.2103796. 4.5.2

[48] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 21–40, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384619. URL http://doi.acm.org/10.1145/2384616.2384619. 4.6

[49] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 73–84, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462160. URL http://doi.acm.org/10.1145/2491956.2462160. 3.1, 3, 3.6, 4.2.1

[50] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. URL http://dl.acm.org/citation.cfm?id=1883978.1884002. 2.4.2

[51] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10): 576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259. 3.6

[52] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, April 1992. ISSN 1055-6400. doi: 10.1145/130943.130947. URL http://doi.acm.org/10.1145/130943.130947. 2.6

[53] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin Heidelberg, 1993. ISBN 978-3-540-57208-4. doi: 10.1007/3-540-57208-2_35. URL http://dx.doi.org/10.1007/3-540-57208-2_35. 4.6

[54] Jonas Braband Jensen and Lars Birkedal. Fictional separation logic. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 377–396, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28868-5. doi: 10.1007/978-3-642-28869-2_19. URL http://dx.doi.org/10.1007/978-3-642-28869-2_19. 1.2, 4, 3.2, 3.6

[55] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM*

Trans. Program. Lang. Syst., 5(4):596–619, October 1983. ISSN 0164-0925. doi: 10.1145/69575.69577. URL http://doi.acm.org/10.1145/69575.69577. 1.1, 3.1, 3.6, 4.6

[56] Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 41–54, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364536. URL http://doi.acm.org/10.1145/2364527.2364536. 1.2, 3, 4, 3.2, 3.5.2, 3.6, 4.6

[57] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating linear and dependent types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 17–30, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676969. URL http://doi.acm.org/10.1145/2676726.2676969. 2.6, 6.1

[58] K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-00589-3. doi: 10.1007/978-3-642-00590-9_27. URL http://dx.doi.org/10.1007/978-3-642-00590-9_27. 3.6

[59] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 213–225, New York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: 10.1145/944705.944725. URL http://doi.acm.org/10.1145/944705.944725. 2, 2.6

[60] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in system f°. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 77–88, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-891-9. doi: 10.1145/1708016.1708027. URL http://doi.acm.org/10.1145/1708016.1708027. 2.6

[61] Filipe Militão and Luís Caires. An exception aware behavioral type system for object-oriented programs. In *Proceedings of INFORUM 2009 - Simpósio de Informática*. Faculdade de Ciências - Universidade de Lisboa, 2009. 2.5, 2.5.1, 2.6

[62] Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing control with view-based typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 7:1–7:7, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0540-2. doi: 10.1145/1924520.1924527. URL http://doi.acm.org/10.1145/1924520.1924527. 2.6

[63] Filipe Militão, Jonathan Aldrich, and Luís Caires. Substructural typestates. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification*, PLPV '14, pages 15–26, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2567-7. doi: 10.1145/2541568.2541574. URL http://doi.acm.org/10.1145/2541568.2541574. 2, 2.1.2, 2.2.5, 4.3.2, A

[64] Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-guarantee protocols. In Richard Jones, editor, *ECOOP 2014 Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 334–359. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44201-2. doi: 10.1007/978-3-662-44202-9_14. URL http://dx.doi.org/10.1007/978-3-662-44202-9_14. 1.1, 1.3.3, 1.4, 2.1.2, 3, A

[65] Filipe Militão. Design and implementation of a behaviorally typed programming system for web services. Master's thesis, Universidade Nova de Lisboa, July 2008. 2.5, 2.5.1, 2.6

[66] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90008-4. URL http://dx.doi.org/10.1016/0890-5401(92)90008-4. 4.5.1, 4.6

[67] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 557–570, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103722. URL http://doi.acm.org/10.1145/2103656.2103722. 2.4.2, 2.6, 3.6

[68] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 62–73, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: 10.1145/1159803.1159812. URL http://doi.acm.org/10.1145/1159803.1159812. 2.6

[69] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in hoare type theory. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 189–204, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71314-2. URL http://dl.acm.org/citation.cfm?id=1762174.1762194. 2.6, 6.1

[70] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 1–15, New York, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: 10.1145/165854.167976. URL http://doi.acm.org/10.1145/165854.167976. 2.6

[71] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040326. URL http://doi.acm.org/10.1145/1040305.1040326. 2.6, 3.6, 6.1

[72] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: 10.1145/237721.237794. URL http://doi.acm.org/10.1145/237721.237794. 4.4.1

[73] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA,

2002. ISBN 0-262-16209-1. 4.3.4, 5

[74] Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '11, pages 73–86, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0484-9. doi: 10.1145/1929553.1929565. URL http://doi.acm.org/10.1145/1929553.1929565. 3.5.1, 3.6, 4.2.1

[75] François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *J. Funct. Program.*, 23(1):38–144, January 2013. ISSN 0956-7968. doi: 10.1017/S0956796812000366. URL http://dx.doi.org/10.1017/S0956796812000366. 2.5

[76] François Pottier and Jonathan Protzenko. Programming with permissions in mezzo. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 173–184, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500598. URL http://doi.acm.org/10.1145/2500365.2500598. 3.6

[77] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 53–65, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480890. URL http://doi.acm.org/10.1145/1480881.1480890. 2.6

[78] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL http://dl.acm.org/citation.cfm?id=645683.664578. 2.1.3, 2.4.2, 3.6

[79] JohnC. Reynolds. Syntactic control of interference part 2. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and SimonettaRonchi Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722. Springer Berlin Heidelberg, 1989. ISBN 978-3-540-51371-1. doi: 10.1007/BFb0035793. URL http://dx.doi.org/10.1007/BFb0035793. 2.1

[80] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, LFP '92, pages 288–298, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: 10.1145/141471.141563. URL http://doi.acm.org/10.1145/141471.141563. 2.1

[81] João Costa Seco and Luís Caires. Subtyping first-class polymorphic components. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25435-5. doi: 10.1007/978-3-540-31987-0_24. URL http://dx.doi.org/10.1007/978-3-540-31987-0_24. 5.3

[82] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.

224987. URL http://doi.acm.org/10.1145/224964.224987. 6.1

[83] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 366–381, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67262-1. URL http://dl.acm.org/citation.cfm?id=645394.651903. 2.4.2, 2.6

[84] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6312929. URL http://dx.doi.org/10.1109/TSE.1986.6312929. 1.1, 2, 2.4, 2.4.2, 2.6

[85] Robert E. Strom. Mechanisms for compile-time enforcement of security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 276–284, New York, NY, USA, 1983. ACM. ISBN 0-89791-090-7. doi: 10.1145/567067.567093. URL http://doi.acm.org/10.1145/567067.567093. 1.1, 2, 2.4, 2.4.2, 2.6

[86] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048122. URL http://doi.acm.org/10.1145/2048066.2048122. 2.4.2, 2.6

[87] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In *Proceedings of the 22nd European conference on Programming Languages and Systems*, ESOP'13, pages 169–188, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37035-9. doi: 10.1007/978-3-642-37036-6_11. URL http://dx.doi.org/10.1007/978-3-642-37036-6_11. 3.6

[88] Tim Sweeney. The next mainstream programming language: A game developer's perspective. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 269–269, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111061. URL http://doi.acm.org/10.1145/1111037.1111061. 1.1

[89] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '94, pages 398–413, London, UK, UK, 1994. Springer-Verlag. ISBN 3-540-58184-7. URL http://dl.acm.org/citation.cfm?id=646423.691947. 2.6

[90] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 447–458, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926436. URL http://doi.acm.org/10.1145/1926385.1926436. 2.6, 3.6, 4.6

[91] Aaron Turon. Reagents: Expressing and composing fine-grained concurrency. In *Proceed-*

ings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, pages 157–168, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254084. URL http://doi.acm.org/10.1145/2254064.2254084. 6.1

[92] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74406-1. doi: 10.1007/978-3-540-74407-8_18. URL http://dx.doi.org/10.1007/978-3-540-74407-8_18. 3.6

[93] J. Van Den Bos and C. Laffra. Procol: A parallel object language with protocols. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 95–102, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. doi: 10.1145/74877.74888. URL http://doi.acm.org/10.1145/74877.74888. 2.6

[94] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990. 2.6

[95] David Walker and J. Gregory Morrisett. Alias types for recursive data structures. In *Selected papers from the Third International Workshop on Types in Compilation*, TIC '00, pages 177–206, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42196-3. URL http://dl.acm.org/citation.cfm?id=647229.719257. 2.6

[96] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, July 2000. ISSN 0164-0925. doi: 10.1145/363911.363923. URL http://doi.acm.org/10.1145/363911.363923. 2.6

[97] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mira Mezini, editor, *ECOOP 2011 Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22654-0. doi: 10.1007/978-3-642-22655-7_22. URL http://dx.doi.org/10.1007/978-3-642-22655-7_22. 3.6, 6.1

[98] Greta Yorsh, Alexey Skidanov, Thomas Reps, and Mooly Sagiv. Automatic assume/guarantee reasoning for heap-manipulating programs. *Electron. Notes Theor. Comput. Sci.*, 131:125–138, May 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.01.028. URL http://dx.doi.org/10.1016/j.entcs.2005.01.028. 3.1

Appendix A

Proofs

We now show the main proofs of the system of Chapter 4, and the necessary auxiliary definitions. For the proofs of the systems of Chapters 2 and 3, we refer interested readers to [63] and [64], respectively. However, our presentation of that work has slight modifications that are either straightforward to adapt from those proofs, or is shown in more detail in this Chapter (but further below).

A.1 Auxiliary Definitions

A.1.1 Well-Formed Types and Environments

Well-formed conditions are not explicitly mentioned, but are assumed to be present whenever they are relevant.

Definition 2 (Well-Formedness)**.** We have the following cases (defined by induction on the structure of the type/environment):

- $\boxed{\Gamma\ \mathbf{wf}}$ **(Gamma)**

$$\frac{}{\cdot\ \mathbf{wf}} \qquad \frac{\Gamma\ \mathbf{wf}}{\Gamma, p : \mathbf{loc}\ \mathbf{wf}} \qquad \frac{\Gamma\ \mathbf{wf} \qquad \Gamma \vdash A\ \mathbf{type}}{\Gamma, X : \mathbf{type}, X <: A\ \mathbf{wf}} \qquad \frac{\Gamma\ \mathbf{wf} \qquad \Gamma \vdash A\ \mathbf{type}}{\Gamma, x : A\ \mathbf{wf}}$$

- $\boxed{\Gamma \vdash \Delta\ \mathbf{wf}}$ **(Delta)**

$$\frac{}{\Gamma \vdash \cdot\ \mathbf{wf}} \qquad \frac{\Gamma \vdash \Delta\ \mathbf{wf} \qquad \Gamma \vdash A\ \mathbf{type}}{\Gamma \vdash \Delta, x : A\ \mathbf{wf}} \qquad \frac{\Gamma \vdash \Delta\ \mathbf{wf} \qquad \Gamma \vdash A\ \mathbf{type}}{\Gamma \vdash \Delta, A\ \mathbf{wf}}$$

- $\boxed{\Gamma \vdash p\ \mathbf{loc}}$ **(Location)**

$$\frac{}{\Gamma, p : \mathbf{loc} \vdash p\ \mathbf{loc}}$$

- $\boxed{\Gamma \vdash A\ \mathbf{type}}$ **(Type)**

$$\frac{}{\Gamma \vdash \mathbf{none}\ \mathbf{type}} \qquad \frac{\Gamma \vdash A\ \mathbf{type}}{\Gamma \vdash\ !A\ \mathbf{type}} \qquad \frac{\overline{\Gamma \vdash A_i\ \mathbf{type}}}{\Gamma \vdash [\overline{\mathtt{f} : A}]\ \mathbf{type}} \qquad \frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash A_0 \multimap A_1\ \mathbf{type}}$$

$$\frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash\ A_0 :: A_1\ \ \mathbf{type}} \qquad \frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash\ A_0 * A_1\ \ \mathbf{type}}$$

$$\frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash A_0 \oplus A_1\ \mathbf{type}} \qquad \frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash A_0 \& A_1\ \mathbf{type}}$$

$$\frac{\begin{array}{c}\Gamma \vdash p\ \ \mathbf{loc} \\ \Gamma \vdash A\ \mathbf{type}\end{array}}{\Gamma \vdash\ \mathbf{rw}\ p\ A\ \ \mathbf{type}} \qquad \frac{\Gamma \vdash p\ \ \mathbf{loc}}{\Gamma \vdash\ \mathbf{ref}\ p\ \ \mathbf{type}} \qquad \frac{\Gamma, l : \mathbf{loc} \vdash A\ \mathbf{type}}{\Gamma \vdash \forall l.A\ \mathbf{type}} \qquad \frac{\Gamma, l : \mathbf{loc} \vdash A\ \mathbf{type}}{\Gamma \vdash \exists l.A\ \mathbf{type}}$$

$$\frac{\cdot \vdash A_0\ \mathbf{type} \qquad \Gamma, X : \mathbf{type}, X <: A_0 \vdash A_1\ \mathbf{type}}{\Gamma \vdash \forall X <: A_0.A_1\ \mathbf{type}}$$

$$\frac{\cdot \vdash A_0\ \mathbf{type} \qquad \Gamma, X : \mathbf{type}, X <: A_0 \vdash A_1\ \mathbf{type}}{\Gamma \vdash \exists X <: A_0.A_1\ \mathbf{type}}$$

$$\frac{\overline{\Gamma \vdash A_i\ \mathbf{type}}}{\Gamma \vdash \sum_i \mathtt{t}_i \# A_i\ \mathbf{type}} \qquad \frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash A_0; A_1\ \mathbf{type}}$$

$$\frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type} \qquad \mathbf{locs}(A_0) = \mathbf{locs}(A_1) \neq \emptyset}{\Gamma \vdash A_0 \Rightarrow A_1\ \mathbf{type}}$$

$$\frac{\begin{array}{c}u_0 : k_0, ..., u_n : k_n, X : k_0 \to ... \to k_n \to \mathbf{type} \vdash A\ \mathbf{type} \\ \Gamma \vdash U_i\ k_i \qquad k_i = \mathtt{kind}(u_i) \qquad i \in \{0, ..., n\}\end{array}}{\Gamma \vdash (\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]\ \mathbf{type}}$$

$$\frac{(X : k_0 \to ... \to k_n \to \mathbf{type}) \in \Gamma \qquad (U_i : k_i) \in \Gamma \qquad i \in \{0, ..., n\}}{\Gamma \vdash X[\overline{U}]\ \mathbf{type}}$$

where:

$$\begin{array}{lcl}\mathtt{kind}(l) & = & \mathbf{loc} \\ \mathtt{kind}(X) & = & \mathbf{type}\end{array}$$

A.1.2 Set of Locations of a Type

Definition 3 (Locations of a Type)**.**

$$
\begin{array}{lcl}
\mathbf{locs}(\mathbf{rw}\ p\ A) & = & \{p\} \\
\mathbf{locs}(A_0 * A_1) & = & \mathbf{locs}(A_0) \cup \mathbf{locs}(A_1) \\
\mathbf{locs}(A_0 \& A_1) & = & \mathbf{locs}(A_0) \cup \mathbf{locs}(A_1) \\
\mathbf{locs}(A_0 \oplus A_1) & = & \mathbf{locs}(A_0) \cup \mathbf{locs}(A_1) \\
\mathbf{locs}(\exists l.A) & = & \mathbf{locs}(A) \setminus l \\
\mathbf{locs}(\forall l.A) & = & \mathbf{locs}(A) \setminus l \\
\mathbf{locs}(\exists X <: A_0.A_1) & = & \mathbf{locs}(A_1) \\
\mathbf{locs}(\forall X <: A_0.A_1) & = & \mathbf{locs}(A_1) \\
\mathbf{locs}(A_0 \Rightarrow A_1) & = & \mathbf{locs}(A_0) \\
\mathbf{locs}(A_0; A_1) & = & \mathbf{locs}(A_0) \\
\mathbf{locs}(\mathbf{none}) & = & \emptyset \\
\mathbf{locs}(P) & = & \emptyset \\
\mathbf{locs}(X[\overline{U}]) & = & \emptyset \\
\mathbf{locs}((\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]) & = & \mathbf{locs}(A\{\mathbf{rec}\ X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\})
\end{array}
$$

If the type is a protocol, we do not yield a location since we will have to lock that protocol's locations separately (i.e. locking is "shallow"). Recursive types are assumed to be non-bottom so that there is a finite number of unfolds that are relevant to extract the set of locations. Finally, recall that any type bound that is directly accessible must be a protocol so that types of the form: "$\forall X <: (\exists p.(\mathbf{rw}\ p\ A)).X$" are forbidden from occurring here.

A.1.3 Store Typing

See Figure 4.15.

A.1.4 Substitution

Definition 4 (Substitution)**.** For clarity, we define substitution on constructs using e even though the grammar will restrict these "expression" to be values (v) in some of those places. This is done just for readability purposes to make it clear which value is being used for the substitution, and where it is being substituted into.

1. **Variable Substitution**, (vs:*)
   We define the usual capture-avoiding (i.e. up to renaming of bounded variables) substitution rules:

$\boxed{e_0\{v/x\} = e_1}$

$$
\begin{array}{lrcll}
\text{(vs:1)} & \rho\{v/x\} & = & \rho & \\
\text{(vs:2)} & x\{v/x\} & = & v & \\
\text{(vs:3)} & x_0\{v/x_1\} & = & x_0 & (x_0 \neq x_1) \\
\text{(vs:4)} & (\lambda x_0.e_0)\{v/x_1\} & = & \lambda x_0.e_0\{v/x_1\} & (x_0 \neq x_1) \\
\text{(vs:5)} & \{\overline{\mathtt{f} = e}\}\{v/x\} & = & \{\overline{\mathtt{f} = e\{v/x\}}\} & \\
\text{(vs:6)} & (e.\mathtt{f})\{v/x\} & = & e\{v/x\}.\mathtt{f} & \\
\text{(vs:7)} & (e_0\ e_1)\{v/x\} & = & e_0\{v/x\}\ e_1\{v/x\} & \\
\text{(vs:8)} & (\mathsf{new}\ e)\{v/x\} & = & \mathsf{new}\ e\{v/x\} & \\
\text{(vs:9)} & (\mathsf{delete}\ e)\{v/x\} & = & \mathsf{delete}\ e\{v/x\} & \\
\text{(vs:10)} & (!e)\{v/x\} & = & !e\{v/x\} & \\
\text{(vs:11)} & (e_0 := e_1)\{v/x\} & = & e_0\{v/x\} := e_1\{v/x\} & \\
\text{(vs:12)} & (\mathtt{t}\#e)\{v/x\} & = & \mathtt{t}\#e\{v/x\} & \\
\text{(vs:13)} & (\mathsf{case}\ e\ \mathsf{of}\ \overline{\mathtt{t}_i\#x_i \to e_i}\ \mathsf{end})\{v/x\} & = & \mathsf{case}\ e\{v/x\}\ \mathsf{of}\ \overline{\mathtt{t}_i\#x_i \to e_i\{v/x\}}\ \mathsf{end} & (x_i \neq x) \\
\text{(vs:14)} & (\mathsf{let}\ x_0 = e_0\ \mathsf{in}\ e_1\ \mathsf{end})\{v/x_1\} & = & \mathsf{let}\ x_0 = e_0\{v/x_1\}\ \mathsf{in}\ e_1\{v/x_1\}\ \mathsf{end} & (x_0 \neq x_1) \\
\text{(vs:15)} & (\mathsf{lock}\ \overline{e})\{v/x\} & = & \mathsf{lock}\ \overline{e\{v/x\}} & \\
\text{(vs:16)} & (\mathsf{unlock}\ \overline{e})\{v/x\} & = & \mathsf{unlock}\ \overline{e\{v/x\}} & \\
\text{(vs:17)} & (\mathsf{fork}\ e)\{v/x\} & = & \mathsf{fork}\ e\{v/x\} &
\end{array}
$$

2. **Location Variable Substitution**, (LS:*)
Similarly, we define location substitution (but here up to renaming of bounded *location* variables) as:

$\boxed{A_0\{p/l\} = A_1}$

$$
\begin{array}{lrcll}
\text{(LS:2.1)} & \rho\{p/l\} & = & \rho & \\
\text{(LS:2.2)} & l\{p/l\} & = & p & \\
\text{(LS:2.3)} & l_0\{p/l_1\} & = & l_0 & (l_0 \neq l_1) \\
\text{(LS:2.4)} & (!A)\{p/l\} & = & !A\{p/l\} & \\
\text{(LS:2.5)} & (A_0 \multimap A_1)\{p/l\} & = & A_0\{p/l\} \multimap A_1\{p/l\} & \\
\text{(LS:2.6)} & (A_0 :: A_1)\{p/l\} & = & A_0\{p/l\} :: A_1\{p/l\} & \\
\text{(LS:2.7)} & [\overline{\mathtt{f} : A}]\{p/l\} & = & [\overline{\mathtt{f} : A\{p/l\}}] & \\
\text{(LS:2.8)} & (\forall l_0.A)\{p/l_1\} & = & \forall l_0.A\{p/l_1\} & (l_0 \neq l_1) \\
\text{(LS:2.9)} & (\exists l_0.A)\{p/l_1\} & = & \exists l_0.A\{p/l_1\} & (l_0 \neq l_1) \\
\text{(LS:2.10)} & (\mathbf{ref}\ p_0)\{p_1/l\} & = & \mathbf{ref}\ p_0\{p_1/l\} & \\
\text{(LS:2.12)} & (\mathbf{rw}\ p_0\ A)\{p_1/l\} & = & \mathbf{rw}\ p_0\{p_1/l\}\ A\{p_1/l\} & \\
\text{(LS:2.13)} & (A_0 * A_1)\{p/l\} & = & A_0\{p/l\} * A_1\{p/l\} & \\
\text{(LS:2.14)} & (\forall X <: A_0.A_1)\{p/l\} & = & \forall X <: A_0\{p/l\}.A_1\{p/l\} & \\
\text{(LS:2.15)} & (\exists X <: A_0.A_1)\{p/l\} & = & \exists X <: A_0\{p/l\}.A_1\{p/l\} & \\
\text{(LS:2.16)} & (X[\overline{U}])\{p/l\} & = & X[\overline{U\{p/l\}}] & \\
\text{(LS:2.17)} & ((\mathbf{rec}\ X(\overline{u}).A)[\overline{U}])\{p/l\} & = & (\mathbf{rec}\ X(\overline{u}).A\{p/l\})[\overline{U\{p/l\}}] & (l \notin \overline{u}) \\
\text{(LS:2.18)} & (\textstyle\sum_i \mathtt{t}_i \# A_i)\{p/l\} & = & \textstyle\sum_i \mathtt{t}_i \# A_i\{p/l\} & \\
\text{(LS:2.19)} & (A_0 \oplus A_1)\{p/l\} & = & A_0\{p/l\} \oplus A_1\{p/l\} & \\
\text{(LS:2.20)} & \mathbf{none}\{p/l\} & = & \mathbf{none} & \\
\text{(LS:2.21)} & (A_0 \Rightarrow A_1)\{p/l\} & = & A_0\{p/l\} \Rightarrow A_1\{p/l\} & \\
\text{(LS:2.22)} & (A_0; A_1)\{p/l\} & = & A_0\{p/l\}; A_1\{p/l\} & \\
\text{(LS:2.23)} & (A_0 \,\&\, A_1)\{p/l\} & = & A_0\{p/l\} \,\&\, A_1\{p/l\} & \\
\text{(LS:2.24)} & \mathbf{top}\{p/l\} & = & \mathbf{top} &
\end{array}
$$

$\boxed{\Gamma_0\{p/l\} = \Gamma_1}$

$$
\begin{array}{lrcll}
\text{(LS:3.1)} & \cdot\{p/l\} & = & \cdot & \\
\text{(LS:3.2)} & (\Gamma, x : A)\{p/l\} & = & \Gamma\{p/l\}, x : A\{p/l\} & \\
\text{(LS:3.3)} & (\Gamma, l_0 : \mathbf{loc})\{p/l_1\} & = & \Gamma\{p/l_1\}, l_0 : \mathbf{loc} & (l_0 \neq l_1) \\
\text{(LS:3.4)} & (\Gamma, X <: A)\{p/l\} & = & \Gamma\{p/l\}, X <: A\{p/l\} & \\
\text{(LS:3.5)} & (\Gamma, X : k)\{p/l\} & = & \Gamma\{p/l\}, X : k &
\end{array}
$$

$\boxed{\Delta_0\{p/l\} = \Delta_1}$

$$
\begin{array}{lrcl}
\text{(LS:4.1)} & \cdot\{p/l\} & = & \cdot \\
\text{(LS:4.2)} & (\Delta, x : A)\{p/l\} & = & \Delta\{p/l\}, x : A\{p/l\} \\
\text{(LS:4.3)} & (\Delta, A)\{p/l\} & = & \Delta\{p/l\}, A\{p/l\}
\end{array}
$$

3. **Type Variable Substitution**, (TS:*)
Finally, we define type substitution (up to renaming of bounded *type* variables) as:

$\boxed{A_0\{A_1/X\} = A_2}$

$$
\begin{array}{lrcll}
\text{(TS:2.1)} & \rho\{A/X\} & = & \rho & \\
\text{(TS:2.2)} & l\{A/X\} & = & l & \\
\text{(TS:2.3)} & (X[\overline{U}])\{A/X\} & = & A[\overline{U\{A/X\}}] & \\
\text{(TS:2.4)} & (X_0[\overline{U}])\{A/X_1\} & = & X_0[\overline{U\{A/X_1\}}] & (X_0 \neq X_1) \\
\text{(TS:2.5)} & (!A_0)\{A_1/X\} & = & !A_0\{A_1/X\} & \\
\text{(TS:2.6)} & (A_0 \multimap A_1)\{A_2/X\} & = & A_0\{A_2/X\} \multimap A_1\{A_2/X\} & \\
\text{(TS:2.7)} & (A_0 :: A_1)\{A_2/X\} & = & A_0\{A_2/X\} :: A_1\{A_2/X\} & \\
\text{(TS:2.8)} & [\overline{\mathtt{f} : A}]\{A_0/X\} & = & [\overline{\mathtt{f} : A\{A_0/X\}}] & \\
\text{(TS:2.9)} & (\forall l.A_0)\{A_1/X\} & = & \forall l.A_0\{A_1/X\} & \\
\text{(TS:2.10)} & (\exists l.A_0)\{A_1/X\} & = & \exists l.A_0\{A_1/X\} & \\
\text{(TS:2.11)} & (\mathbf{ref}\ p)\{A/X\} & = & \mathbf{ref}\ p & \\
\text{(TS:2.13)} & (\mathbf{rw}\ p\ A_0)\{A_1/X\} & = & \mathbf{rw}\ p\ A_0\{A_1/X\} & \\
\text{(TS:2.14)} & (A_0 * A_1)\{A_2/X\} & = & A_0\{A_2/X\} * A_1\{A_2/X\} & \\
\text{(TS:2.15)} & (\forall X_0 <: A_2.A_0)\{A_1/X_1\} & = & \forall X_0 <: A_2\{A_1/X_1\}.A_0\{A_1/X_1\} & (X_0 \neq X_1) \\
\text{(TS:2.16)} & (\exists X_0 <: A_2.A_0)\{A_1/X_1\} & = & \exists X_0 <: A_2\{A_1/X_1\}.A_0\{A_1/X_1\} & (X_0 \neq X_1) \\
\text{(TS:2.17)} & ((\mathbf{rec}\ X_0(\overline{u}).A_0)[\overline{U}])\{A_1/X_1\} & = & (\mathbf{rec}\ X_0(\overline{u}).A_0\{A_1/X_1\})[\overline{U}\{A_1/X_1\}] & (X_0 \neq X_1, X_1 \notin \overline{u}) \\
\text{(TS:2.18)} & (\textstyle\sum_i \mathtt{t}_i \# A_i)\{A/X\} & = & \textstyle\sum_i \mathtt{t}_i \# A_i\{A/X\} & \\
\text{(TS:2.19)} & (A_0 \oplus A_1)\{A/X\} & = & A_0\{A/X\} \oplus A_1\{A/X\} & \\
\text{(TS:2.20)} & \mathbf{none}\{A/X\} & = & \mathbf{none} & \\
\text{(TS:2.21)} & (A_0 \Rightarrow A_1)\{A/X\} & = & A_0\{A/X\} \Rightarrow A_1\{A/X\} & \\
\text{(TS:2.22)} & (A_0; A_1)\{A/X\} & = & A_0\{A/X\}; A_1\{A/X\} & \\
\text{(TS:2.23)} & (A_0 \& A_1)\{A/X\} & = & A_0\{A/X\} \& A_1\{A/X\} & \\
\text{(TS:2.24)} & \mathbf{top}\{A/X\} & = & \mathbf{top} &
\end{array}
$$

$\boxed{\Gamma_0\{A/X\} = \Gamma_1}$

$$
\begin{array}{lrcll}
\text{(TS:3.1)} & \cdot\{A/X\} & = & \cdot & \\
\text{(TS:3.2)} & (\Gamma, x : A_0)\{A_1/X\} & = & \Gamma\{A_1/X\}, x : A_0\{A_1/X\} & \\
\text{(TS:3.3)} & (\Gamma, l : \mathbf{loc})\{A/X\} & = & \Gamma\{A/X\}, l : \mathbf{loc} & \\
\text{(TS:3.4)} & (\Gamma, X_0 <: A_0)\{A_1/X_1\} & = & \Gamma\{A_1/X_1\}, X_0 <: A_0\{A_1/X_1\} & (X_0 \neq X_1) \\
\text{(TS:3.5)} & (\Gamma, X_0 : k)\{A_1/X_1\} & = & \Gamma\{A_1/X_1\}, X_0 : k & (X_0 \neq X_1)
\end{array}
$$

$\boxed{\Delta_0\{A/X\} = \Delta_1}$

$$
\begin{array}{lrcl}
\text{(TS:4.1)} & \cdot\{A/X\} & = & \cdot \\
\text{(TS:4.2)} & (\Delta, x : A_0)\{A_1/X\} & = & \Delta\{A_1/X\}, x : A_0\{A_1/X\} \\
\text{(TS:4.3)} & (\Delta, A_0)\{A_1/X\} & = & \Delta\{A_1/X\}, A_0\{A_1/X\}
\end{array}
$$

A.2 Main Theorems

A.2.1 Subtyping Lemmas

Lemma 15 (Subtyping Inversion)**.** We have the following cases for *types* (A) and for the *linear typing environment* (Δ):

- (Type) If $\Gamma \vdash A <: A'$ then one of the following holds (omitting congruence rules):
  1. $A = A'$.
  2. if $A = !A_0$ then either:
     (a) $A' = A_1$ and $\Gamma \vdash A_0 <: A_1$, or;
     (b) $A' = !A_1$ and $\Gamma \vdash A_0 <: A_1$, or;
     (c) $A' = ![]$.
  3. if $A = A_0 \multimap A_1$ then $A' = A_2 \multimap A_3$ and $\Gamma \vdash A_1 <: A_3$ and $\Gamma \vdash A_2 <: A_0$.
  4. if $A = A_0 :: A_2$ then $A' = A_1 :: A_3$ and $\Gamma \vdash A_0 <: A_1$ and $\Gamma \vdash A_2 <: A_3$.
  5. if $A = [\overline{\mathtt{f} : A}]$ then either:
     (a) $A = [\overline{\mathtt{f} : A}, \mathtt{f}_i : A_i]$ and $A' = [\overline{\mathtt{f} : B}]$ and $[\overline{\mathtt{f} : A}] \neq \emptyset$ and $\Gamma \vdash [\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}]$.
     (b) $A = [\overline{\mathtt{f} : A}, \mathtt{f}_i : A_0]$ and $A' = [\overline{\mathtt{f} : A}, \mathtt{f}_i : B_0]$ and $\Gamma \vdash A_0 <: B_0$ and $\Gamma \vdash [\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}]$.
  6. if $A = \mathbf{rw}\ p\ A_0$ then $A' = \mathbf{rw}\ p\ A_1$ and $\Gamma \vdash A_0 <: A_1$.
  7. if $A = \exists l.A_0$ then $A' = \exists l.A_1$ and $\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1$.
  8. if $A = \forall l.A_0$ then either:
     (a) $A' = \forall l.A_1$ and $\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1$, or;
     (b) $A' = A_1\{p/l\}$ and $\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1$.
  9. if $A = \exists X <: A_3.A_0$ then $A' = \exists X <: A_3.A_1$ and $\Gamma, X : \mathbf{type}, X <: A_3 \vdash A_0 <: A_1$.
  10. if $A = \forall X <: A_3.A_0$ then either:
      (a) $A' = \forall X <: A_3.A_1$ and $\Gamma, X : \mathbf{type}, X <: A_3 \vdash A_0 <: A_1$, or;
      (b) $A' = A_0'\{A_2/X\}$ and $\Gamma \vdash A_2 <: A_1$ and $\Gamma, X : \mathbf{type}, X <: A_3 \vdash A_0 <: A_0'$.
  11. if $A = A_0 * A_1$ then $A' = A_0 * A_2$ and $\Gamma \vdash A_1 <: A_2$.
  12. if $A = \sum_i \mathtt{t}_i \# A_i$ then $A' = \sum_i \mathtt{t}_i \# B_i$ and $n \leq m$ and $\overline{\Gamma \vdash A_i <: B_i}$.
  13. if $A = A_0 \& A_1$ then $A' = A_2$ and $\Gamma \vdash A_0 <: A_2$.
  14. $A' = A \oplus A''$ and $\Gamma \vdash A <: A''$.
  15. $A' = \mathbf{top}$.
  16. $A' = \exists l.A_0\{l/p\}$ and $\Gamma \vdash A <: A_0$.
  17. $A' = \exists X <: A_1.A_0\{X/A_2\}$ and $\Gamma \vdash A_2 <: A_1$ and $\Gamma \vdash A <: A_0$.
  18. $A = X$ and $X <: A_0 \in \Gamma$ and $\Gamma \vdash A_0 <: A'$.
  19. $\Gamma = \Gamma'', \Gamma'$ and $\Gamma' \vdash A <: A'$

- (Delta) If $\Gamma \vdash \Delta <: \Delta'$ then one of the following holds:
  1. $\Delta = \Delta'$.
  2. if $\Delta = \Delta_0, x : A_0$ then $\Delta' = \Delta_1, x : A_1$ and $\Gamma \vdash \Delta_0 <: \Delta_1$ and $\Gamma \vdash A_0 <: A_1$.
  3. if $\Delta = \Delta_0, A_0$ then $\Delta' = \Delta_1, A_1$ and $\Gamma \vdash \Delta_0 <: \Delta_1$ and $\Gamma \vdash A_0 <: A_1$.

4. if $\Delta = \Delta_0, A_0, A_1$ then either:
   (a) $\Delta' = \Delta'_0, A'_0 * A'_1$ and $\Gamma \vdash \Delta_0, A_0, A_1 <: \Delta'_0, A'_0, A'_1$, or;
   (b) case (3) with A_0, or;
   (c) case (3) with A_1.
5. if $\Delta = \Delta_0, A_0 * A_1$ then $\Delta' = \Delta'_0, A'_0, A'_1$ and $\Gamma \vdash \Delta_0, A_0 * A_1 <: \Delta'_0, A'_0 * A'_1$.
6. if $\Delta = \Delta_0, \mathbf{none}$ then $\Delta' = \Delta_1$ and $\Gamma \vdash \Delta_0 <: \Delta_1$.
7. $\Delta' = \Delta_0, \mathbf{none}$ and $\Gamma \vdash \Delta <: \Delta_0$.
8. if $\Delta = \Delta_0, A_0 \oplus A_1$ then $\Gamma \vdash \Delta_0, A_0 <: \Delta'$ and $\Gamma \vdash \Delta_0, A_1 <: \Delta'$.
9. if $\Delta' = \Delta_1, A_0 \& A_1$ then $\Gamma \vdash \Delta <: \Delta_1, A_0$ and $\Gamma \vdash \Delta <: \Delta_1, A_1$.
10. if $\Delta = \Delta_0, A_0$ and $\Delta' = \Delta_1, A_1, A_2$ then $\Gamma \vdash \Delta_0 <: \Delta_1$ and $\Gamma \vdash A_0 \Rrightarrow A_1 \parallel A_2$.

Proof. The proof is straightforward by induction on the subtyping derivation.

□

Lemma 16 (Subtype Transitivity)**.** We have that:

- If $\Gamma \vdash A_0 <: A_1$ and $\Gamma \vdash A_1 <: A_2$ then $\Gamma \vdash A_0 <: A_2$.
- If $\Gamma \vdash \Delta_0 <: \Delta_1$ and $\Gamma \vdash \Delta_1 <: \Delta_2$ then $\Gamma \vdash \Delta_0 <: \Delta_2$.

Proof. The proof is straightforward (but lengthy) by induction on the derivation of subtyping. Note that transitivity for (SD:SHARE) requires the subtyping extension. □

A.2.2 Store Typing Lemmas

Lemma 17 (Store Typing Inversion)**.** If

$$\Gamma \mid \Delta \vdash H$$

then one of the following holds:

1. $\Gamma = \cdot$ and $\Delta = \cdot$ and $H = \cdot$.
2. if $\Gamma = \Gamma', \rho : \mathbf{loc}$ then $\Gamma' \mid \Delta \vdash H$.
3. $\Gamma \vdash \Delta' <: \Delta$ and $\Gamma \mid \Delta' \vdash H$.
4. if $\Delta = \Delta', \mathbf{rw}\ \rho\ A$ and $H = H', \rho \hookrightarrow v$ then $\Gamma \mid \Delta', \Delta_v \vdash H'$ and $\Gamma \mid \Delta_v \vdash v : A \dashv \cdot$.
5. if $\Delta = \Delta', A_0,\ A_1; A_2$ and $H = H', \overline{\rho^\bullet \hookrightarrow v}$ then $\mathbf{locs}(A_0) = \mathbf{locs}(A_1) = \overline{\rho}$ and $\Gamma \mid A_0 \vdash \overline{\rho \hookrightarrow v}$ and $\Gamma \mid A_1 \vdash \overline{\rho \hookrightarrow v'}$ and $\Gamma \mid \Delta', A_2 \vdash H', \overline{\rho \hookrightarrow v'}$.
6. if $\Delta = \Delta',\ A_1; A_2$ then $\mathbf{locs}(A_1) = \overline{\rho}$ and $\overline{\rho \hookrightarrow v} \notin H$ and $\Gamma \mid A_1 \vdash \overline{\rho \hookrightarrow v'}$ and $\Gamma \mid \Delta', A_2 \vdash H, \overline{\rho \hookrightarrow v'}$.
7. if $\Gamma \mid \Delta, \forall l.A \vdash H$ then $\Gamma, l : \mathbf{loc} \mid \Delta, A \vdash H$.
8. if $\Gamma \mid \Delta, \forall X <: A''.A \vdash H$ then $\Gamma, X : \mathbf{type}, X <: A'' \mid \Delta, A \vdash H$.

Proof. Straightforward induction on the derivation of $\Gamma \mid \Delta \vdash H$. □

Lemma 18 (Protocol Store Typing)**.** If we have:

$$\mathbf{locs}(A_0) = \overline{\rho} \qquad \Gamma \mid A_0 \vdash H \qquad \overline{\rho \hookrightarrow v} \in H \qquad \Gamma \vdash A_0 \Rrightarrow A_1 \parallel A_2$$

then each choice (&) of A_1 and A_2 must include an alternative ($\oplus$) such that its rely type is A_0. I.e.:

$$A_0 \in \texttt{rely}(A_1) \qquad A_0 \in \texttt{rely}(A_2)$$

where:

$$\begin{array}{lcll} \texttt{rely}(\ P_0 \& P_1\ ) & = & \texttt{rely}(P_0) & \text{if } \texttt{rely}(P_0) = \texttt{rely}(P_1) \\ \texttt{rely}(\ P_0 \oplus P_1\ ) & = & \texttt{rely}(P_0) \cup \texttt{rely}(P_1) & \\ \texttt{rely}(\ A \Rightarrow P\ ) & = & \{A\} & \end{array}$$

Proof. Straightforward by protocol composition. We know that by the conditions for protocols to be well-formed that only one alternative ($\oplus$) can exists for a given rely type. Thus, only one such alternative can rely on A_0. Likewise, we have that any choice (&) must itself confirm with the state of the shared state. Thus, whenever the shared locations are shared they must respect the rely type of all the protocols that are sharing that state. □

A.2.3 Values Inversion Lemma

Lemma 19 (Values Inversion)**.** If v is a value such that

$$\Gamma \mid \Delta \vdash v : A_0 \dashv \cdot$$

then one of the following holds:

1. if $A_0 = ![]$ then:
$$\Delta = \cdot \qquad \Gamma \mid \cdot \vdash v : ![] \dashv \cdot$$
2. if $A_0 = !A_1$ then:
$$\Delta = \cdot \qquad \Gamma \mid \cdot \vdash v : A_1 \dashv \cdot$$
3. if $A_0 = A_1 :: A_2$ then:
$$\Gamma \mid \Delta \vdash v : A_1 \dashv A_2$$
4. if $A_0 = \mathbf{ref}\, \rho$ then:
$$v = \rho \qquad \rho : \mathbf{loc} \in \Gamma \qquad \Delta = \cdot$$
5. if $A_0 = A \multimap A'$ then:
$$v = \lambda x.e \qquad \Gamma \mid \Delta, x : A \vdash e : A' \dashv \cdot$$
6. if $A_0 = \forall l.A$ then:
$$\Gamma, l : \mathbf{loc} \mid \Delta \vdash v : A \dashv \cdot$$
7. if $A_0 = \exists l.A$ then:
$$\Gamma \mid \Delta \vdash v : A\{p/l\} \dashv \cdot$$
8. if $A_0 = [\overline{\mathtt{f} : A}]$ then:
$$v = \{\overline{\mathtt{f} = v'}\} \qquad \overline{\Gamma \mid \Delta \vdash v'_i : A_i \dashv \cdot}$$
(Note: the record value can have more fields than those listed in the type but only the fields in the type will be known by inversion.)
9. if $A_0 = \forall X <: A'.A$ then:
$$\Gamma, X : \mathbf{type}, X <: A' \mid \Delta \vdash v : A \dashv \cdot$$
10. if $A_0 = \exists X <: A'.A$ then:
$$\Gamma \mid \Delta \vdash v : A\{A'/X\} \dashv \cdot$$
11. if $A_0 = \sum_i \mathtt{t}_i \# A_i$ then:
$$v = \mathtt{t}_i \# v_i \qquad \Gamma \mid \Delta \vdash v_i : A_i \dashv \cdot$$
for some i.
12. if $A_0 = (\mathbf{rec}\, X(\overline{u}).A)[\overline{U}]$ then:
$$\Gamma \mid \Delta \vdash v : A\{\mathbf{rec}\, X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\} \dashv \cdot$$

13. if $\Delta = \Delta', A_1 \oplus A_2$ then:

$$\Gamma \mid \Delta', A_1 \vdash v : A_0 \dashv \cdot \qquad \Gamma \mid \Delta', A_2 \vdash v : A_0 \dashv \cdot$$

14. if $A_0 = A_1 \oplus A_2$ then either:

$$\Gamma \mid \Delta \vdash v : A_1 \dashv \cdot \qquad \text{or} \qquad \Gamma \mid \Delta \vdash v : A_2 \dashv \cdot$$

15. if $A_0 = \mathbf{top}$ then:

$$\Gamma \mid \Delta \vdash v : A_1 \dashv \cdot$$

(Note: the remaining types, such as &, do not appear in this lemma since they are related to capabilities, not values, and therefore cannot be used to directly type some value—i.e. they can get stacked on top of some other type, but not be used to type the value itself).

Proof. By induction on the derivation of $\Gamma \mid \Delta \vdash v : A_0 \dashv \cdot$.

Case (T:REF) - We have:

$\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\, \rho \dashv \cdot$ (1)

by hypothesis.

Thus, we conclude by case 4 of the definition.

Case (T:PURE) - We have:

$\Gamma \mid \cdot \vdash v : !A_1 \dashv \cdot$ (1)

by hypothesis.

$\Gamma \mid \cdot \vdash v : A_1 \dashv \cdot$ (2)

by inversion on (T:PURE).

Thus, we conclude by case 2 of the definition.

Case (T:UNIT) - We have:

$\Gamma \mid \cdot \vdash v : ![] \dashv \cdot$ (1)

by hypothesis.

Thus, we conclude by case 1 of the definition.

Case (T:PURE-READ), (T:LINEAR-READ), (T:PURE-ELIM), (T:NEW), (T:DELETE), (T:ASSIGN), (T:DEREFERENCE-LINEAR), (T:DEREFERENCE-PURE) - Not applicable.

Case (T:RECORD) - We have:

$\Gamma \mid \Delta \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (1)

by hypothesis.

$\overline{\Gamma \mid \Delta \vdash v_i : A_i \dashv \cdot}$ (2)

by inversion on (T:RECORD).

Thus, we conclude by case 8 of the definition.

Case (T:SELECTION), (T:APPLICATION) - Not applicable.

Case (T:FUNCTION) - We have:

$\Gamma \mid \Delta \vdash \lambda x.e : A_0 \multimap A_1 \dashv \cdot$ (1)
by hypothesis.

$\Gamma \mid \Delta, x : A_0 \vdash e : A_1 \dashv \cdot$ (2)
by inversion on (T:FUNCTION).

Thus, we conclude by case 5 of the definition.

Case (T:CAP-ELIM), (T:CAP-UNSTACK), (T:APPLICATION) - Not applicable.

Case (T:CAP-STACK) - We have:

$\Gamma \mid \Delta \vdash v : A_0 :: A_1 \dashv \cdot$ (1)
by hypothesis.

$\Gamma \mid \Delta \vdash v : A_0 \dashv A_1$ (2)
by inversion on (T:CAP-STACK).

Thus, we conclude by case 3 of the definition.

Case (T:FORALL-LOC-VAL) We have:

$\Gamma \mid \Delta \vdash v : \forall l.A \dashv \cdot$ (1)
by hypothesis.

$\Gamma, l : \mathbf{loc} \mid \Delta \vdash v : A \dashv \cdot$ (2)
by inversion on (T:FORALL-LOC-VAL) with (1).

Thus, we conclude by case 6 of the definition.

Case (T:FORALL-TYPE-VAL) We have:

$\Gamma \mid \Delta \vdash v : \forall X <: A'.A \dashv \cdot$ (1)
by hypothesis.

$\Gamma, X : \mathbf{type}, X <: A' \mid \Delta \vdash v : A \dashv \cdot$ (2)
by inversion on (T:FORALL-LOC-VAL) with (1).

Thus, we conclude by case 9 of the definition.

Case (T:TAG) We have:

$\Gamma \mid \Delta \vdash \mathtt{t}\#v : \mathtt{t}\#A \dashv \cdot$ (1)
by hypothesis.

$\Gamma \mid \Delta \vdash v : A \dashv \cdot$ (2)
by inversion on (T:TAG).

Thus, we conclude by case 11 of the definition.

Case (T:CASE) Not applicable.

Case (T:ALTERNATIVE-LEFT) We have:

$\Gamma \mid \Delta, A_0 \oplus A_1 \vdash v : A_2 \dashv \cdot$ (1)

by hypothesis.

$\Gamma \mid \Delta, A_0 \vdash v : A_2 \dashv \cdot$ (2)

$\Gamma \mid \Delta, A_1 \vdash v : A_2 \dashv \cdot$ (3)

by inversion on (T:Alternative-Left).

Thus, we conclude by case 13 of the definition.

Case (T:Frame) Only case is when Δ environment on the right is not empty which is immediate by applying the induction hypothesis.

Case (T:TypeOpenBind), (T:TypeOpenCap), (T:LocOpenBind), (T:LocOpenCap) - Immediate by applying the induction hypothesis.

Case (T:Subsumption) We have:

$\Gamma \mid \Delta \vdash v : A_1 \dashv \cdot$ (1)

by hypothesis.

$\Gamma \vdash \Delta <: \Delta'$ (2)

$\Gamma \mid \Delta' \vdash v : A_0 \dashv \cdot$ (3)

$\Gamma \vdash A_0 <: A_1$ (4)

$\Gamma \vdash \cdot <: \cdot$ (5)

by inversion on (T:Subsumption).

Remember that we are showing that (1) obeys the definition above.
By applying the induction hypothesis on (3) we have that one of the following holds:

1. if $A_0 = ![]$ then:

   $\Delta' = \cdot$ (1.1)

   $\Gamma \mid \cdot \vdash v : ![] \dashv \cdot$ (1.2)

   $\Gamma \vdash ![] <: A_1$ (1.3)

   by case 1 of the hypothesis and rewriting (4).

   Then, by (Subtyping Inversion) on (1.3) we have that either:

   • $[1/2(c)]\ \ A_1 = ![]$ (1.5)

   we conclude as case 2 of the definition.

   • $[14]\ \ A_1 = ![] \oplus A'$ (1.6)

   we conclude as case 14 of the definition using (3).

   • $[15]\ \ A_1 = \mathbf{top}$ (1.6)

   we conclude as case 15 of the definition using (3).

   Similarly, sub-cases [16] and [17] are immediate by cases 10 and 7 of the definition.

2. if $A_0 = !A$ then:

   $\Delta' = \cdot$ (2.1)

   $\Gamma \mid \cdot \vdash v : A \dashv \cdot$ (2.2)

   $\Gamma \vdash !A <: A_1$ (2.3)

   by case 2 of the hypothesis and rewriting (4).

by (Subtyping Inversion) on (2.3) we have that either:
• [1] $A_1 = !A$
Thus, we conclude by case 2 of the definition through (2.2).
• [2(a)] $A_1 = A$
Thus, we conclude by induction hypothesis on (2.2).
• [2(b)] $A_1 = !A'$ and $\Gamma \vdash A <: A'$
$\Gamma \mid \cdot \vdash v : A' \dashv \cdot$ (2.4)
by (T:Subsumption) on (2.2) with $\Gamma \vdash A <: A'$.
Thus, we conclude by case 2 of the definition with (2.4).
• [2(c)] $A_1 = ![]$
$\Gamma \mid \cdot \vdash v : ![] \dashv \cdot$ (2.5)
by (T:Unit) on v.
Thus, we conclude by case 2 of the definition.
• [14] $A_1 = !A \oplus A'$ (2.6)
and we conclude as case 14 of the definition using (3).
Similarly, sub-cases [15], [16] and [17] are immediate by cases 15, 10, and 7 of the definition.

3. if $A_0 = A \multimap A'$ then:

$v = \lambda x.e$ (3.1)
$\Gamma \mid \Delta, x : A \vdash e : A' \dashv \cdot$ (3.2)
$\Gamma \vdash (A \multimap A') <: A_1$ (3.3)
by case 5 of the hypothesis and rewriting (4).
by (Subtyping Inversion):
(Note: we omit the remaining cases since they are straightforward)
$A_1 = A'' \multimap A'''$ (3.4)
$\Gamma \vdash A' <: A'''$ (3.5)
$\Gamma \vdash A'' <: A$ (3.6)
by (Subtyping Inversion) on (3.3) we have that:
$\Gamma \mid \Delta, x : A \vdash e : A''' \dashv \cdot$ (3.7)
by (T:Subsumption) on (3.2) and (3.5)
$\Gamma \mid \Delta, x : A'' \vdash e : A''' \dashv \cdot$ (3.8)
by (T:Subsumption) on (3.7), (3.6) and (SD:Var) with (2).
Thus, with (3.8) and (3.1) we conclude by case 5 of the definition.

4. if $A_0 = A :: A'$ then:

$\Gamma \mid \Delta' \vdash v : A \dashv A'$ (4.1)
$\Gamma \vdash A :: A' <: A_1$ (4.2)
by case 3 of the hypothesis and rewriting (4).
by (Subtyping Inversion) on (4.2) we have that:
(Note: we omit the remaining cases since they are straightforward)
$A_1 = A'' :: A'''$ (4.3)

$\Gamma \vdash A <: A''$ (4.4)
$\Gamma \vdash A' <: A'''$ (4.5)
$\Gamma \mid \Delta \vdash v : A'' \dashv A'''$ (4.6)
by (T:Subsumption) on (4.1) with (4.4) and (4.5).
Thus, we conclude by case 3 of the definition.

5. if $A_0 = [\overline{\mathtt{f} : A}]$ then:
$v = \{\overline{\mathtt{f} = v'}\}$ (5.1)
$\overline{\Gamma \mid \Delta' \vdash v'_i : A_i \dashv \cdot}$ (5.2)
$\Gamma \vdash [\overline{\mathtt{f} : A}] <: A_1$ (5.5)
by case 8 of the hypothesis and rewriting (4).
by (Subtyping Inversion) on (5.5) we have that either:
(Note: the remaining [1], [14], [15], [16], [17] cases are straightforward)
• [5(*b*)] $A_0 = [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A']$ and
$A_1 = [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}_i : A'']$ (5.6)
$\Gamma \vdash A' <: A''$ (5.7)
Thus, by (T:Subsumption) on (5.2) and (5.7) we conclude by case 8 of the definition.
• [5(*a*)] $A_0 = [\overline{\mathtt{f} : A}, \mathtt{f}_i : A]$ and
$A_1 = [\overline{\mathtt{f} : A}]$ and $A_1 \neq \emptyset$.
Thus, by (T:Record) with (5.1) and ignoring the dropped field, we conclude by case 8 of the definition. Note that all fields have the same effect and by $i > 0$ we ensure that subtyping leaves at least one field to do such effect.
6. if $A_0 = \exists l.A$ then:
$\Gamma \mid \Delta' \vdash v : A\{p/l\} \dashv \cdot$ (6.1)
$\Gamma \vdash \exists l.A <: A_1$ (6.2)
by case 7 of the hypothesis and rewriting (4).
by (Subtyping Inversion) on (6.2) we have that:
$A_1 = \exists l.A'$ (6.4)
$\Gamma \vdash A <: A'$ (6.5)
$\Gamma \mid \Delta \vdash v' : A'\{p/l\} \dashv \cdot$ (6.6)
by (T:Subsumption) on (6.2) and (6.5).
Thus, we conclude by case 7 of the definition.

The remaining cases are straightforward since we can either "pack again" but that means (6.1) is the unpacked type thus obeying the definition, or we have one of the cases that are similar to those above such as [1],[14], or [15].

7. if $A_0 = \forall l.A$ then:
$\Gamma, l : \mathbf{loc} \mid \Delta' \vdash v : A \dashv \cdot$ (7.1)
$\Gamma \vdash \forall l.A <: A_1$ (7.2)
by case 6 of the hypothesis and rewriting (4).

by (Subtyping Inversion) on (7.2) we have that:
(Note: the remaining cases are straightforward and are omitted)
• [8(a)] $A_1 = \forall l.A'$ (7.3)
$\Gamma \vdash A <: A'$ (7.4)
$\Gamma, l : \mathbf{loc} \mid \Delta \vdash e : A' \dashv \cdot$ (7.5)
by (T:Subsumption) on (7.1) and (7.4).
• [8(b)] $A_1 = A\{p/l\}$ (7.6)
Immediate by (Substitution Lemma) and induction hypothesis on (7.1).

8. if $A_0 = \mathbf{ref}\,\rho$ then:
$v = \rho$ (8.1)
$\rho : \mathbf{loc} \in \Gamma$ (8.2)
$\Delta = \cdot$ (8.3)
$\Gamma \vdash \mathbf{ref}\,\rho <: A_1$ (8.4)
by case 4 of the hypothesis and rewriting (4).
(Note: the remaining [14] is straightforward)
by (Subtyping Inversion) on (8.4) we have:
• [1] $A1 = (\mathbf{ref}\ p)$
Thus, we conclude by case 2 of the definition.

9. if $A_0 = \exists X <: A'.A$, analogous to $\exists l.A$.

10. if $A_0 = \forall X <: A'.A$, analogous to $\forall l.A$.

11. if $A_0 = \sum_i \mathtt{t}_i \# A'_i$ then:
$v = \mathtt{t}_i \# v_i$ (11.1)
$\Gamma \mid \Delta' \vdash v_i : A'_i \dashv \cdot$ (11.2)
for some i.
$\Gamma \vdash \sum_i \mathtt{t}_i \# A'_i <: A_1$ (11.3)
(Note: the remaining [1] and [14] cases are straightforward)
by (Subtyping Inversion) on (8.4) we have that:
$A_1 = \mathtt{t}' \# A' + ... + \sum_i \mathtt{t}_i \# A'_i$ (11.4)
Thus, by (11.2) we conclude by case 11 of the definition.

12. if $A_0 = (\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]$ then:
$\Gamma \mid \Delta' \vdash v : A_1 \dashv \cdot$ (12.1)
$\Gamma \vdash (\mathbf{rec}\ X(\overline{u}).A)[\overline{U}] <: A_1$ (12.2)
by case 12 of the hypothesis and rewriting (4).
(Note: the remaining cases are straightforward)
by (Subtyping Inversion) on (12.2) we have:
• [1] $A_1 = A\{\mathbf{rec}\ X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\}$
Thus, we conclude by induction hypothesis on (12.1) combined with (T:Subsumption).

13. if $\Delta = \Delta', A_2 \oplus A_3$ then:
    $\Gamma \mid \Delta', A_2 \vdash v : A_0 \dashv \cdot$ (13.1)
    $\Gamma \mid \Delta', A_3 \vdash v : A_0 \dashv \cdot$ (13.2)
    $\Gamma \vdash A_0 <: A_1$ (13.3)
    By induction hypothesis on each case and then (T:SUBSUMPTION).

14. if $A_0 = A_1 \oplus A_2$ then either:
    $\Gamma \mid \Delta' \vdash v : A_1 \dashv \cdot$ (14.1)
    $\Gamma \mid \Delta' \vdash v : A_2 \dashv \cdot$ (14.2)
    and:
    $\Gamma \vdash A_1 \oplus A_2 <: A'$ (14.3)

    This case is analogous to previous ones by applying (Subtyping Inversion) on (14.3) yielding cases [1] and [14]. The first is immediate, the second is closed by considering either (14.1) or (14.2) through (T:SUBSUMPTION).
15. if $A_0 =$ **top** then $A_1 =$ **top** is the only possibility and we conclude by (1) with definition 15.

Case (T:LET), (T:SHARE), (T:LOCK-RELY), (T:UNLOCK-GUARANTEE), (T:FORK) - Not values.

□

A.2.4 Free Variables Lemma

Lemma 20 (Free Variables)**.** If $\Gamma \mid \Delta_0, x : A_0 \vdash e : A_1 \dashv \Delta_1$ and $x \in \mathtt{fv}(e)$ then $x \notin \Delta_1$.

where $\mathtt{fv}(e) \triangleq$ "set of all free variables in the expression e"

Proof. We proceed by induction on the derivation of $\Gamma \mid \Delta_0, x : A_0 \vdash e : A_1 \dashv \Delta_1$.

Case (T:REF), (T:PURE), (T:UNIT), (T:PURE-READ) - the linear typing environment is empty.

Case (T:LINEAR-READ) - We have:

$\Gamma \mid x : A \vdash x : A \dashv \cdot$ (1)
$x \in \mathtt{fv}(x)$ (2)
by hypothesis.

Therefore, we immediately conclude $x \notin \cdot$.

Case (T:PURE-ELIM) - We have:

$\Gamma \mid \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(e)$ (2)
by hypothesis.
$\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1$ (3)
by inversion on (T:PURE-ELIM).
$x \notin \Delta_1$ (4)
because x is in the linear environment (and cannot appear duplicated in Δ's).

Therefore, we conclude.

(Note: case when x is not the one used in the (T:PURE-ELIM) rule is a direct application of the induction hypothesis.)

Case (T:NEW) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash \mathsf{new}\ v : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(\mathsf{new}\ v)$ (2)
by hypothesis.
$\Gamma \mid \Delta_0, x : A_0 \vdash v : A \dashv \Delta_1$ (3)
by inversion on (T:NEW) with (1).
$x \in \mathtt{fv}(v)$ (4)
[$\mathtt{fv}(\mathsf{new}\ v) = \mathtt{fv}(v)$]
by definition of $\mathtt{fv}$ and (2).
$x \notin \Delta_1$ (5)
by induction hypothesis on (3) and (4).

Therefore, we conclude.

Case (T:DELETE) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash \mathsf{delete}\ v : \exists l.A \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(\mathsf{delete}\ v)$ (2)
by hypothesis.

$\Gamma \mid \Delta_0, x : A_0 \vdash v : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta_1$ (3)
by inversion on (T:DELETE) with (1).

$x \in \mathtt{fv}(v)$ (4)
[$\mathtt{fv}(\mathsf{delete}\ v) = \mathtt{fv}(v)$]
by definition of $\mathtt{fv}$ and (2).

$x \notin \Delta_1$ (5)
by induction hypothesis on (3) and (4).

Therefore, we conclude.

Case (T:ASSIGN) - We have:

$\Gamma \mid \Delta_0, x : A \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_0$ (1)
$x \in \mathtt{fv}(v_0 := v_1)$ (2)
by hypothesis.

$\Gamma \mid \Delta_0, x : A \vdash v_1 : A_0 \dashv \Delta_1$ (3)
$\Gamma \mid \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (4)
by inversion on (T:ASSIGN) with (1).
[$\mathtt{fv}(v_0 := v_1) = \mathtt{fv}(v_0) \cup \mathtt{fv}(v_1)$]

Therefore, we have the following possibilities:

1. $x \in \mathtt{fv}(v_0) \wedge x \notin \mathtt{fv}(v_1)$
   $(x : A) \in \Delta_1$ (1.1)
   by $x \notin \mathtt{fv}(v_1)$.
   $x \notin \Delta_2, \mathbf{rw}\ p\ A_1$ (1.2)
   by induction hypothesis on (4) with (1.1).
   $x \notin \Delta_2, \mathbf{rw}\ p\ A_0$ (1.3)
   since the capability trivially obeys the restriction (since x is not a type).
   Thus, we conclude.
2. $x \in \mathtt{fv}(v_1) \wedge x \notin \mathtt{fv}(v_0)$
   $x \notin \Delta_1$ (2.1)
   by induction hypothesis on (3) and case assumption.
   $x \notin \Delta_2, \mathbf{rw}\ p\ A_1$ (2.2)
   by (2.1) and (4).
   $x \notin \Delta_2, \mathbf{rw}\ p\ A_0$ (2.3)
   since the capability trivially obeys the restriction on (2.2).
   Thus, we conclude.
3. $x \in \mathtt{fv}(v_0) \wedge x \in \mathtt{fv}(v_1)$
   $x \notin \Delta_1$ (3.1)
   by induction hypothesis on (3) and case assumption.
   We reach a contradiction since v_0 is well-typed by (4) but $x \in \mathtt{fv}(v_1)$ contradicts (3.1).
   Thus, such case is impossible to occur in a well-typed expression.

Thus, we conclude.

Case (T:Dereference-Linear) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash\ !v : A \dashv \Delta_1, \mathbf{rw}\ p\ ![]$ (1)
$x \in \mathtt{fv}(!v)$ (2)
by hypothesis.

$\Gamma \mid \Delta_0, x : A_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A$ (3)
by inversion on (T:Dereference-Linear).
[$\mathtt{fv}(!v) = \mathtt{fv}(v)$]

$x \in \mathtt{fv}(v)$ (4)
by definition of $\mathtt{fv}$ and (2).

$x \notin \Delta_1, \mathbf{rw}\ p\ A$ (5)
by induction hypothesis on (3) and (4).

$x \notin \Delta_1, \mathbf{rw}\ p\ ![]$ (6)
by (5) and since x cannot be in $\mathbf{rw}\ p\ ![]$.

Thus, we conclude.

Case (T:Dereference-Pure) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash\ !v : !A_1 \dashv \Delta_1, \mathbf{rw}\ p\ !A_1$ (1)
$x \in \mathtt{fv}(!v)$ (2)
by hypothesis.

$\Gamma \mid \Delta_0, x : A_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ !A_1$ (3)
by inversion on (T:Dereference-Pure).
[$\mathtt{fv}(!e) = \mathtt{fv}(v)$]

$x \in \mathtt{fv}(v)$ (4)
by definition of $\mathtt{fv}$ and (2).

$x \notin \Delta_1, \mathbf{rw}\ p\ !A_1$ (5)
by induction hypothesis on (3) and (4).

Thus, we conclude.

Case (T:Record) - We have:

$\Gamma \mid \Delta, x : A_0 \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (1)
$x \in \mathtt{fv}(\{\overline{\mathtt{f} = v}\})$ (2)
by hypothesis.

Therefore, we immediately conclude $x \notin \cdot$.

Case (T:Selection) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash v.\mathtt{f}_i : A_i \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(v.\mathtt{f})$ (2)
by hypothesis.

$\Gamma \mid \Delta_0, x : A_0 \vdash v : [\overline{\mathtt{f} : A}] \dashv \Delta_1$ (3)
by inversion on (T:Selection).

$[\ \mathtt{fv}(v.\mathtt{f}) = \mathtt{fv}(v)\ ]$

$x \in \mathtt{fv}(v)$ (4)

by definition of $\mathtt{fv}$ and (2).

$x \notin \Delta_1$ (5)

by induction hypothesis on (3) and (4).

Thus, we conclude.

Case (T:APPLICATION) - We have:

$\Gamma \mid \Delta_0, x : A \vdash v_0\ v_1 : A_1 \dashv \Delta_2$ (1)

$x \in \mathtt{fv}(v_0\ v_1)$ (2)

$[\ \mathtt{fv}(v_0\ v_1) = \mathtt{fv}(v_0) \cup \mathtt{fv}(v_1)\ ]$

by hypothesis.

$\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1$ (3)

$\Gamma \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2$ (4)

by inversion on (T:APPLICATION) with (1).

Therefore, we have the following possibilities:

1. $x \in \mathtt{fv}(v_1) \wedge x \notin \mathtt{fv}(v_0)$

   $\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1$ (1.1)

   $\Delta_1 = \Delta_1', x : A$ (1.2)

   by $x \notin \mathtt{fv}(v_0)$.

   $\Gamma \mid \Delta_1', x : A \vdash v_1 : A_0 \dashv \Delta_2$ (1.3)

   by rewriting (4) with (1.2).

   $x \notin \Delta_2$ (1.4)

   by induction hypothesis on (1.3) and sub-case hypothesis.

   Thus, we conclude.
2. $x \in \mathtt{fv}(v_0) \wedge x \in \mathtt{fv}(v_1)$

   $x \notin \Delta_1$ (2.1)

   by induction hypothesis on (3) and case assumption.

   We reach a contradiction since v_1 is well-typed by (4) but $x \in \mathtt{fv}(v_1)$ contradicts (2.1). Thus, such case is impossible to occur in a well-typed expression. Therefore, we conclude.
3. $x \in \mathtt{fv}(v_0) \wedge x \notin \mathtt{fv}(v_1)$

   $x \notin \Delta_1$ (3.1)

   by induction hypothesis on (3) and case assumption.

   $x \notin \Delta_2$ (3.2)

   by (3.1) and (4).

   Thus, we conclude.

Case (T:FUNCTION) - We have:

$\Gamma \mid \Delta, x : A_0 \vdash \lambda x_0.e : A_2 \multimap A_1 \dashv \cdot$ (1)

$x \in \mathtt{fv}(\lambda x_0.e)$ (2)

by hypothesis.

$x \notin \cdot$ (3)

since it is the empty environment.

Thus, we conclude.

Case (T:CAP-ELIM) - We have:

$\Gamma \mid \Delta_0, x : A_1 :: A_2 \vdash e : A_0 \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(e)$ (2)

by hypothesis.

$\Gamma \mid \Delta_0, x : A_1, A_2 \vdash e : A_0 \dashv \Delta_1$ (3)

by inversion on (T:CAP-ELIM) on (1).

$x \notin \Delta_1$ (4)

by induction hypothesis on (2) and (3).

Thus, we conclude.

Case (T:CAP-STACK) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash e : A_1 :: A_2 \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(e)$ (2)

by hypothesis.

$\Gamma \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1, A_2$ (3)

by inversion on (T:CAP-STACK) on (1).

$x \notin \Delta_1, A_2$ (4)

by induction hypothesis on (3) and (2).

$x \notin \Delta_1$ (5)

by (4).

Thus, we conclude.

Case (T:CAP-UNSTACK) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash e : A_1 \dashv \Delta_1, A_2$ (1)
$x \in \mathtt{fv}(e)$ (2)

by hypothesis.

$\Gamma \mid \Delta_0, x : A_0 \vdash e : A_1 :: A_2 \dashv \Delta_1$ (3)

by inversion on (T:CAP-UNSTACK) with (1).

$x \notin \Delta$ (4)

by induction hypothesis with (3) and (2).

Thus, we conclude.

Case (T:FRAME) - We have:

$\Gamma \mid (\Delta_0, x : A_0), \Delta_2 \vdash e : A \dashv \Delta_1, \Delta_2$ (1)
$x \in \mathtt{fv}(e)$ (2)

by hypothesis.

$\Gamma \mid \Delta_0, x : A_0 \vdash e : A \dashv \Delta_1$ (3)

by inversion on (T:FRAME) with (1), note by (2) x must be in environment.

$x \notin \Delta_1$ (4)

by induction hypothesis.

$x \notin (\Delta_1, \Delta_2)$ (5)

since by (1) x cannot be in Δ_2.

Thus, we conclude.

Case (T:SUBSUMPTION) - We have:

$\Gamma \mid \Delta_0, x : A \vdash e : A_1 \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(e)$ (2)

by hypothesis.

$\Gamma \vdash \Delta_0, x : A <: \Delta_0', x : A'$ (3)
$\Gamma \mid \Delta_0' \vdash e : A_0 \dashv \Delta_1'$ (4)
$\Gamma \vdash A_0 <: A_1$ (5)
$\Gamma \vdash \Delta_1' <: \Delta_1$ (6)

by inversion on (T:SUBSUMPTION) with (1).

$x \notin \Delta_1'$ (7)

by induction hypothesis on (2) and (4).

$x \notin \Delta_1$ (8)

by (6) and (7) noting the members of Δ_1 and Δ_1' are the same.

Thus, we conclude.

Case (T:TAG) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash \mathtt{t}\#v : A_1 \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(\mathtt{t}\#v)$ (2)

by hypothesis.

$\Gamma \mid \Delta_0, x : A_0 \vdash v : A_1 \dashv \Delta_1$ (3)

by inversion on (T:TAG) with (1).

[$\mathtt{fv}(\mathtt{t}\#v) = \mathtt{fv}(v)$]

$x \in \mathtt{fv}(e)$ (4)

by definition of $\mathtt{fv}$ and (2).

$x \notin \Delta_1$ (5)

by induction hypothesis on (3) and (4).

Thus, we conclude.

Case (T:CASE) - We have:

$\Gamma \mid \Delta_0, x : A' \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_1$ (1)
$x \in \mathtt{fv}(\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end})$ (2)

[$\mathtt{fv}(\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end}) = \mathtt{fv}(v) \cup \overline{\mathtt{fv}(e_i)}$], for some $i \leq j$

by hypothesis.

$\Gamma \mid \Delta_0, x : A' \vdash v : \sum_i \mathtt{t}_i\#A_i \dashv \Delta'$ (3)
$\overline{\Gamma \mid \Delta', x_i : A_i \vdash e_i : A \dashv \Delta_1}$ (4)
$i \leq j$ (5)

by inversion on (T:CASE) with (1).

Therefore, we have the following possibilities:

1. $x \in \mathtt{fv}(v) \wedge x \notin \overline{\mathtt{fv}(e_i)}$

   $x \notin \Delta'$ (1.1)

   by induction hypothesis on (3) and case assumption.

   $x \notin \Delta_1$ (1.2)

   by (1.1) and (4).

   Thus, we conclude.

2. $x \notin \mathtt{fv}(v) \wedge x \in \overline{\mathtt{fv}(e_i)}$

   $(x : A') \in \Delta'$ (2.1)

   by $x \notin \mathtt{fv}(e)$.

   $x \notin \Delta_1$ (2.2)

   by induction hypothesis on (4) and (2.1).

   Thus, we conclude.

3. $x \in \mathtt{fv}(v) \wedge x \in \overline{\mathtt{fv}(e_i)}$

   $x \notin \Delta_1$ (3.1)

   by induction hypothesis on (3) and sub-case hypothesis.

   We reach a contradiction since v is well-typed by (4) but $x \in \overline{\mathtt{fv}(e_i)}$ contradicts (3.1). Thus, such case is impossible to occur in a well-typed expression.

Case (T:ALTERNATIVE-LEFT) - We have:

$\Gamma \mid \Delta_0, x : A_0, A_1 \oplus A_2 \vdash e : A_3 \dashv \Delta_1$ (1)

$x \in \mathtt{fv}(e)$ (2)

by hypothesis.

$\Gamma \mid \Delta_0, x : A_0, A_1 \vdash e : A_3 \dashv \Delta_1$ (3)

$\Gamma \mid \Delta_0, x : A_0, A_2 \vdash e : A_3 \dashv \Delta_1$ (4)

by inversion on (T:ALTERNATIVE-LEFT) with (1).

$x \notin \Delta_1$ (5)

by induction hypothesis with (2) and (3).

Thus, we conclude.

Case (T:INTERSECTION-RIGHT) - Analogous to previous case but using (T:INTERSECTION-RIGHT).

Case (T:LET) - We have:

$\Gamma \mid \Delta_0, x : A \vdash \mathsf{let}\ x_0 = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_2$ (1)

$x \in \mathtt{fv}(\mathsf{let}\ x_0 = e_0\ \mathsf{in}\ e_1\ \mathsf{end})$ (2)

[$\mathtt{fv}(\mathsf{let}\ x_0 = e_0\ \mathsf{in}\ e_1\ \mathsf{end}) = \mathtt{fv}(e_0) \cup \mathtt{fv}(e_1)$]

by hypothesis.

$\Gamma \mid \Delta_0, x : A \vdash e_0 : A_0 \dashv \Delta_1$ (3)

$\Gamma \mid \Delta_1, x_0 : A_0 \vdash e_1 : A_1 \dashv \Delta_2$ (4)

by inversion on (T:LET) with (1).

Therefore, we have the following possibilities:

1. $x \in \mathtt{fv}(e_1) \land x \notin \mathtt{fv}(e_0)$

   $(x : A) \in \Delta_1$ (1.1)

   by $x \notin \mathtt{fv}(e_0)$.

   $x \notin \Delta_2$ (1.2)

   by induction hypothesis on (4) with (1.1).

   Thus, we conclude.

2. $x \in \mathtt{fv}(e_0) \land x \in \mathtt{fv}(e_1)$

   $x \notin \Delta_1$ (2.1)

   by induction hypothesis on (3) and case assumption.

   We reach a contradiction since e_0 is well-typed by (4) but $x \in \mathtt{fv}(e_1)$ contradicts (2.1). Thus, such case is impossible to occur in a well-typed expression.

3. $x \in \mathtt{fv}(e_0) \land x \notin \mathtt{fv}(e_1)$

   $x \notin \Delta_1$ (3.1)

   by induction hypothesis on (3) and case assumption.

   $x \notin \Delta_2$ (3.2)

   by (3.1) and (4).

   Thus, we conclude.

Case (T:FORK) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash \mathsf{fork}\ e : ![] \dashv \cdot$ (1)

$x \in \mathtt{fv}(e)$ (2)

by hypothesis.

$x \notin \cdot$ (3)

since it is the empty environment.

Thus, we conclude.

Case (T:LOCK-RELY) - We have:

$\Gamma \mid \Delta_0, x : A_0, A_1 \Rightarrow A_2 \vdash \mathsf{lock}\ \overline{v} : ![] \dashv \Delta_0, A_1, A_2$ (1)

$x \in \mathtt{fv}(\mathsf{lock}\ \overline{v})$ (2)

by hypothesis.

$\overline{\Gamma \mid \cdot \vdash v : \mathbf{ref}\ p \dashv \cdot}$ (3)

$\overline{p} \in A_0$ (4)

by inversion on (1).

We have a contradiction of (3) with (2) since x cannot be in v as the linear environment is empty.

Thus, we conclude since this case cannot occur.

Case (T:UNLOCK-GUARANTEE) - Analogous to the previous case.

Case (T:FORALL-LOC-VAL) - We have:

$\Gamma \mid \Delta_0, x : A_0 \vdash v : A_1 \dashv \cdot$ (1)

$x \in \mathtt{fv}(v)$ (2)

by hypothesis.

$x \notin \cdot$ (3)

since it is the empty environment.

Thus, we conclude.

Case (T:FORALL-TYPE-VAL) - Analogous to the previous case.

Cases (T:TYPEOPENBIND), (T:TYPEOPENCAP), (T:LOCOPENCAP), (T:LOCOPENBIND) - Analogous to the previous case by inversion on the typing rule and then applying the induction hypothesis.

□

A.2.5 Well-Form Lemmas

Lemma 21 (Well-Formed Type Substitution)**.** We have:

- For *location variables*:
  1. If $\Gamma, l : \mathbf{loc}\ \mathbf{wf}$ and $\rho : \mathbf{loc} \in \Gamma$ then $\Gamma\{\rho/l\}\ \mathbf{wf}$.
  2. If $\Gamma, l : \mathbf{loc} \vdash \Delta\ \mathbf{wf}$ and $\rho : \mathbf{loc} \in \Gamma$ then $\Gamma\{\rho/l\} \vdash \Delta\{\rho/l\}\ \mathbf{wf}$.
  3. If $\Gamma, l : \mathbf{loc} \vdash A\ \mathbf{type}$ and $\rho : \mathbf{loc} \in \Gamma$ then $\Gamma\{\rho/l\} \vdash A\{\rho/l\}\ \mathbf{type}$.
- For *type variables*:
  1. If $\Gamma, X <: A_0\ \mathbf{wf}$ and $\Gamma \vdash A_1\ \mathbf{type}$ and $\Gamma \vdash A_1 <: A_0$ then $\Gamma\{A_1/X\}\ \mathbf{wf}$.
  2. If $\Gamma, X <: A_0 \vdash \Delta\ \mathbf{wf}$ and $\Gamma \vdash A_1\ \mathbf{type}$ and $\Gamma \vdash A_1 <: A_0$ then $\Gamma\{A_1/X\} \vdash \Delta\{A_1/X\}\ \mathbf{wf}$.
  3. If $\Gamma, X <: A_0 \vdash A\ \mathbf{type}$ and $\Gamma \vdash A_1\ \mathbf{type}$ and $\Gamma \vdash A_1 <: A_0$ then $\Gamma\{A_1/X\} \vdash A\{A_1/X\}\ \mathbf{type}$.

Proof. Straightforward by induction on the structure of Γ, Δ and types. □

Lemma 22 (Well-Formed Subtyping)**.** We have two cases:

1. (Type) If $\Gamma \vdash A\ \mathbf{type}$ and $\Gamma \vdash A <: A'$ then $\Gamma \vdash A'\ \mathbf{type}$.
2. (Delta) If $\Gamma \vdash \Delta\ \mathbf{wf}$ and $\Gamma \vdash \Delta <: \Delta'$ then $\Gamma \vdash \Delta'\ \mathbf{wf}$.

Proof. Straightforward by induction on the definition of <: for types and linear typing environments, respectively. □

A.2.6 Substitution Lemma

Lemma 23 (Substitution Lemma)**.** We have the following substitution properties for both *expression typing* and *type formation*:

1. (Linear) If
$$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2$$
then
$$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A_1 \dashv \Delta_2$$
2. (Pure) If
$$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot \qquad \Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1$$
then
$$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A_1 \dashv \Delta_1$$
(note that due to the required pure types, the Δ environments to check v must be empty)
3. (Location Variable) If
$$\Gamma, l : \mathbf{loc} \mid \Delta_0 \vdash e : A \dashv \Delta_1 \qquad \rho : \mathbf{loc} \in \Gamma$$
then
$$\Gamma\{\rho/l\} \mid \Delta_0\{\rho/l\} \vdash e : A\{\rho/l\} \dashv \Delta_1\{\rho/l\}$$
4. (Type Variable) If
$$\Gamma, X : \mathbf{type}, X <: A_2 \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1 \qquad \Gamma \vdash A_1 \ \mathbf{type} \qquad \Gamma \vdash A_1 <: A_2$$
then
$$\Gamma\{A_1/X\} \mid \Delta_0\{A_1/X\} \vdash e : A_0\{A_1/X\} \dashv \Delta_1\{A_1/X\}$$

Proof. We split the proof on each of the lemma's sub-parts:

1. (Linear)

   Proof. We proceed by induction on the typing derivation of $\Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2$.

   Case (T:Ref), (T:Pure), (T:Unit), (T:Pure-Read) - Not applicable due to empty Δ environment.

   Case (T:Linear-Read) - We have:

   $\Gamma \mid \Delta \vdash v : A \dashv \cdot$ (1)

   $\Gamma \mid x : A \vdash x : A \dashv \cdot$ (2)

   by hypothesis.

   (note v's ending environment must be $\cdot$ to apply (T:Linear-Read)).

   $\Gamma \mid \Delta \vdash x\{v/x\} : A \dashv \cdot$ (3)

   by (vs:2) with (1) and x.

   Thus, we conclude.

Case (T:PURE-ELIM) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x_1 : !A_2, x_0 : A_0 \vdash e : A_1 \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma, x_1 : A_2 \mid \Delta_1, x_0 : A_0 \vdash e : A_1 \dashv \Delta_2$ (3)

by inversion on (T:PURE-ELIM) with (2).

$\Gamma, x_1 : A_2 \mid \Delta_1 \vdash e\{v/x_0\} : A_1 \dashv \Delta_2$ (4)

by induction hypothesis on (3) with (1).

$\Gamma \mid \Delta_1, x_1 : !A_2 \vdash e\{v/x_0\} : A_1 \dashv \Delta_2$ (5)

by (T:PURE-ELIM) with (4).

Thus, we conclude.

Case (T:NEW) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash \mathsf{new}\ v_0 : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 : A_1 \dashv \Delta_2$ (3)

by inversion on (T:NEW) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : A_1 \dashv \Delta_2$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v_0\{v/x\} : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_2$ (5)

by (T:NEW) with (4).

$\Gamma \mid \Delta_0 \vdash (\mathsf{new}\ v_0)\{v/x\} : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_2$ (6)

by (VS:8) with (5).

Thus, we conclude.

Case (T:DELETE) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash \mathsf{delete}\ v_0 : \exists l.A_1 \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_2$ (3)

by inversion on (T:DELETE) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_2$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v_0\{v/x\} : \exists l.A_1 \dashv \Delta_2$ (5)

by (T:DELETE) with (4).

$\Gamma \mid \Delta_0 \vdash (\mathsf{delete}\ v_0)\{v/x\} : \exists l.A_1 \dashv \Delta_2$ (6)

by (VS:9) with (5).

Thus, we conclude.

Case (T:ASSIGN) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_1 : A_2 \dashv \Delta'$ (3)

$\Gamma \mid \Delta' \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (4)

by inversion on (T:Assign) with (2).

We have that either:

(a) $x \in \mathtt{fv}(v_1)$

$x \notin \Delta'$ (1.1)

by (Free Variables) on (3).

$\Gamma \mid \Delta' \vdash v_0\{v/x\} : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (1.2)

since x cannot occur in e_0 by (1.1).

$\Gamma \mid \Delta_1 \vdash v_1\{v/x\} : A_2 \dashv \Delta'$ (1.3)

by induction hypothesis on (1) and (3).

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} := v_1\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (1.4)

by (T:Assign) on (1.2) and (1.3).

$\Gamma \mid \Delta_1 \vdash (v_0 := v_1)\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (1.5)

by (vs:11) on (1.4).

Thus, we conclude.

(b) $x \notin \mathtt{fv}(v_1)$

$(x : A_0) \in \Delta'$ (2.1)

by (9) and $x \notin \mathtt{fv}(v_1)$.

$\Gamma \mid \Delta'' \vdash v_0\{v/x\} : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (2.2)

by induction hypothesis (since it is applied to x wherever is in the environment) and where Δ'' is the same as Δ' without x.

$\Gamma \mid \Delta_1 \vdash v_1\{v/x\} : A_2 \dashv \Delta''$ (2.3)

since x cannot occur in e_1 by $x \notin \mathtt{fv}(e_1)$.

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} := v_1\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (2.4)

by (T:Assign) using (2.4) and (2.5).

$\Gamma \mid \Delta_1 \vdash (v_0 := v_1)\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (2.5)

by (vs:11) on (2.6).

Thus, we conclude.

Case (T:Dereference-Linear) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash !v_0 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ ![]$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (3)

by inversion on (T:Dereference-Linear) on (2).

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash !v_0\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ ![]$ (5)

by (T:Dereference-Linear) on (4).

$\Gamma \mid \Delta_1 \vdash (!v_0)\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ ![]$ (6)

by (vs:10) on (5).

Thus, we conclude.

Case (T:Dereference-Pure) - Analogous to (T:Dereference-Linear).

Case (T:Record) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash \{\overline{\mathtt{f} = v'}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (2)

by hypothesis.

$\overline{\Gamma \mid \Delta_1, x : A_0 \vdash v'_i : A_i \dashv \cdot}$ (3)

by inversion with (T:Record) on (2).

$\overline{\Gamma \mid \Delta_1 \vdash v'_i\{v/x\} : A_i \dashv \cdot}$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash \{\overline{\mathtt{f} = v'\{v/x\}}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (5)

by (T:Record) on (4).

$\Gamma \mid \Delta_1 \vdash (\{\overline{\mathtt{f} = v'}\})\{v/x\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (6)

by (vs:5) on (5).

Thus, we conclude.

Case (T:Selection) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0.\mathtt{f} : A_1 \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 : [\mathtt{f} : A_1] \dashv \Delta_2$ (3)

by inversion on (T:Selection) with (2).

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} : [\mathtt{f} : A_1] \dashv \Delta_2$ (4)

by induction hypothesis on (3) with (1).

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\}.\mathtt{f} : [\mathtt{f} : A_1] \dashv \Delta_2$ (5)

by (T:Selection) on (4).

$\Gamma \mid \Delta_1 \vdash (v_0.\mathtt{f})\{v/x\} : [\mathtt{f} : A_1] \dashv \Delta_2$ (6)

by (vs:6) on (5).

Thus, we conclude.

Case (T:Application) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 : A_2 \multimap A_1 \dashv \Delta'$ (3)

$\Gamma \mid \Delta' \vdash v_1 : A_2 \dashv \Delta_2$ (4)

by inversion on (T:Application) with (2).

We have that either:

(a) $x \in \mathtt{fv}(v_0)$

$x \notin \Delta'$ (1.1)

by (Free Variables) on (3).

$\Gamma \mid \Delta' \vdash v_1\{v/x\} : A_2 \dashv \Delta_2$ (1.2)

since x cannot occur in v_1 by (1.1).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : A_2 \multimap A_1 \dashv \Delta'$ (1.3)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\}\ v_1\{v/x\} : A_1 \dashv \Delta_2$ (1.4)

by (T:Application) with (1.2) and (1.3).

$\Gamma \mid \Delta_0 \vdash (v_0\ v_1)\{v/x\} : A_1 \dashv \Delta_2$ (1.5)

by (vs:7) on (1.4).

Thus, we conclude.

(b) $x \notin \mathtt{fv}(v_0)$

$(x : A_0) \in \Delta'$ (2.1)

by $x \notin \mathtt{fv}(v_1)$.

$\Gamma \mid \Delta'' \vdash v_1\{v/x\} : A_2 \dashv \Delta_2$ (2.2)

by induction hypothesis where Δ'' is Δ' without x.

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : A_2 \multimap A_1 \dashv \Delta''$ (2.3)

since x cannot occur in v_0 by $x \notin \mathtt{fv}(v_0)$ and (2.1).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\}\ v_1\{v/x\} : A_1 \dashv \Delta_2$ (2.4)

by (T:Application) on (2.2) and (2.3).

$\Gamma \mid \Delta_0 \vdash (v_0\ v_1)\{v/x\} : A_1 \dashv \Delta_2$ (2.5)

by (vs:7) on (2.4).

Thus, we conclude.

Case (T:Function) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x_0 : A_0 \vdash \lambda x_1.e : A_1 \multimap A_2 \dashv \cdot$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x_1 : A_1, x_0 : A_0 \vdash e : A_2 \dashv \cdot$ (3)

$x_1 \neq x_0$ (4)

by def. of substitution up to rename of bounded variables.

$\Gamma \mid \Delta_1, x_1 : A_1 \vdash e\{v/x\} : A_2 \dashv \cdot$ (5)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash \lambda x_1.e\{v/x\} : A_1 \multimap A_2 \dashv \cdot$ (6)

by (T:Function) with (5).

$\Gamma \mid \Delta_1 \vdash (\lambda x_1.e)\{v/x\} : A_1 \multimap A_2 \dashv \cdot$ (7)

by (vs:4) on (6) and (4).

Thus, we conclude.

Case (T:CAP-ELIM) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1, x_1 : A_2 :: A_3$ (1)

$\Gamma \mid \Delta_1, x_1 : A_2 :: A_3, x_0 : A_0 \vdash e : A_1 \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x_1 : A_2, A_3, x_0 : A_0 \vdash e : A_1 \dashv \Delta_2$ (3)

by inversion on (T:CAP-ELIM) with (2).

$\Gamma \mid \Delta_1, x_1 : A_2, A_3 \vdash e\{v/x_0\} : A_1 \dashv \Delta_2$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1, x_1 : A_2 :: A_3 \vdash e\{v/x_0\} : A_1 \dashv \Delta_2$ (5)

by (T:CAP-ELIM) with (4).

Thus, we conclude.

Case (T:CAP-STACK) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 :: A_2 \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2, A_2$ (3)

by inversion on (T:CAP-STACK) with (2).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : A_1 \dashv \Delta_2, A_2$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : A_1 :: A_2 \dashv \Delta_2$ (5)

by (T:CAP-STACK) on (4).

Thus, we conclude.

Case (T:CAP-UNSTACK) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2, A_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 :: A_2 \dashv \Delta_2$ (3)

by inversion (T:CAP-UNSTACK) with (2).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : A_1 :: A_2 \dashv \Delta_2$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : A_1 \dashv \Delta_2, A_2$ (5)

by (T:CAP-UNSTACK) with (4).

Thus, we conclude.

Case (T:SUBSUMPTION) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \vdash \Delta_1, x : A_0 <: \Delta_1', x : A_0'$ (3)

$\Gamma \mid \Delta_1', x : A_0' \vdash e : A_2 \dashv \Delta_2'$ (4)
$\Gamma \vdash A_2 <: A_1$ (5)
$\Gamma \vdash \Delta_2' <: \Delta_2$ (6)
by inversion on (T:SUBSUMPTION) on (2).

$\Gamma \vdash A_0 <: A_0'$ (7)
by (Subtyping Inversion) on (3) on x.

$\Gamma \mid \Delta_0 \vdash v : A_0' \dashv \Delta_1'$ (8)
by (T:SUBSUMPTION) on (1) with (7).

$\Gamma \mid \Delta_1' \vdash e\{v/x\} : A_2 \dashv \Delta_2'$ (9)
by induction hypothesis on (4) and (8).

$\Gamma \vdash \Delta_1 <: \Delta_1'$ (10)
by (Subtyping Inversion) on (3).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : A_1 \dashv \Delta_2$ (11)
by (T:SUBSUMPTION) on (9) with (10), (5) and (6).

Thus, we conclude.

Case (T:FRAME) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)
$\Gamma \mid (\Delta_1, x : A_0), \Delta_3 \vdash e : A_1 \dashv \Delta_2, \Delta_3$ (2)
by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2$ (3)
by inversion on (T:FRAME) with (2).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : A_1 \dashv \Delta_2$ (4)
by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1, \Delta_3 \vdash e\{v/x\} : A_1 \dashv \Delta_2, \Delta_3$ (5)
by (T:FRAME) on (4) with Δ_3.

Thus, we conclude.

Case (T:TAG) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)
$\Gamma \mid \Delta_1, x : A_0 \vdash \mathtt{t}\#v_0 : \mathtt{t}\#A_1 \dashv \Delta_2$ (2)
by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 : A_1 \dashv \Delta_2$ (3)
by inversion (T:TAG) with (2).

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} : A_1 \dashv \Delta_2$ (4)
by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash \mathtt{t}\#v_0\{v/x\} : \mathtt{t}\#A_1 \dashv \Delta_2$ (5)
by (T:TAG) with (4).

$\Gamma \mid \Delta_1 \vdash (\mathtt{t}\#v_0)\{v/x\} : \mathtt{t}\#A_1 \dashv \Delta_2$ (6)
by (VS:12) on (5).

Thus, we conclude.

Case (T:CASE) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash \mathsf{case}\ v_0\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_2$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0 : \sum_i \mathtt{t}_i \# A'_i \dashv \Delta'$ (3)

$\overline{\Gamma \mid \Delta', x_i : A'_i \vdash e_i : A \dashv \Delta_2}$ (4)

$i \leq j$ (5)

by inversion (T:CASE) with (2).

We have that either:

(a) $x \in \mathtt{fv}(v_0)$

$x \notin \Delta'$ (1.1)

by (Free Variables) on (3).

$\overline{x \neq x_j}$ (1.2)

by def. of substitution up to rename of bounded variables.

$\overline{\Gamma \mid \Delta', x_i : A'_i \vdash e_i\{v/x\} : A \dashv \Delta_2}$ (1.3)

since x cannot occur in e_i and by (1.1) nor in Γ by (3).

$\Gamma \mid \Delta_1, x : A_0 \vdash v_0\{v/x\} : \sum_i \mathtt{t}_i \# A'_i \dashv \Delta'$ (1.4)

by induction hypothesis on (1) and (3).

$\Gamma \mid \Delta_1 \vdash \mathsf{case}\ v_0\{v/x\}\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j\{v/x\}}\ \mathsf{end} : A \dashv \Delta_2$ (1.5)

by (T:CASE) on (5), (1.3) and (1.4).

$\Gamma \mid \Delta_1 \vdash (\mathsf{case}\ v_0\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end})\{v/x\} : A \dashv \Delta_2$ (1.6)

by (vs:13) on (1.6) and (1.2).

Thus, we conclude.

(b) $x \notin \mathtt{fv}(v_0)$

$(x : A_0) \in \Delta'$ (2.1)

by $x \notin \mathtt{fv}(e)$.

$\overline{x \neq x_j}$ (2.2)

by def. of substitution up to rename of bounded variables.

$\overline{\Gamma \mid \Delta'', x_i : A'_i \vdash e_i\{v/x\} : A \dashv \Delta_2}$ (2.3)

by induction hypothesis where Δ'' is same as Δ' without x.

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} : \sum_i \mathtt{t}_i \# A'_i \dashv \Delta''$ (2.4)

since x cannot occur in e by $x \notin \mathtt{fv}(e)$.

$\Gamma \mid \Delta_1 \vdash \mathsf{case}\ v_0\{v/x\}\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j\{v/x\}}\ \mathsf{end} : A \dashv \Delta_2$ (2.5)

by (T:CASE) on (5), (2.3) and (2.4).

$\Gamma \mid \Delta_1 \vdash (\mathsf{case}\ v_0\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end})\{v/x\} : A \dashv \Delta_2$ (2.6)

by (vs:13) on (2.1) and (2.5).

Thus, we conclude.

Case (T:LET) - Analogous to previous cases. First, we consider the different sub-cases where x may or not appear. If x appears on the first expression we just apply the induction

hypothesis there. Otherwise, we apply the induction hypothesis on the body of the let. Finally, we use the substitution definition (vs:14) to "push" the substitution outside.

Cases (T:ALTERNATIVE-LEFT), (T:INTERSECTION-RIGHT), (T:FORALL-LOC-VAL), (T:FORALL-TYPE-VAL), (T:TYPEOPENBIND), (T:TYPEOPENCAP), (T:LOCOPENCAP), (T:LOCOPENBIND) - immediate by applying the induction hypothesis on the inversion and then re-applying the rule.

Case (T:FORK) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1$ (1)

$\Gamma \mid \Delta_1, x : A_0 \vdash \mathsf{fork}\ e : ![] \dashv \cdot$ (2)

by hypothesis.

$\Gamma \mid \Delta_1, x : A_0 \vdash e : ![] \dashv \cdot$ (3)

by inversion (T:FORK) with (2).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : ![] \dashv \cdot$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash \mathsf{fork}\ e\{v/x\} : ![] \dashv \cdot$ (5)

by (T:FORK) with (4).

$\Gamma \mid \Delta_1 \vdash (\mathsf{fork}\ e)\{v/x\} : ![] \dashv \cdot$ (6)

by (vs:17) on (5).

Thus, we conclude.

Case (T:LOCK-RELY), (T:UNLOCK-GUARANTEE) - immediate since x cannot occur in "$\mathsf{lock}\ \overline{v}$" nor "$\mathsf{unlock}\ \overline{v}$" as all those values ($\overline{v}$) must be typed without linear resources. Thus, in this case substitution is vacuously true.

□

2. (Pure)

Proof. We proceed by induction on the typing derivation of $\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1$.

Case (T:REF) - We have:

$\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash v_0 : !A_0 \dashv \cdot$ (1)

$\Gamma, \rho : \mathbf{loc}, x : A_0 \mid \cdot \vdash \rho : \mathbf{ref}\ \rho \dashv \cdot$ (2)

by hypothesis.

$\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\ \rho \dashv \cdot$ (3)

by $x \notin \mathtt{fv}(\rho)$ on (2).

$\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho\{v/x\} : \mathbf{ref}\ \rho \dashv \cdot$ (4)

by (vs:1) on (3) using x and v.

Thus, we conclude.

Case (T:PURE) - We have:

$\Gamma \mid \cdot \vdash v_0 : !A_0 \dashv \cdot$ (1)

$\Gamma, x_0 : A_0 \mid \cdot \vdash v_1 : !A_1 \dashv \cdot$ (2)

by hypothesis.

$\Gamma, x_0 : A_0 \mid \cdot \vdash v_1 : A_1 \dashv \cdot$ (3)

by inversion on (T:PURE) with (2).

$\Gamma \mid x_0 : !A_0 \vdash v_1 : A_1 \dashv \cdot$ (4)

by (T:PURE-ELIM) on (3) with x_0.

$\Gamma \mid \cdot \vdash v_1\{v_0/x_0\} : A_1 \dashv \cdot$ (5)

by (Substitution Lemma - Linear) with (1) and (4).

$\Gamma \mid \cdot \vdash v_1\{v_0/x_0\} : !A_1 \dashv \cdot$ (6)

by (T:PURE) on (5).

Thus, we conclude.

Case (T:UNIT) - We have:

$\Gamma \mid \cdot \vdash v_0 : !A_0 \dashv \cdot$ (1)

$\Gamma, x : A_0 \mid \cdot \vdash v_1 : ![] \dashv \cdot$ (2)

by hypothesis.

$\Gamma \mid \cdot \vdash v_1\{v_0/x\} : ![] \dashv \cdot$ (3)

substitution on x cannot change the type since [] is always valid by (T:UNIT).
(and substitution cannot change a value to become an expression).

Thus, we conclude.

Case (T:PURE-READ) - We have:

$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot$ (1)

$\Gamma, x_0 : A_0 \mid \cdot \vdash x_1 : !A_1 \dashv \cdot$ (2)

by hypothesis (matching environments and type with (T:PURE-READ)).

We have that either:

(a) $x_0 = x_1$

$\Gamma \mid \cdot \vdash v : !A \dashv \cdot$ (1.1)

$\Gamma, x : A \mid \cdot \vdash x : !A \dashv \cdot$ (1.2)

by restated hypothesis with $x = x_0 = x_1$.
and with $A = A_0 = A_1$.

$\Gamma \mid \cdot \vdash x\{v/x\} : !A \dashv \cdot$ (1.3)

by (vs:2) on (1.1) using x and v.

Thus, we conclude.

(b) $x_0 \neq x_1$

$\Gamma \mid \cdot \vdash x_1 : !A_1 \dashv \cdot$ (2.1)

by $x_0 \notin \mathtt{fv}(x_1)$ on (2).

$\Gamma \mid \cdot \vdash x_1\{v/x_0\} : !A_1 \dashv \cdot$ (2.2)

by (vs:3) on (2.1) using x_0 and v.

Thus, we conclude.

Case (T:LINEAR-READ) - We have:

$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot$ (1)

$\Gamma, x_0 : A_0 \mid x_1 : A_1 \vdash x_1 : A_1 \dashv \cdot$ (2)

by hypothesis.

$x_0 \neq x_1$ (3)

since Γ and Δ identifiers cannot collide.

$\Gamma \mid x_1 : A_1 \vdash x_1\{v/x_0\} : A_1 \dashv \cdot$ (4)

by (vs:3) on (2) using x_0 and v.

Thus, we conclude.

Case (T:PURE-ELIM) - We have:

$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot$ (1)

$\Gamma, x_0 : A_0 \mid \Delta_0, x_1 : !A_2 \vdash e : A_1 \dashv \Delta_1$ (2)

by hypothesis.

$\Gamma, x_0 : A_0, x_1 : A_2 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1$ (3)

by inversion on (T:PURE-ELIM) with (2)

$\Gamma, x_1 : A_2 \mid \Delta_0 \vdash e\{v/x_0\} : A_1 \dashv \Delta_1$ (4)

by induction hypothesis on (1) with (3).

$\Gamma \mid \Delta_0, x_1 : !A_2 \vdash e\{v/x_0\} : A_1 \dashv \Delta_1$ (5)

by (T:PURE-ELIM) on (4).

Thus, we conclude.

Case (T:NEW) - We have:

$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot$ (1)

$\Gamma, x : A_0 \mid \Delta_0 \vdash \mathsf{new}\ v_0 : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_1$ (2)

by hypothesis.

$\Gamma, x : A_0 \mid \Delta_0 \vdash v_0 : A_1 \dashv \Delta_1$ (3)

by inversion on (T:NEW) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : A_1 \dashv \Delta_1$ (4)

by induction hypothesis with (3) and (1).

$\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v_0\{v/x\} : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_1$ (5)

by (T:NEW) with (4).

$\Gamma \mid \Delta_0 \vdash (\mathsf{new}\ v_0)\{v/x\} : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_1$ (6)

by (vs:8) on (5).

Thus, we conclude.

Case (T:DELETE) - We have:

$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot$ (1)

$\Gamma, x : A_0 \mid \Delta_0 \vdash \mathsf{delete}\ v_0 : \exists l.A_1 \dashv \Delta_1$ (2)

by hypothesis.

$\Gamma, x : A_0 \mid \Delta_0 \vdash v_0 : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_1$ (3)

by inversion on (T:DELETE) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A_1) \dashv \Delta_1$ (4)

by induction hypothesis with (3) and (1).

$\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v_0\{v/x\} : \exists l.A_1 \dashv \Delta_1$ (5)

by (T:DELETE) with (4).

$\Gamma \mid \Delta_0 \vdash (\mathsf{delete}\ v_0)\{v/x\} : \exists l.A_1 \dashv \Delta_1$ (6)

by (vs:9) on (5).

Thus, we conclude.

Case (T:ASSIGN) - We have:

$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot$ (1)

$\Gamma, x : A_0 \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (2)

by hypothesis.

$\Gamma, x : A_0 \mid \Delta_0 \vdash v_1 : A_2 \dashv \Delta_1$ (3)

$\Gamma, x : A_0 \mid \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (4)

by inversion on (T:ASSIGN) with (2).

$\Gamma \mid \Delta_0 \vdash v_1\{v/x\} : A_2 \dashv \Delta_1$ (5)

by induction hypothesis on (3) with (1).

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1$ (6)

by induction hypothesis on (4) with (1).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} := v_1\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (7)

by (T:ASSIGN) with (5) and (6).

$\Gamma \mid \Delta_0 \vdash (v_0 := v_1)\{v/x\} : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_2$ (8)

by (vs:11) on (7).

Thus, we conclude.

Case (T:DEREFERENCE-LINEAR) - We have:

$\Gamma \mid \cdot \vdash v : !A_0 \dashv \cdot$ (1)

$\Gamma, x : A_0 \mid \Delta_0 \vdash !v_0 : A_1 \dashv \Delta_1, \mathbf{rw}\ p\ ![]$ (2)

by hypothesis.

$\Gamma, x : A_0 \mid \Delta_0 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A_1$ (3)

by inversion on (T:DEREFERENCE-LINEAR) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A_1$ (4)

by induction hypothesis on (3) with (1).

$\Gamma \mid \Delta_0 \vdash !v_0\{v/x\} : A_1 \dashv \Delta_1, \mathbf{rw}\ p\ ![]$ (5)

by (T:DEREFERENCE-LINEAR) with (4).

$\Gamma \mid \Delta_0 \vdash (!v_0)\{v/x\} : A_1 \dashv \Delta_1, \mathbf{rw}\ p\ ![]$ (6)

by (vs:10) on (5).

Thus, we conclude.

Case (T:DEREFERENCE-PURE) - Analogous to (T:DEREFERENCE-LINEAR).

Case (T:RECORD) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta \vdash \{\overline{\mathtt{f} = v'}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (2)
by hypothesis.

$\overline{\Gamma, x : A' \mid \Delta \vdash v'_i : A_i \dashv \cdot}$ (3)
by inversion on (T:RECORD) with (2).

$\overline{\Gamma \mid \Delta \vdash v'_i\{v/x\} : A_i \dashv \cdot}$ (4)
by induction hypothesis on (3) with (1).

$\Gamma \mid \Delta \vdash \{\overline{\mathtt{f} = v'\{v/x\}}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (5)
by (T:RECORD) on (4).

$\Gamma \mid \Delta \vdash (\{\overline{\mathtt{f} = v'}\})\{v/x\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (6)
by (vs:5) on (5).

Thus, we conclude.

Case (T:SELECTION) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)
$\Gamma, x : A' \mid \Delta_0 \vdash v_0.\mathtt{f} : A \dashv \Delta_1$ (2)
by hypothesis.

$\Gamma, x : A' \mid \Delta_0 \vdash v_0 : [\mathtt{f} : A] \dashv \Delta_1$ (3)
by inversion on (T:SELECTION) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : [\mathtt{f} : A] \dashv \Delta_1$ (4)
by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\}.\mathtt{f} : A \dashv \Delta_1$ (5)
by (T:SELECTION) with (4).

$\Gamma \mid \Delta_0 \vdash (v_0.\mathtt{f})\{v/x\} : A \dashv \Delta_1$ (6)
by (vs:6) on (5).

Thus, we conclude.

Case (T:APPLICATION) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)
$\Gamma, x : A' \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2$ (2)
by hypothesis.

$\Gamma, x : A' \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1$ (3)
$\Gamma, x : A' \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2$ (4)
by inversion on (T:APPLICATION) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : A_0 \multimap A_1 \dashv \Delta_1$ (5)
by induction hypothesis with (1) on (3).

$\Gamma \mid \Delta_1 \vdash v_1\{v/x\} : A_0 \dashv \Delta_2$ (6)
by induction hypothesis with (1) on (4).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\}\ v_1\{v/x\} : A_1 \dashv \Delta_2$ (7)
by (T:APPLICATION) with (5) and (6).

$\Gamma \mid \Delta_0 \vdash (v_0\ v_1)\{v/x\} : A_1 \dashv \Delta_2$ (8)
by (vs:7) on (7).

Thus, we conclude.

Case (T:FUNCTION) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x_0 : A' \mid \Delta \vdash \lambda x_1.e : A_0 \multimap A_1 \dashv \cdot$ (2)

by hypothesis.

$\Gamma, x_0 : A' \mid \Delta, x_1 : A_0 \vdash e : A_1 \dashv \cdot$ (3)

by inversion on (T:FUNCTION) with (2).

$x_0 \neq x_1$ (4)

by def. of substitution up to rename of bounded variables.

$\Gamma \mid \Delta, x_1 : A_0 \vdash e\{v/x_0\} : A_1 \dashv \cdot$ (5)

by induction hypothesis with (3) and (1).

$\Gamma \mid \Delta \vdash \lambda x_1.e\{v/x_0\} : A_0 \multimap A_1 \dashv \cdot$ (6)

by (T:FUNCTION) with (6).

$\Gamma \mid \Delta \vdash (\lambda x_1.e)\{v/x_0\} : A_0 \multimap A_1 \dashv \cdot$ (7)

by (vs:4) on (6) and (4).

Thus, we conclude.

Case (T:CAP-ELIM) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta_0, x_0 : A_0 :: A_2 \vdash e : A_1 \dashv \Delta_1$ (2)

by hypothesis.

$\Gamma, x : A' \mid \Delta_0, x_0 : A_0, A_2 \vdash e : A_1 \dashv \Delta_1$ (3)

by inversion on (T:CAP-ELIM) with (2).

$\Gamma \mid \Delta_0, x_0 : A_0, A_2 \vdash e\{v/x\} : A_1 \dashv \Delta_1$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0, x_0 : A_0 :: A_2 \vdash e\{v/x\} : A_1 \dashv \Delta_1$ (5)

by (T:CAP-ELIM) with (4).

Thus, we conclude.

Case (T:CAP-STACK) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1$ (2)

by hypothesis.

$\Gamma, x : A' \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1$ (3)

by inversion on (T:CAP-STACK) with (2).

$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A_0 \dashv \Delta_1, A_1$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A_0 :: A_1 \dashv \Delta_1$ (5)

by (T:CAP-STACK) with (4).

Thus, we conclude.

Case (T:CAP-UNSTACK) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)
$\Gamma, x : A' \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1$ (2)
by hypothesis.

$\Gamma, x : A' \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1$ (3)
by inversion on (T:Cap-Unstack) with (2).

$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A_0 :: A_1 \dashv \Delta_1$ (4)
by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A_0 \dashv \Delta_1, A_1$ (5)
by (T:Cap-Unstack) with (4).

Thus, we conclude.

Case (T:Frame) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)
$\Gamma, x : A' \mid \Delta_0, \Delta_2 \vdash e : A \dashv \Delta_1, \Delta_2$ (2)
by hypothesis.

$\Gamma, x : A' \mid \Delta_0 \vdash e : A \dashv \Delta_1$ (3)
by inversion on (T:Frame) with (2).

$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A \dashv \Delta_1$ (4)
by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0, \Delta_2 \vdash e\{v/x\} : A \dashv \Delta_1, \Delta_2$ (5)
by (T:Frame) with Δ_2.

Thus, we conclude.

Case (T:Subsumption) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)
$\Gamma, x : A' \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1$ (2)
by hypothesis.

$\Gamma \vdash \Delta_0 <: \Delta_0'$ (3)
$\Gamma, x : A' \mid \Delta_0' \vdash e : A_0 \dashv \Delta_1'$ (4)
$\Gamma \vdash A_0 <: A_1$ (5)
$\Gamma \vdash \Delta_1' <: \Delta_1$ (6)
by inversion on (T:Subsumption) with (2).

$\Gamma \mid \Delta_0' \vdash e\{v/x\} : A_0 \dashv \Delta_1'$ (7)
by induction hypothesis with (1) and (4).

$\Gamma \mid \Delta_0 \vdash e\{v/x\} : A_1 \dashv \Delta_1$ (8)
by (T:Subsumption) with (7), (3), (5) and (6).

Thus, we conclude.

Case (T:Tag) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)
$\Gamma, x : A' \mid \Delta_0 \vdash \mathtt{t\#}v_0 : \mathtt{t\#}A_1 \dashv \Delta_1$ (2)
by hypothesis.

$\Gamma, x : A' \mid \Delta_0 \vdash v_0 : A_1 \dashv \Delta_1$ (3)

by inversion (T:Tag) with (2).

$\Gamma \mid \Delta_0 \vdash v_0\{v/x\} : A_1 \dashv \Delta_1$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_0 \vdash \mathtt{t}\#v_0\{v/x\} : \mathtt{t}\#A_1 \dashv \Delta_1$ (5)

by (T:Tag) with (4).

$\Gamma \mid \Delta_0 \vdash (\mathtt{t}\#v_0)\{v/x\} : \mathtt{t}\#A_1 \dashv \Delta_1$ (6)

by (vs:12) on (5).

Thus, we conclude.

Case (T:Case) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta_0 \vdash \mathsf{case}\ v_0\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_1$ (2)

by hypothesis.

$\Gamma, x : A' \mid \Delta_1 \vdash v_0 : \sum_i \mathtt{t}_i\#A'_i \dashv \Delta'$ (3)

$\overline{\Gamma, x : A' \mid \Delta', x_i : A'_i \vdash e_i : A \dashv \Delta_2}$ (4)

$i \leq j$ (5)

by inversion (T:Case) with (2).

$\overline{x \neq x_j}$ (6)

by def. of substitution up to rename of bounded variables.

$\Gamma \mid \Delta_1 \vdash v_0\{v/x\} : \sum_i \mathtt{t}_i\#A'_i \dashv \Delta'$ (7)

by induction hypothesis on (3) and (1).

$\overline{\Gamma \mid \Delta', x_i : A'_i \vdash e_i\{v/x\} : A \dashv \Delta_2}$ (8)

by induction hypothesis on (4) and (1).

$\Gamma \mid \Delta_1 \vdash \mathsf{case}\ v_0\{v/x\}\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j\{v/x\}}\ \mathsf{end} : A \dashv \Delta_2$ (9)

by (T:Case) on (5), (7) and (8).

$\Gamma \mid \Delta_1 \vdash (\mathsf{case}\ v_0\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end})\{v/x\} : A \dashv \Delta_2$ (10)

by (vs:13) on (9) and (6).

Thus, we conclude.

Case (T:Alternative-Left), (T:Intersection-Right) - Immediate by applying the induction hypothesis on the inversion and then re-applying the rule.

Case (T:Let) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta_0 \vdash \mathsf{let}\ x_1 = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_1$ (2)

by hypothesis.

$\Gamma, x : A' \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_2$ (3)

$\Gamma, x : A' \mid \Delta_2, x_1 : A_0 \vdash e_1 : A_1 \dashv \Delta_1$ (4)

by inversion on (T:Let) with (2).

$x_0 \neq x_1$ (5)

by def. of substitution up to rename of bounded variables.

$\Gamma \mid \Delta_0 \vdash e_0\{v/x\} : A_0 \dashv \Delta_2$ (6)

by induction hypothesis on (3) and (1).

$\Gamma \mid \Delta_2, x_1 : A_0 \vdash e_1\{v/x\} : A_1 \dashv \Delta_1$ (7)

by induction hypothesis on (4) and (1).

$\Gamma \mid \Delta_0 \vdash \mathsf{let}\ x_1 = e_0\{v/x\}\ \mathsf{in}\ e_1\{v/x\}\ \mathsf{end} : A_1 \dashv \Delta_1$ (8)

by (T:Let) with (6) and (7).

$\Gamma \mid \Delta_0 \vdash (\mathsf{let}\ x_1 = e_0\ \mathsf{in}\ e_1\ \mathsf{end})\{v/x\} : A_1 \dashv \Delta_1$ (9)

by (vs:14) on (8) and (5).

Thus, we conclude.

Cases (T:Alternative-Left), (T:Intersection-Right), (T:Forall-Type-Val), (T:Forall-Loc-Val), (T:TypeOpenBind), (T:TypeOpenCap), (T:LocOpenCap), (T:LocOpenBind) - are immediate by applying the induction hypothesis on the inversion and then re-applying the rule.

Case (T:Fork) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta_1 \vdash \mathsf{fork}\ e : ![] \dashv \cdot$ (2)

by hypothesis.

$\Gamma, x : A' \mid \Delta_1 \vdash e : ![] \dashv \cdot$ (3)

by inversion (T:Fork) with (2).

$\Gamma \mid \Delta_1 \vdash e\{v/x\} : ![] \dashv \cdot$ (4)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta_1 \vdash \mathsf{fork}\ e\{v/x\} : ![] \dashv \cdot$ (5)

by (T:Fork) with (4).

$\Gamma \mid \Delta_1 \vdash (\mathsf{fork}\ e)\{v/x\} : ![] \dashv \cdot$ (6)

by (vs:17) on (5).

Thus, we conclude.

Case (T:Lock-Rely) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta, A_0 \Rightarrow A_1 \vdash \mathsf{lock}\ \overline{v'} : ![] \dashv \Delta, A_0, A_1$ (2)

by hypothesis.

$\overline{\Gamma, x : A' \mid \cdot \vdash v' : \mathbf{ref}\ p \dashv \cdot}$ (3)

$\overline{p} \in A_0$ (4)

by inversion (T:Lock-Rely) with (2).

$\overline{\Gamma \mid \cdot \vdash v'\{v/x\} : ![] \dashv \cdot}$ (5)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta, A_0 \Rightarrow A_1 \vdash \mathsf{lock}\ \overline{v'}\{v/x\} : ![] \dashv \Delta, A_0, A_1$ (6)

by (T:Lock-Rely) with (4) and (5).

$\Gamma \mid \Delta, A_0 \Rightarrow A_1 \vdash (\mathsf{lock}\ \overline{v'})\{v/x\} : ![] \dashv \Delta, A_0, A_1$ (7)

by (vs:15) on (6).

Thus, we conclude.

Case (T:UNLOCK-GUARANTEE) - We have:

$\Gamma \mid \cdot \vdash v : !A' \dashv \cdot$ (1)

$\Gamma, x : A' \mid \Delta, A_0, A_0; A_1 \vdash \mathsf{unlock}\ \overline{v'} : ![] \dashv \Delta, A_1$ (2)

by hypothesis.

$\overline{\Gamma, x : A' \mid \cdot \vdash v' : \mathbf{ref}\ p \dashv \cdot}$ (3)

$\overline{p} \in A_0$ (4)

by inversion (T:UNLOCK-GUARANTEE) with (2).

$\overline{\Gamma \mid \cdot \vdash v'\{v/x\} : ![] \dashv \cdot}$ (5)

by induction hypothesis with (1) and (3).

$\Gamma \mid \Delta, A_0, A_0; A_1 \vdash \mathsf{unlock}\ \overline{v'}\{v/x\} : ![] \dashv \Delta, A_1$ (6)

by (T:LOCK-RELY) with (4) and (5).

$\Gamma \mid \Delta, A_0, A_0; A_1 \vdash (\mathsf{unlock}\ \overline{v'})\{v/x\} : ![] \dashv \Delta, A_1$ (7)

by (vs:16) on (6).

Thus, we conclude.

□

3. (Location Variable) - immediate by (Well-Formed Type Substitution) since our expressions do not contain types.

4. (Type Variable) - analogous to (Location Variable).

□

A.2.7 Values Lemma

Lemma 24 (Values Lemma)**.** If v is a closed value such that

$$\Gamma \mid \Delta \vdash v : A \dashv \Delta'$$

then

$$\Gamma \vdash \Delta <: \Delta_v, \Delta' \qquad \Gamma \mid \Delta_v \vdash v : A \dashv \cdot$$

Proof. By induction on the typing derivation of $\Gamma \mid \Delta \vdash v : A \dashv \Delta'$.

Case (T:REF) - We have:

$\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\, \rho \dashv \cdot$ (1)
by hypothesis.

Thus, by making:
$\Delta_v = \cdot$ (2)
$\Delta' = \cdot$ (3)
We immediately conclude.

Case (T:PURE) - We have:

$\Gamma \mid \cdot \vdash v : !A \dashv \cdot$ (1)
by hypothesis.

Thus, by making:
$\Delta_v = \cdot$ (2)
$\Delta' = \cdot$ (3)
We immediately conclude.

Case (T:UNIT) - We have:

$\Gamma \mid \cdot \vdash v : ![] \dashv \cdot$ (1)
by hypothesis.

Thus, by making:
$\Delta_v = \cdot$ (2)
$\Delta' = \cdot$ (3)
We immediately conclude.

Case (T:PURE-READ), (T:LINEAR-READ) - value not closed.

Case (T:PURE-ELIM) - Environment not closed.

Case (T:NEW),(T:DELETE), (T:ASSIGN), (T:DEREFERENCE-LINEAR), (T:DEREFERENCE-PURE) - Not a value.

Case (T:RECORD) - We have:

$\Gamma \mid \Delta_0 \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \Delta_1$ (1)

by hypothesis.

$\overline{\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1}$ (2)

by inversion on (T:RECORD) with (1).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1$ (3)

$\overline{\Gamma \mid \Delta_v \vdash v : A \dashv \cdot}$ (4)

by induction hypothesis on (2).

$\Gamma \mid \Delta_v \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (5)

by (T:RECORD) on (4).

Therefore, by (3) and (5) we conclude.

Case (T:SELECTION), (T:APPLICATION) - Not a value.

Case (T:FUNCTION) - We have:

$\Gamma \mid \Delta \vdash \lambda x.e : A_0 \multimap A_1 \dashv \cdot$ (1)

by hypothesis.

Thus, by making:
$\Delta' = \cdot$ (2)
We immediately conclude.

Case (T:CAP-ELIM) - Environment not closed.

Case (T:CAP-STACK) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 :: A_1 \dashv \Delta_1$ (1)

by hypothesis.

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1, A_1$ (2)

by inversion on (T:CAP-STACK) with (1).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1, A_1$ (3)
$\Gamma \mid \Delta_v \vdash v : A_0 \dashv \cdot$ (4)

by induction hypothesis on (2).

$\Gamma \mid \Delta_v, A_1 \vdash v : A_0 \dashv A_1$ (5)

by (T:FRAME) on (4) using A_1.

$\Gamma \mid \Delta_v, A_1 \vdash v : A_0 :: A_1 \dashv \cdot$ (6)

by (T:CAP-STACK) on (5).

Therefore, by (3) and (6) we conclude.

Case (T:CAP-UNSTACK) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1, A_1$ (1)

by hypothesis.

$\Gamma \mid \Delta_0 \vdash v : A_0 :: A_1 \dashv \Delta_1$ (2)

by inversion on (T:CAP-UNSTACK) with (1).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1$ (3)
$\Gamma \mid \Delta_v \vdash v : A_0 :: A_1 \dashv \cdot$ (4)

by induction hypothesis on (2).

$\Gamma \mid \Delta_v \vdash v : A_0 \dashv A_1$ (5)

by (T:Cap-Unstack) with (4).

$\Gamma \vdash \Delta_v <: \Delta'_v, A_1$ (6)

$\Gamma \mid \Delta'_v \vdash v : A_0 \dashv \cdot$ (7)

by induction hypothesis on (5).

$\Gamma \vdash \Delta_0 <: \Delta'_v, A_1, \Delta_1$ (8)

by transitivity of subtyping with (3) and (6).

Therefore, by (7) and (8) we conclude.

Case (T:Frame) - We have:

$\Gamma \mid \Delta_0, \Delta_2 \vdash v : A \dashv \Delta_1, \Delta_2$ (1)

by hypothesis.

$\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1$ (2)

by inversion on (T:Frame) with (1).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1$ (3)

$\Gamma \mid \Delta_v \vdash v : A \dashv \cdot$ (4)

by induction hypothesis on (2).

$\Gamma \vdash \Delta_0, \Delta_2 <: \Delta_v, \Delta_1, \Delta_2$ (5)

by (1) and (3) we know we can add Δ_2.

Therefore, by (4) and (5) we immediately conclude.

Case (T:Subsumption) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_1 \dashv \Delta_1$ (1)

by hypothesis.

$\Gamma \vdash \Delta_0 <: \Delta'_0$ (2)

$\Gamma \mid \Delta'_0 \vdash v : A_0 \dashv \Delta'_1$ (3)

$\Gamma \vdash A_0 <: A_1$ (4)

$\Gamma \vdash \Delta'_1 <: \Delta_1$ (5)

by inversion on (T:Subsumption) with (1).

$\Gamma \vdash \Delta'_0 <: \Delta_v, \Delta'_1$ (6)

$\Gamma \mid \Delta_v \vdash v : A_0 \dashv \cdot$ (7)

by induction hypothesis on (3).

$\Gamma \vdash \Delta'_0 <: \Delta_v, \Delta_1$ (8)

by transitivity of subtyping with (5) and (6).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1$ (9)

by transitivity of subtyping with (2) and (8).

$\Gamma \mid \Delta_v \vdash v : A_1 \dashv \cdot$ (10)

by (T:Subsumption) with (SD:Symmetry) and (4) on (7).

Therefore, by (9) and (10) we conclude.

Case (T:Tag) - We have:

$\Gamma \mid \Delta_0 \vdash \mathtt{t\#}v : \mathtt{t\#}A \dashv \cdot$ (1)

by hypothesis.

$\Gamma \mid \Delta_0 \vdash v : A \dashv \cdot$ (2)

by inversion on (T:TAG) with (1).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1$ (3)

$\Gamma \mid \Delta_v \vdash v : A \dashv \cdot$ (4)

by induction hypothesis on (2).

$\Gamma \mid \Delta_v \vdash \mathtt{t\#}v : \mathtt{t\#}A \dashv \cdot$ (5)

by (T:TAG) on (4).

Therefore, by (5) and (3) we conclude.

Case (T:CASE) - Not a value.

Case (T:ALTERNATIVE-LEFT) - We have:

$\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash v : A_2 \dashv \Delta_1$ (1)

by hypothesis.

$\Gamma \mid \Delta_0, A_0 \vdash v : A_2 \dashv \Delta_1$ (2)

$\Gamma \mid \Delta_0, A_1 \vdash v : A_2 \dashv \Delta_1$ (3)

by inversion on (T:ALTERNATIVE-LEFT) with (1).

$\Gamma \vdash \Delta_0, A_0 <: \Delta_v, \Delta_1$ (4)

$\Gamma \mid \Delta_v \vdash v : A_2 \dashv \cdot$ (5)

by induction hypothesis on (2).

$\Gamma \vdash \Delta_0, A_1 <: \Delta_v, \Delta_1$ (6)

$\Gamma \mid \Delta_v \vdash v : A_2 \dashv \cdot$ (7)

by induction hypothesis on (3).

$\Gamma \vdash \Delta_0, A_0 \oplus A_1 <: \Delta_v, \Delta_1$ (8)

by (SD:ALTERNATIVE-L) on (4) and (6).

Therefore, by (8) and (7) we conclude.

Case (T:INTERSECTION-RIGHT) - We have:

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1, A_1 \& A_2$ (1)

by hypothesis.

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1, A_1$ (2)

$\Gamma \mid \Delta_0 \vdash v : A_0 \dashv \Delta_1, A_2$ (3)

by inversion on (T:INTERSECTION-RIGHT) with (1).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1, A_1$ (4)

$\Gamma \mid \Delta_v \vdash v : A_0 \dashv \cdot$ (5)

by induction hypothesis on (2).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1, A_2$ (6)

$\Gamma \mid \Delta_v \vdash v : A_0 \dashv \cdot$ (7)

by induction hypothesis on (3).

$\Gamma \vdash \Delta_0 <: \Delta_v, \Delta_1, A_1 \& A_2$ (8)

by (SD:INTERSECTION-R) on (4) and (6).

Thus, by (8) and (7) we conclude.

Case (T:LET), (T:FORK), (T:LOCK-RELY), (T:UNLOCK-GUARANTEE) - Not values.

Case (T:FORALL-LOC-VAL), (T:FORALL-TYPE-VAL) - Immediate since both rules have no resulting effects.

Case (T:TYPEOPENBIND), (T:TYPEOPENCAP), (T:LOCOPENCAP), (T:LOCOPENBIND) - Environment not closed.

□

A.2.8 Protocol Lemmas

Lemma 25 (Composition Progress)**.** If $\Gamma \vdash S \Rrightarrow P \parallel Q$ then $\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto C$.

In other words, if two protocols conform, then the configuration can always take a step to some other set of configurations.

Proof. By induction on the derivation of $\Gamma \vdash S \Rrightarrow P \parallel Q$:

$\Gamma \vdash S \Rrightarrow P \parallel Q$ (1)
by hypothesis.
$\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle\!\uparrow$ (2)
by inversion on (1) with (WF:SPLIT).
$\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto C$ (3)
$C\!\uparrow$ (4)
by inversion on (2) with (WF:CONFIGURATION).

Thus, we conclude by (3).

□

Lemma 26 (Composition Preservation)**.** If $\Gamma \vdash S \Rrightarrow P \parallel Q$ and
$\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash S' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C$ then $\Gamma' \vdash S' \Rrightarrow P' \parallel Q'$.

If two protocols conform then the respective configuration can take a step by (Composition Progress). We now show that the resulting configuration still conforms. Thus, all interference produced by a protocol is expected by all other protocols of that state, regardless of how they are interleaved or how many times the protocol is split. In other words, the protocols never get stuck as their rely assumptions are valid.

Proof. Immediate by (WF:CONFIGURATION) and then (WF:SPLIT) on each new configuration. That is, since we know the configurations conformed to begin with, then by the definition of (WF:CONFIGURATION) they will conform to all possible future reachable configurations.

By induction on the derivation of $\Gamma \vdash S \Rrightarrow P \parallel Q$:

$\Gamma \vdash S \Rrightarrow P \parallel Q$ (1)
$\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash S' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C$ (2)
by hypothesis.
$\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle\!\uparrow$ (3)
by inversion on (1) with (WF:SPLIT).
$\langle\, \Gamma \vdash S \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash S' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C$ (4)
$\langle\, \Gamma' \vdash S' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C\!\uparrow$ (5)
by inversion on (2) with (WF:CONFIGURATION).
$\langle\, \Gamma' \vdash S' \Rrightarrow P' \parallel Q' \,\rangle\!\uparrow$ (6)
by (WF:CONFIGURATION) and (C:ALLSTEP) on (5).
$\Gamma' \vdash S' \Rrightarrow P' \parallel Q'$ (7)
by (WF:SPLIT) on (6).

Thus, we conclude by (7).

□

Lemma 27 (Protocol Properties)**.** Protocol composition obeys the follow properties:

$$\Gamma \vdash S \Rrightarrow S \parallel \textbf{none} \qquad \text{(Identity)}$$
$$\text{If } \Gamma \vdash S \Rrightarrow P_0 \parallel P_1 \text{ then } \Gamma \vdash S \Rrightarrow P_1 \parallel P_0 \qquad \text{(Commutativity)}$$
$$\text{If } \Gamma \vdash S \Rrightarrow P_0 \parallel (P_1 \parallel P_2) \text{ then } \Gamma \vdash S \Rrightarrow (P_0 \parallel P_1) \parallel P_2 \qquad \text{(Associativity)}$$

Showing identity and that protocol composition is commutative are immediate. We now show that protocol composition is associative.

Lemma 28 (Associativity)**.** If we have:

$$\Gamma \vdash S \Rrightarrow P \parallel Q \qquad \Gamma \vdash P \Rrightarrow P_0 \parallel P_1$$

then, exists W such that:

$$\Gamma \vdash S \Rrightarrow P_0 \parallel W \qquad \Gamma \vdash W \Rrightarrow P_1 \parallel Q$$

(i.e. if $\Gamma \vdash S \Rrightarrow \underbrace{(P_0 \parallel P_1)}_{P} \parallel Q$ then $\Gamma \vdash S \Rrightarrow P_0 \parallel \underbrace{(P_1 \parallel Q)}_{W}$).

Proof. We proceed by induction on the structure of S. Note that we omit cases related to the use of (c-rs:Subsumption) and (c-rs:Weakening) since they are straightforward and detract from the core aspects of the proof.

1. Case $S = \textbf{none}$. Immediate since neither protocol can take a step.
2. Case $S = S_0 \oplus S_1$, we have:

   $\Gamma \vdash S_0 \oplus S_1 \Rrightarrow P \parallel Q$ (1)
   $\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)
   by hypothesis.
   $\Gamma \vdash S_0 \Rrightarrow P \parallel Q$ (3)
   $\Gamma \vdash S_1 \Rrightarrow P \parallel Q$ (4)
   by inversion with (c-rs:StateAlternative) on (1).

   Then, by induction hypothesis we have that exists W such that (remember that we can weaken W to match both application of the induction hypothesis, for instance by applying subtyping rules):

   $\Gamma \vdash S_0 \Rrightarrow P_0 \parallel W$ (5)
   $\Gamma \vdash W \Rrightarrow P_1 \parallel Q$ (6)
   $\Gamma \vdash S_1 \Rrightarrow P_0 \parallel W$ (7)
   by induction hypothesis on (2) and (3), and (2) and (4).
   $\Gamma \vdash S_0 \oplus S_1 \Rrightarrow P_0 \parallel W$ (8)
   by (c-rs:StateAlternative) with (5) and (7).

   Thus, we conclude.

3. Case $S = S_0 \& S_1$, we have:

$\Gamma \vdash S_0 \& S_1 \Rrightarrow P \parallel Q$ (1)
$\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)

by hypothesis.

$\Gamma \vdash S_0 \Rrightarrow P \parallel Q$ (3)

by inversion with (c-rs:StateIntersection) on (1).

Then, by induction hypothesis we have that exists W such that:

$\Gamma \vdash S_0 \Rrightarrow P_0 \parallel W$ (4)
$\Gamma \vdash W \Rrightarrow P_1 \parallel Q$ (5)

by induction hypothesis with (2) and (3).

$\Gamma \vdash S_0 \& S_1 \Rrightarrow P_0 \parallel W$ (6)

by (c-rs:StateIntersection) on (4).

Thus, we conclude.

4. Case $S = A$ (a non-protocol type), we have:

   $\Gamma \vdash A \Rrightarrow P \parallel Q$ (1)
   $\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)

   by hypothesis.

   We now do a case analysis on the structure of P. We omit cases where $P = \textbf{none}$, $P = P' \oplus P''$, and $P = P' \& P'$ since they are straightforward and instead focus on actual protocol steps of P.

   (a) Case $P = A \Rightarrow A'; P'$, we can re-write our hypothesis:
   $\Gamma \vdash A \Rrightarrow (A \Rightarrow A'; P') \parallel Q$ (1)
   $\Gamma \vdash (A \Rightarrow A'; P') \Rrightarrow P_0 \parallel P_1$ (2)

   by hypothesis.

   Then, by inversion on (2) we have that:
   $P_0 = A \Rightarrow A'; P'_0$ (3)
   $P_1 = A \Rightarrow A'; P'_1$ (4)

   by inversion on (2) with (c-ps:Step).

   Thus, we can build W such that it intertwines both P_1 and Q with a &. Assume that W' is such a protocol but built for the next step of the protocol and where Q' is the intertwine version Q so that it obeys the intended protocol condition. Then we can have:
   $W = (A \Rightarrow A'; W') \& Q'$ (5)
   Thus, we conclude (noting that by (3) we see that P_0 also steps with A).

   (b) Cases $P = A \Rightarrow \forall l.P'$ and $P = A \Rightarrow \forall X <: S'.P'$ are straightforward by inversion and application of the induction hypothesis. Similarly, the sub-case where P_0 (or P_1) transition by (c-ps:TypeApp) or by (c-ps:LocApp) do not affect the step since the rely type is unchanged. Furthermore, we can build W with the applied type/location already there.

(c) Cases $P = \exists X <: S'.P'$ and $P = \exists l.P'$ are straightforward by inversion on (c-ss:Open*) and (c-ps:Exists*), induction hypothesis, and re-applying the rule.

(d) Case $P = A$ (ownership recovery), we have:

$\Gamma \vdash A \Rrightarrow A \parallel Q$ (1)
$\Gamma \vdash A \Rrightarrow P_0 \parallel P_1$ (2)
by hypothesis.

Then, by inversion on (2) we have that either:

- P_0 or P_1 are also A. Then, since the protocols conform the other protocol cannot touch the shared state since the shared state becomes **none**. Therefore, W is simply the combination of Q and the other protocol that recovers ownership.

- P_0 or P_1 extend the uses of A, i.e. they are appending some protocol to what was previously ownership recovery. Thus, we have:

  $P_0 = A \Rightarrow A'; P'_0$ (3)
  $P_1 = A \Rightarrow A'; P'_1$ (4)
  by inversion on (2) with (c-ps:Step).

  But since Q conforms with A, Q cannot use the shared state and must be equivalent of **none**. Therefore, we can build $W = P_1$ and immediately conclude.

5. Case $S = A \Rightarrow A'; S'$ (a protocol type), we have:

$\Gamma \vdash (A \Rightarrow A'; S') \Rrightarrow P \parallel Q$ (1)
$\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)
by hypothesis.

$P = A \Rightarrow A'; P'$ (3)
$P_0 = A \Rightarrow A'; P'_0$ (4)
$P_1 = A \Rightarrow A'; P'_1$ (5)
by inversion on (2) with (c-ps:Step).

As before, we can build W by intertwining Q and P_1 with the W' protocol resulting from applying the induction hypothesis to the next step. So that, as done above:

$W = (A \Rightarrow A'; W')\&Q'$ (6)
Thus, we conclude.

6. Cases $S = \exists l.S'$ (case $S = \exists X <: S''.S'$ is analogous), we have:

$\Gamma \vdash \exists l.S' \Rrightarrow P \parallel Q$ (1)
$\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)
by hypothesis.

$P = \exists l.P'$ (3)
$P_0 = \exists l.P'_0$ (4)
$P_1 = \exists l.P'_1$ (5)

$Q = \exists l.Q'$ (6)

since the protocols conform with S and P.

$\Gamma, l : \mathbf{loc} \vdash S' \Rrightarrow P' \parallel Q'$ (7)

$\Gamma, l : \mathbf{loc} \vdash P' \Rrightarrow P'_0 \parallel P'_1$ (8)

by inversion on (2) with (c-ps:ExistsLoc).

Then, by induction hypothesis there exists a W such that:

$\Gamma, l : \mathbf{loc} \vdash S' \Rrightarrow P'_0 \parallel W$ (9)

$\Gamma, l : \mathbf{loc} \vdash W \Rrightarrow P'_1 \parallel Q'$ (10)

by induction hypothesis on (7) and (8).

$\Gamma \vdash \exists l.S' \Rrightarrow \exists l.P'_0 \parallel \exists l.W$ (11)

$\Gamma \vdash \exists l.W \Rrightarrow \exists l.P'_1 \parallel \exists l.Q'$ (12)

by (c-ps:ExistsLoc) on (9) and (10).

Thus, we conclude.

7. Case $S = A \Rightarrow \forall l.S''$ (case $S = A \Rightarrow \forall X <: S'.S''$ is analogous), we have:

   $\Gamma \vdash A \Rightarrow \forall l.S' \Rrightarrow P \parallel Q$ (1)

   $\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)

   by hypothesis.

   We have that P will either "re-use" the $\forall$ or do a type application. Either case is straightforward by inversion and induction hypothesis.

□

A.2.9 Preservation

Note that preservation is only defined over closed programs and expressions so that they can step.
Theorem 16 (Program Preservation)**.** If we have

$$\Gamma_0 \mid \Delta_0 \vdash H_0 \qquad \Gamma_0 \mid \Delta_0 \vdash T_0 \qquad H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$$

then, for some Δ_1 and Γ_1, we have

$$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash H_1 \qquad \Gamma_0, \Gamma_1 \mid \Delta_1 \vdash T_1$$

Proof. We proceed by induction on the typing derivation of $\Gamma_0 \mid \Delta_0 \vdash T_0$.

Case (WF:PROGRAM) - we have:

$\Gamma_0 \mid \Delta_0 \vdash H_0$ (1)
$\Gamma_0 \mid \Delta_0 \vdash T_0$ (2)
$H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$ (3)
by hypothesis.
$\Gamma_0 \mid \Delta_{0_i} \vdash e_i : ![] \dashv \cdot$ (4)
$i \in \{0, ..., n\}$ (5)
$n \geq 0$ (6)
$T_0 = e_0 \cdot ... \cdot e_n$ (7)
$\Delta_0 = \Delta_{0_0}, ..., \Delta_{0_n}$ (8)
by inversion on (WF:PROGRAM) with (2) noting that the order of each threads in T_0 is not important.
$H_0 \; ; \; e_j \; \mapsto \; H_1 \; ; \; e'_j \cdot T'$ (9)
by inversion on (D:THREAD) with (3) on some thread j such that $j \in \{0, ..., n\}$.
For clarity we now rewrite some of the above to reflect e_j:
$\Gamma_0 \mid \underbrace{\Delta_j, \Delta_{j_T}, \Delta'}_{\Delta_0} \vdash H_0$ (10)
by rewriting (1).
$\Gamma_0 \mid \underbrace{\Delta_j, \Delta_{j_T}}_{\Delta_{0_j}} \vdash e_j : ![] \dashv \cdot$ (11)
by rewriting (4).

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e'_j : ![] \dashv \cdot$ (12)
$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_{j_T}, \Delta' \vdash H_1$ (13)
$\Gamma_0, \Gamma_1 \mid \Delta_{j_T} \vdash T'$ (14)
by (Expression Preservation) with (10), (11), and (9).

Thus, we conclude by (WF:PROGRAM) with (12), (13), (14) and (4) to accommodate the remaining threads in T_0 combined with weakening the lexical environment to include Γ_1. □

Theorem 17 (Expression Preservation)**.** If we have

$$\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0 \qquad \Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A \dashv \Delta \qquad H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$$

then, for some Δ_1 and Γ_1, we have

$$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1 \qquad \Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta \qquad \Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$$

Proof. We proceed by induction on the typing derivation of $\Gamma_0 \mid \Delta_0 \vdash e_0 : A \dashv \Delta$.

Case (T:Ref), (T:Pure), (T:Unit) - are values (no step available).

Case (T:Pure-Read), (T:Linear-Read), (T:Pure-Elim) - not applicable, environments not closed.

Case (T:New) - We have:

$\Gamma_0 \mid \Delta_0 \vdash \mathsf{new}\ v : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta$ (1)

$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0$ (2)

$H_0\ ;\ \mathcal{E}[\mathsf{new}\ v] \ \mapsto\ H_0\ ,\ \rho \hookrightarrow v\ ;\ \mathcal{E}[\rho]$ (3)

by hypothesis, with (D:New) where $\mathcal{E} = \square$ note that we have $\Delta_T = \cdot$ since there is no resulting thread from this step.

$\Gamma_0 \mid \Delta_0 \vdash v : A \dashv \Delta$ (4)

by inversion on (T:New) with (1).

$\Gamma_0 \vdash \Delta_0 <: \Delta_v, \Delta$ (5)

$\Gamma_0 \mid \Delta_v \vdash v : A \dashv \cdot$ (6)

by (Values Lemma) with (4).

ρ **fresh** (7)

by inversion on (D:New) with (3).

$\Gamma_0 \mid \Delta_v, \Delta, \Delta_2 \vdash H_0$ (8)

by (STR:Subsumption) with (2) and (5).

Thus, if we make:

$\Gamma_1 = \rho : \mathbf{loc}$ (9)

We have that:

$\Gamma_0, \Gamma_1 \mid \Delta_v \vdash v : A \dashv \cdot$ (10)

by (Weakening) (6) with Γ_1 (note that weakening is only valid in the lexical environments, Γ).

$\Gamma_0, \Gamma_1 \mid \Delta_v, \Delta, \Delta_2 \vdash H_0$ (11)

by (STR:Loc) with Γ_1 (that contains ρ) on (8).

$\Gamma_0, \Gamma_1 \mid \Delta, \Delta_2, \mathbf{rw}\ \rho\ A \vdash H_0\ ,\ \rho \hookrightarrow v$ (12)

by (STR:Binding) with (10) and (11) with ρ.

Thus, if we make:

$\Delta_1 = \Delta, \mathbf{rw}\ \rho\ A$ (13)

We have that:

$\Gamma_0, \Gamma_1 \mid \cdot \vdash \rho : !\mathbf{ref}\ \rho \dashv \cdot$ (14)

by (T:Ref) with ρ and (T:Pure).

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash \rho : !\mathbf{ref}\ \rho \dashv \Delta_1$ (15)

by (T:Frame) on (14) with Δ_1.

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash \rho : !\mathbf{ref}\ \rho :: \mathbf{rw}\ \rho\ A \dashv \Delta$ (16)

by (T:Cap-Stack) on (15) noting that (13).

If l **fresh** then:
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash \rho : (\ !\mathbf{ref}\ \rho :: \mathbf{rw}\ \rho\ A)\{\rho/l\} \dashv \Delta$ (17)
by type substitution on (14).
Note that, by (4), ρ cannot occur in A since it is a fresh location constant not present in Γ_0.
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash \rho : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta$ (18)
by (T:Subsumption) together with (ST:PackLoc) on (17).
Thus:
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash \rho : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta$ (19)
for some Δ_1, Γ_1.
by (18).
Therefore, by (12) and (19) we conclude noting that T is empty meaning that we also have $\Delta_T = \cdot$.

Case (T:Delete) - We have:

1. Case where ρ is not shared (and thus not locked):
$\Gamma_0 \mid \Delta_0 \vdash \mathsf{delete}\ \rho : \exists l.A \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0\ ,\ \rho \hookrightarrow v$ (2)
$H_0\ ,\ \rho^? \hookrightarrow v\ ;\ \mathcal{E}[\mathsf{delete}\ \rho]\ \mapsto\ H_0\ ;\ \mathcal{E}[v]$ (3)
by hypothesis, with (D:Delete) where $\mathcal{E} = \square$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.
$\Gamma_0 \mid \Delta_0 \vdash \rho : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta$ (4)
by inversion on (T:Delete) with (1).
$\Gamma_0 \vdash \Delta_0 <: \Delta_\rho, \Delta$ (5)
$\Gamma_0 \mid \Delta_\rho \vdash \rho : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \cdot$ (6)
by (Values Lemma) with (4).
$\Gamma_0 \mid \Delta_\rho \vdash \rho : (!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A)\{\rho/l\} \dashv \cdot$ (7)
by (Values Inversion) with (6).
$\Gamma_0 \mid \Delta_\rho \vdash \rho : !\mathbf{ref}\ \rho :: \mathbf{rw}\ \rho\ A\{\rho/l\} \dashv \cdot$ (8)
by (LS:2.6), (LS:2.10), (LS:2.1), (LS:2.12) with (7).
$\Gamma_0 \mid \Delta_\rho \vdash \rho : !\mathbf{ref}\ \rho \dashv \mathbf{rw}\ \rho\ A\{\rho/l\}$ (9)
by (Values Inversion) with (8).
$\Gamma_0 \vdash \Delta_\rho <: \Delta'_\rho, \mathbf{rw}\ \rho\ A\{\rho/l\}$ (10)
$\Gamma_0 \mid \Delta'_\rho \vdash \rho : !\mathbf{ref}\ \rho \dashv \cdot$ (11)
by (Values Lemma) with (9).
$\Delta'_\rho = \cdot$ (12)
by inversion on (T:Ref) with (11).
Therefore:
$\Gamma_0 \mid \Delta'_\rho, \mathbf{rw}\ \rho\ A\{\rho/l\}, \Delta, \Delta_2 \vdash H_0\ ,\ \rho \hookrightarrow v$ i.e.:
$\Gamma_0 \mid \mathbf{rw}\ \rho\ A\{\rho/l\}, \Delta, \Delta_2 \vdash H_0\ ,\ \rho \hookrightarrow v$ (13)
by (STR:Subsumption) using (2), (10) and (12).
$\Gamma_0 \mid \Delta_v, \Delta, \Delta_2 \vdash H_0$ (14)
$\Gamma_0 \mid \Delta_v \vdash v : A\{\rho/l\} \dashv \cdot$ (15)
by (Store Typing Inversion) with (13).

$\Gamma_0 \mid \Delta_v \vdash v : \exists l.A \dashv \cdot$ (16)

by (T:Subsumption) together with (ST:PackLoc) on (15).

$\Gamma_0 \mid \Delta_v, \Delta \vdash v : \exists l.A \dashv \Delta$ (17)

by (T:Frame) with (16) using Δ.

Using:

$\Gamma_1 = \cdot$ (18)

$\Delta_1 = \Delta_v, \Delta$ (19)

We have:

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash v : \exists l.A \vdash \Delta$ (20)

by (17) with (18) and (19).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_2 \vdash H_0$ (21)

by rewriting (14) with (18) and (19).

Therefore, by (20) and (21) we conclude, noting that T is empty meaning that we also have $\Delta_T = \cdot$.

2. Case when ρ is shared (and thus the location is locked and there is a pending guarantee) is analogous to the previous case except for the use of (STR:Dead-Locked) at the end to store typing the resulting environments and heap.

Case (T:Assign) - We have:

$\Gamma_0 \mid \Delta_0 \vdash \rho := v_1 : A_1 \dashv \Delta, \mathbf{rw}\ \rho\ A_0$ (1)

$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v_0$ (2)

$H_0\ ,\ \rho^? \hookrightarrow v_0\ ;\ \mathcal{E}[\rho := v_1]\ \mapsto\ H_0\ ,\ \rho^? \hookrightarrow v_1\ ;\ \mathcal{E}[v_0]$ (3)

by hypothesis, with (D:Assign) where $\mathcal{E} = \square$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.

$\Gamma_0 \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta'$ (4)

$\Gamma_0 \mid \Delta' \vdash \rho : \mathbf{ref}\ \rho \dashv \Delta, \mathbf{rw}\ \rho\ A_1$ (5)

by inversion on (T:Assign) with (1).

$\Gamma_0 \vdash \Delta_0 <: \Delta_{v_1}, \Delta'$ (6)

$\Gamma_0 \mid \Delta_{v_1} \vdash v_1 : A_0 \dashv \cdot$ (7)

by (Values Lemma) on (4).

$\Gamma_0 \vdash \Delta' <: \Delta_\rho, \Delta, \mathbf{rw}\ \rho\ A_1$ (8)

$\Gamma_0 \mid \Delta_\rho \vdash \rho : \mathbf{ref}\ \rho \dashv \cdot$ (9)

by (Values Lemma) on (5).

$\Delta_\rho = \cdot$ (10)

by inversion on (T:Ref) with (9).

$\Gamma_0 \mid \Delta_{v_1}, \Delta, \mathbf{rw}\ \rho\ A_1, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v_0$ (11)

by (STR:Subsumption) with (2), (6), (8) and (10).

$\Gamma_0 \mid \Delta_{v_1}, \Delta_{v_0}, \Delta, \Delta_2 \vdash H_0$ (12)

$\Gamma_0 \mid \Delta_{v_0} \vdash v_0 : A_1 \dashv \cdot$ (13)

by (Store Typing Inversion) on (11).

$\Gamma_0 \mid \Delta_{v_0}, \Delta, \mathbf{rw}\ \rho\ A_0, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v_1$ (14)

by (STR:Binding) with ρ on (7) and (12).

by making:

$\Gamma_1 = \cdot$ (15)

$\Gamma_0, \Gamma_1 \mid \Delta_{v_0}, \Delta, \mathbf{rw}\ \rho\ A_0, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v_1$ (16)

by (Weakening) with (14).

$\Gamma_0, \Gamma_1 \mid \Delta_{v_0} \vdash v_0 : A_1 \dashv \cdot$ (17)

by (Weakening) on (13).

$\Gamma_0, \Gamma_1 \mid \Delta_{v_0}, \Delta, \mathbf{rw}\ \rho\ A_0 \vdash v_0 : A_1 \dashv \Delta, \mathbf{rw}\ \rho\ A_0$ (18)

by (T:Frame) using $\Delta, \mathbf{rw}\ \rho\ A_0$ with (17).

Therefore, by (16) and (18) we conclude (noting that T is empty meaning that we also have $\Delta_T = \cdot$).

Case (T:Dereference-Linear) - We have:

$\Gamma_0 \mid \Delta_0 \vdash !\rho : A \dashv \Delta, \mathbf{rw}\ \rho\ ![]$ (1)

$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v$ (2)

$H_0\ ,\ \rho^? \hookrightarrow v\ ;\ \mathcal{E}[!\rho]\ \mapsto\ H_0\ ,\ \rho^? \hookrightarrow v\ ;\ \mathcal{E}[v]$ (3)

by hypothesis, (D:Dereference) where $\mathcal{E} = \square$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.

$\Gamma_0 \mid \Delta_0 \vdash \rho : \mathbf{ref}\ \rho \dashv \Delta, \mathbf{rw}\ \rho\ ![]$ (4)

by inversion on (T:Dereference-Linear) with (1).

$\Gamma_0 \vdash \Delta_0 <: \Delta_\rho, \Delta, \mathbf{rw}\ \rho\ A$ (5)

$\Gamma_0 \mid \Delta_\rho \vdash \rho : \mathbf{ref}\ \rho \dashv \cdot$ (6)

by (Values Lemma) on (4).

$\Delta_\rho = \cdot$ (7)

by (Values Inversion) on (6).

$\Gamma_0 \vdash \Delta_0 <: \Delta, \mathbf{rw}\ \rho\ A$ (8)

by rewriting (5) with (7).

$\Gamma_0 \mid \Delta, \mathbf{rw}\ \rho\ A, \Delta_2 \vdash H_0, \rho^? \hookrightarrow v$ (9)

by (str:Subsumption) with (8) and (2).

$\Gamma_0 \mid \Delta_v \vdash v : A \dashv \cdot$ (10)

$\Gamma_0 \mid \Delta, \Delta_v, \Delta_2 \vdash H_0$ (11)

by (Store Typing Inversion) with (9).

$\Gamma_0 \mid \cdot \vdash v : ![] \dashv \cdot$ (12)

by (T:Unit) with value v.

$\Gamma_0 \mid \Delta, \Delta_v, \mathbf{rw}\ \rho\ ![], \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v$ (13)

by (str:Binding) using ρ, (11) and (12).

by making:

$\Gamma_1 = \cdot$ (14)

$\Gamma_0, \Gamma_1 \mid \Delta, \Delta_v, \mathbf{rw}\ \rho\ ![], \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v$ (15)

by (Weakening) using Γ_1 on (13).

$\Gamma_0, \Gamma_1 \mid \Delta_v \vdash v : A \dashv \cdot$ (16)

by (Weakening) using Γ_1 on (10).

$\Gamma_0, \Gamma_1 \mid \Delta_v, \Delta, \mathbf{rw}\ \rho\ ![] \vdash v : A \dashv \Delta, \mathbf{rw}\ \rho\ ![]$ (17)

by (T:Frame) using $\Delta, \mathbf{rw}\ \rho\ ![]$ on (16).

Therefore, by (15) and (17) we conclude (noting that T is empty meaning that we also have $\Delta_T = \cdot$).

Case (T:Dereference-Pure) - We have:

$\Gamma_0 \mid \Delta_0 \vdash !\rho : !A \dashv \Delta, \mathbf{rw}\ \rho\ \ !A$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v$ (2)
$H_0\ ,\ \rho^? \hookrightarrow v\ ;\ !\rho\ \mapsto\ H_0\ ,\ \rho^? \hookrightarrow v\ ;\ \mathcal{E}[v]$ (3)

by hypothesis, with (D:Dereference) where $\mathcal{E} = \square$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.

$\Gamma_0 \mid \Delta_0 \vdash \rho : \mathbf{ref}\ \rho \dashv \Delta, \mathbf{rw}\ \rho\ \ !A$ (4)
by inversion on (T:Dereference-Pure) with (1).
$\Gamma_0 \vdash \Delta_0 <: \Delta_\rho, \Delta, \mathbf{rw}\ \rho\ \ !A$ (5)
$\Gamma_0 \mid \Delta_\rho \vdash \rho : \mathbf{ref}\ \rho \dashv \cdot$ (6)
by (Values Lemma) on (4).
$\Delta_\rho = \cdot$ (7)
by (Values Inversion) on (6).
$\Gamma_0 \vdash \Delta_0 <: \Delta, \mathbf{rw}\ \rho\ \ !A$ (8)
by rewriting (5) with (7).
$\Gamma_0 \mid \Delta, \mathbf{rw}\ \rho\ \ !A, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v$ (9)
by (str:Subsumption) with (8) and (2).
$\Gamma_0 \mid \Delta_v \vdash v : !A \dashv \cdot$ (10)
$\Gamma_0 \mid \Delta, \Delta_v, \Delta_2 \vdash H_0$ (11)
by (Store Typing Inversion) with (9).
$\Delta_v = \cdot$ (12)
$\Gamma_0 \mid \cdot \vdash v : !A \dashv \cdot$ (13)
by (Values Inversion) on (10).
$\Gamma_0 \mid \Delta, \Delta_2 \vdash H_0$ (14)
by rewriting (11) with (12).

by making:

$\Gamma_1 = \cdot$ (15)
$\Gamma_0, \Gamma_1 \mid \Delta, \mathbf{rw}\ \rho\ \ !A, \Delta_2 \vdash H_0\ ,\ \rho^? \hookrightarrow v$ (16)
by (Weakening) using Γ_1 on (9).
$\Gamma_0, \Gamma_1 \mid \cdot \vdash v : !A \dashv \cdot$ (17)
by (Weakening) using Γ_1 on (13).
$\Gamma_0, \Gamma_1 \mid \Delta, \mathbf{rw}\ \rho\ \ !A \vdash v : !A \dashv \Delta, \mathbf{rw}\ \rho\ \ !A$ (18)
by (T:Frame) using $\Delta, \mathbf{rw}\ \rho\ \ !A$ on (17).

Therefore, by (16) and (18) we conclude (noting that T is empty meaning that we also have $\Delta_T = \cdot$).

Case (T:Record) - is a value.

Case (T:Selection) - We have:

$\Gamma_0 \mid \Delta_0 \vdash \{\overline{\mathtt{f} = v}\}.\mathtt{f}_i : A_i \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0$ (2)
$H_0 \,;\, \mathcal{E}[\{\overline{\mathtt{f} = v}\}.\mathtt{f}_i] \;\mapsto\; H_0 \,;\, \mathcal{E}[v_i]$ (3)
by hypothesis, with (D:SELECTION) where $\mathcal{E} = \Box$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.
$\Gamma_0 \mid \Delta_0 \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \Delta$ (4)
by inversion on (T:SELECTION) with (1).
$\Gamma_0 \vdash \Delta_0 <: \Delta', \Delta$ (5)
$\Gamma_0 \mid \Delta' \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot$ (6)
by (Values Lemma) on (4).
$\Gamma_0 \mid \Delta' \vdash v_i : A_i \dashv \cdot$ (7)
by (Values Inversion) with (6) .
$\Gamma_0 \mid \Delta', \Delta \vdash v_i : A_i \dashv \Delta$ (8)
by (T:FRAME) with Δ with (7).
$\Gamma_0 \mid \Delta', \Delta, \Delta_2 \vdash H_0$ (9)
by (STR:SUBSUMPTION) with (2) and (5).
Therefore, by making:
$\Gamma_1 = \cdot$ (10)
$\Delta_1 = \Delta', \Delta$ (11)
$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_2 \vdash H_0$ (12)
by (Weakening) with (10) on (9) and rewriting (9) using (11).
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash v_i : A_i \dashv \Delta$ (13)
by (Weakening) with (10) on (8) and rewriting (8) using (11).
Therefore, by (12) and (13) we conclude (noting that T is empty meaning that we also have $\Delta_T = \cdot$).

Case (T:APPLICATION) - We have:

$\Gamma_0 \mid \Delta_0 \vdash (\lambda x.e)\; v : A_1 \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0$ (2)
$H_0 \,;\, \mathcal{E}[(\lambda x.e)\; v] \;\mapsto\; H_0 \,;\, \mathcal{E}[e\{v/x\}]$ (3)
by hypothesis, with (D:APPLICATION) where $\mathcal{E} = \Box$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.
$\Gamma_0 \mid \Delta_0 \vdash \lambda x.e : A_0 \multimap A_1 \dashv \Delta'$ (4)
$\Gamma_0 \mid \Delta' \vdash v : A_0 \dashv \Delta$ (5)
by inversion on (T:APPLICATION) with (1).
$\Gamma_0 \vdash \Delta_0 <: \Delta', \Delta_v$ (6)
$\Gamma_0 \mid \Delta_v \vdash \lambda x.e : A_0 \multimap A_1 \dashv \cdot$ (7)
by (Values Lemma) on (4).
$\Gamma_0 \vdash \Delta' <: \Delta, \Delta'_v$ (8)
$\Gamma_0 \mid \Delta'_v \vdash v : A_0 \dashv \cdot$ (9)
by (Values Lemma) on (5).
$\Gamma_0 \mid \Delta_v, x : A_0 \vdash e : A_1 \dashv \cdot$ (10)

$v = \lambda x.e$ (11)
by (Values Inversion) with (7).

$\Gamma_0 \mid \Delta'_v, \Delta_v, \Delta \vdash v : A_0 \dashv \Delta_v, \Delta$ (12)
by (T:Frame) on (9) with Δ_v, Δ.

$\Gamma_0 \mid \Delta_v, x : A_0, \Delta \vdash e : A_1 \dashv \Delta$ (13)
by (T:Frame) on (10) with Δ.

$\Gamma_0 \mid \Delta_v, \Delta'_v, \Delta \vdash e\{v/x\} : A_1 \dashv \Delta$ (14)
by (Substitution Lemma - Linear) with (12) and (13).

By making:
$\Gamma_1 = \cdot$
$\Delta_1 = \Delta_v, \Delta'_v, \Delta$
We immediately have:
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e\{v/x\} : A_1 \dashv \Delta$ (15)
with (14).

$\Gamma_0, \Gamma_1 \mid \Delta', \Delta_v, \Delta_2 \vdash H_0$ (16)
by (str:Subsumption) with (2) and (6).

$\Gamma_0, \Gamma_1 \mid \Delta, \Delta'_v, \Delta_v, \Delta_2 \vdash H_0$ (17)
by (str:Subsumption) with (16) and (8).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_2 \vdash H_0$ (18)
by renaming the environment.

Therefore, by (15) and (18) we conclude (noting that T is empty meaning that we also have $\Delta_T = \cdot$).

Case (T:Function) - is a value.

Case (T:Cap-Elim) - Not applicable, environment not closed.

Case (T:Cap-Stack) - We have:

$\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A_0 :: A_1 \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0$ (2)
$H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \cdot T$ (3)
by hypothesis.

$\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A_0 \dashv \Delta, A_1$ (4)
by inversion on (T:Cap-Stack) on (1).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (5)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_0 \dashv \Delta, A_1$ (6)
$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (7)
for some Δ_1, Γ_1.
by induction hypothesis on (2), (3) and (4).

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_0 :: A_1 \dashv \Delta$ (8)
by (T:Cap-Stack) on (6).

Therefore, by (5), (7) and (8) we conclude.

Case (T:CAP-UNSTACK) - We have:

$\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A_0 \dashv \Delta, A_1$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0$ (2)
$H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \cdot T$ (3)
by hypothesis.

$\Gamma_0 \mid \Delta_0 \vdash e_0 : A_0 :: A_1 \dashv \Delta$ (4)
by inversion on (T:CAP-UNSTACK) on (1).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (5)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_0 :: A_1 \dashv \Delta$ (6)
$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (7)
for some Δ_1, Γ_1.
by induction hypothesis on (2), (3) and (4).

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_0 \dashv \Delta, A_1$ (8)
by (T:CAP-UNSTACK) on (6).

Therefore, by (5), (7) and (8) we conclude.

Case (T:SUBSUMPTION) - We have:

$\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A_1 \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0$ (2)
$H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \cdot T$ (3)
by hypothesis.

$\Gamma_0 \vdash \Delta_0, \Delta_T <: \Delta_0', \Delta_T'$ (4)
$\Gamma_0 \mid \Delta_0', \Delta_T' \vdash e_0 : A_0 \dashv \Delta'$ (5)
$\Gamma_0 \vdash A_0 <: A_1$ (6)
$\Gamma_0 \vdash \Delta' <: \Delta$ (7)
by inversion on (T:SUBSUMPTION) with (1).

$\Gamma_0 \mid \Delta_0', \Delta_T', \Delta_2 \vdash H_0$ (8)
by (STR:SUBSUMPTION) with (2) and (4).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T', \Delta_2 \vdash H_1$ (9)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_0 \dashv \Delta'$ (10)
$\Gamma_0, \Gamma_1 \mid \Delta_T' \vdash T$ (11)
for some Δ_1, Γ_1.
by induction hypothesis on (3), (5) and (8).

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_1 \dashv \Delta$ (12)
by (T:SUBSUMPTION) with (6), (7) and (10).

$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (13)
by (WF:PROGRAM) and (T:SUBSUMPTION) with (11) and (4).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (14)
by (2) and (4) since Δ_T is disjoint, its support on H_1 remains by (STR:SUBSUMPTION) from (2).
Therefore, by (14), (12) and (13) we conclude.

Case (T:TAG) - is a value.

Case (T:Case) - We have:

$\Gamma_0 \mid \Delta_0 \vdash \mathsf{case}\ \mathtt{t}_i\#v_i\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end} : A \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0$ (2)
$H_0 \,;\, \mathcal{E}[\mathsf{case}\ \mathtt{t}_i\#v_i\ \mathsf{of}\ \overline{\mathtt{t}_j\#x_j \to e_j}\ \mathsf{end}] \mapsto H_0 \,;\, \mathcal{E}[e_i\{v_i/x_i\}]$ (3)

by hypothesis, (D:Case) where $\mathcal{E} = \Box$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.

$\Gamma_0 \mid \Delta_0 \vdash \mathtt{t}_i\#v_i : \sum_i \mathtt{t}_i\#A_i \dashv \Delta'$ (4)
$\overline{\Gamma_0 \mid \Delta', x_i : A_i \vdash e_i : A \dashv \Delta}$ (5)
$i \leq j$ (6)

by inversion on (D:Case) with (1).

$\Gamma_0 \vdash \Delta_0 <: \Delta_v, \Delta'$ (7)
$\Gamma_0 \mid \Delta_v \vdash \mathtt{t}_i\#v_i : \sum_i \mathtt{t}_i\#A_i \dashv \cdot$ (8)

by (Values Lemma) with (4).

$\Gamma_0 \mid \Delta_v \vdash v_i : A_i \dashv \cdot$ (9)
for some i.

by (Values Inversion) with (8).

$\Gamma_0 \mid \Delta_v, \Delta' \vdash v_i : A_i \dashv \Delta$ (10)

by (T:Frame) on (9) with Δ'.

$\Gamma_0 \mid \Delta_0 \vdash e_i\{v_i/x_i\} : A \dashv \Delta$ (11)

by (Substitution Lemma - Linear) with (10) and (5), for some i.

By making:
$\Gamma_1 = \cdot$
$\Delta_1 = \Delta_0$
We trivially have:

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_i\{v_i/x_i\} : A \dashv \Delta$ (12)

by (11).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_2 \vdash H_0$ (13)

by (2).

Thus, by (12) and (13) we conclude (noting that T is empty meaning that we also have $\Delta_T = \cdot$).

Case (T:Alternative-Left) - We have:

$\Gamma_0 \mid \Delta_0, A_0 \oplus A_1, \Delta_T \vdash e_0 : A_2 \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, A_0 \oplus A_1, \Delta_T, \Delta_2 \vdash H_0$ (2)
$H_0 \,;\, e_0 \mapsto H_1 \,;\, e_1 \cdot T$ (3)

by hypothesis.

$\Gamma_0 \mid \Delta_0, A_0, \Delta_T \vdash e_0 : A_2 \dashv \Delta$ (4)
$\Gamma_0 \mid \Delta_0, A_1, \Delta_T \vdash e_0 : A_2 \dashv \Delta$ (5)

by inversion on (T:Alternative-Left) with (1).

By (Store Typing Inversion) on (2), we have that either:

- $\Gamma_0 \mid \Delta_0, A_0, \Delta_T, \Delta_2 \vdash H_0$ (1.1)

by sub-case hypothesis.

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (1.2)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_2 \dashv \Delta$ (1.3)
$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (1.4)
for some Δ_1, Γ_1.

by induction hypothesis with (1.1), (3) and (4).

Therefore, we conclude.
• $\Gamma_0 \mid \Delta_0, A_1, \Delta_T, \Delta_2 \vdash H_0$ (2.1)
analogous to previous sub-case but using (5).

Thus, we conclude.

Case (T:INTERSECTION-RIGHT) - We have:

$\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A_2 \dashv \Delta, A_0 \& A_1$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0$ (2)
$H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \cdot T$ (3)
by hypothesis.

$\Gamma_0 \mid \Delta_0 \vdash e_0 : A_2 \dashv \Delta, A_0$ (4)
$\Gamma_0 \mid \Delta_0 \vdash e_0 : A_2 \dashv \Delta, A_1$ (5)
by inversion on (T:INTERSECTION-RIGHT) with (1).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (6)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_2 \dashv \Delta, A_0$ (7)
$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (8)
by induction hypothesis with (2), (3) and (4).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (9)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_2 \dashv \Delta, A_1$ (10)
$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (11)
by induction hypothesis with (2), (3) and (5).

$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_2 \dashv \Delta, A_0 \& A_1$ (12)
by (T:INTERSECTION-RIGHT) on (7) and (10).

Thus, by (11), (12) and (9) we conclude.

Case (T:FRAME) - We have:

$\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash e_0 : A \dashv \Delta, \Delta_2$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2, \Delta_3 \vdash H_0$ (2)
$H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \cdot T$ (3)
by hypothesis.

$\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A \dashv \Delta$ (4)
by inversion on (T:FRAME) with (1).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2, \Delta_3 \vdash H_1$ (5)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta$ (6)
$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (7)
by induction hypothesis on (2), (3) and (4).

$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_2 \vdash e_1 : A \dashv \Delta, \Delta_2$ (8)
by (T:FRAME) on (7) using Δ_2.

Therefore, by (5), (8), and (7) we conclude.

Case (T:LET) - We have two reductions:

1. **Sub-Case:** $\mathcal{E} = \square$

   $\Gamma_0 \mid \Delta_0 \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta$ (1)

   $\Gamma_0 \mid \Delta_0, \Delta_2 \vdash H_0$ (2)

   $H_0\ ;\ \mathcal{E}[\mathsf{let}\ x = v\ \mathsf{in}\ e\ \mathsf{end}] \ \mapsto\ H_0\ ;\ \mathcal{E}[e\{v/x\}]$ (3)

   by hypothesis, (D:LET) where $\mathcal{E} = \square$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.

   $\Gamma_0 \mid \Delta_0 \vdash v : A_0 \dashv \Delta'$ (5)

   $\Gamma_0 \mid \Delta', x : A_0 \vdash e : A_1 \dashv \Delta$ (6)

   by inversion on (T:LET) with (1).

   $\Gamma_0 \vdash \Delta_0 <: \Delta_v, \Delta'$ (7)

   $\Gamma_0 \mid \Delta_v \vdash v : A_0 \dashv \cdot$ (8)

   by (Values Lemma) with (4).

   $\Gamma_0 \mid \Delta_v, \Delta' \vdash v : A_0 \dashv \Delta'$ (9)

   by (T:FRAME) with (8).

   $\Gamma_0 \mid \Delta_v, \Delta' \vdash e\{v/x\} : A_1 \dashv \Delta$ (10)

   by (Substitution Lemma - Linear) with (6) and (9).

   $\Gamma_0 \mid \Delta_v, \Delta', \Delta_2 \vdash H_0$ (11)

   by (STR:SUBSUMPTION) with (2) and (7).

   Therefore, by (Weakening) with $\Gamma_1 = \cdot$ and by (10) and (11) we conclude.

2. **Sub-Case:** $\mathcal{E} = (\mathsf{let}\ x = \mathcal{E}'[e_0]\ \mathsf{in}\ e_2\ \mathsf{end})$

   $\Gamma_0 \mid \Delta_0, \Delta_T \vdash \mathsf{let}\ x = \mathcal{E}'[e_0]\ \mathsf{in}\ e_2\ \mathsf{end} : A_1 \dashv \Delta$ (1)

   $\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0$ (2)

   $H_0\ ;\ \mathcal{E}'[e_0] \ \mapsto\ H_1\ ;\ \mathcal{E}'[e_1] \cdot T$ (3)

   by hypothesis.

   $\Gamma_0 \mid \Delta_0, \Delta_T \vdash \mathcal{E}'[e_0] : A_0 \dashv \Delta'$ (4)

   $\Gamma_0 \mid \Delta', x : A_0 \vdash e_2 : A_1 \dashv \Delta$ (5)

   by inversion on (T:LET) with (1).

   $\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (6)

   $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash \mathcal{E}'[e_1] : A_0 \dashv \Delta'$ (7)

   $\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (8)

   by induction hypothesis on (2), (4) and (5).

   $\Gamma_0, \Gamma_1 \mid \Delta', x : A_0 \vdash e_2 : A_1 \dashv \Delta$ (8)

   by (Weakening) on (6).

   $\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash \mathsf{let}\ x = \mathcal{E}'[e_1]\ \mathsf{in}\ e_2\ \mathsf{end} : A_1 \dashv \Delta$ (9)

   by (T:LET) with (7) and (8).

   Therefore, by (6), (8), (9) we conclude.

Case (T:FORK) - We have:

$\Gamma_0 \mid \Delta_T \vdash \mathsf{fork}\ e : ![] \dashv \cdot$ (1)
$\Gamma_0 \mid \Delta_T, \Delta_2 \vdash H_0$ (2)
$H_0\ ;\ \mathcal{E}[\mathsf{fork}\ e] \ \mapsto\ H_0\ ;\ \mathcal{E}[\{\}] \cdot e$ (3)
by hypothesis, with (D:Fork) where $\mathcal{E} = \square$.
$\Gamma_0 \mid \Delta_T \vdash e : ![] \dashv \cdot$ (4)
by inversion on (T:Fork).
$\Gamma_0 \mid \cdot \vdash \{\} : ![] \dashv \cdot$ (5)
by (T:Record).

Thus, by making:
$\Delta_1 = \cdot$ (6)
We conclude by (2), (4) and (5).

Case (T:Forall-Loc-Val), (T:Forall-Type-Val) - Are values.

Case (T:TypeOpenBind), (T:LocOpenBind) - Not applicable, environment not closed.

Case (T:LocOpenCap) - We have:

$\Gamma_0 \mid \Delta_0, \Delta_T, \exists l.A_0 \vdash e_0 : A_1 \dashv \Delta$ (1)
$\Gamma_0 \mid \Delta_0, \Delta_T, \exists l.A_0, \Delta_2 \vdash H_0$ (2)
$H_0\ ;\ e_0 \ \mapsto\ H_1\ ;\ e_1 \cdot T$ (3)
by hypothesis.
$\Gamma_0, t : \mathbf{loc} \mid \Delta_0, \Delta_T, A_0 \vdash e_0 : A_1 \dashv \Delta$ (4)
by inversion on (t:LocOpenCap) with (1).
$\Gamma_0 \mid \Delta_0, \Delta_T, A_0\{\rho/l\}, \Delta_2 \vdash H_0$ (5)
by (Store Typing Inversion) with (2).
$\Gamma_0 \mid \Delta_0, \Delta_T, A_0\{\rho/l\} \vdash e_0 : A_1 \dashv \Delta$ (6)
by (Substitution Lemma - Location Variable) with ρ and (4).
$\Gamma_0, \Gamma_1 \mid \Delta_1, \Delta_T, \Delta_2 \vdash H_1$ (7)
$\Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A_0 \dashv \Delta'$ (8)
$\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$ (9)
by induction hypothesis with (3), (5) and (6).

Thus, we conclude.

Case (T:TypeOpenCap) - Analogous to (T:LocOpenCap).

Case (T:Unlock-Guarantee) - We have:

$\Gamma_0 \mid \Delta_0,\ A_0,\ A_0;A_1 \vdash \mathsf{unlock}\ \overline{v} : ![] \dashv \Delta_0, A_1$ (1)
$\Gamma_0 \mid \Delta_0,\ A_0,\ A_0;A_1, \Delta_2 \vdash H_0, \overline{\rho^\bullet \hookrightarrow v}$ (2)
$H_0, \overline{\rho^\bullet \hookrightarrow v}\ ;\ \mathcal{E}[\mathsf{unlock}\ \overline{\rho}] \ \mapsto\ H_0, \overline{\rho \hookrightarrow v}\ ;\ \mathcal{E}[\{\}]$ (3)
by hypothesis with (T:Unlock-Guarantee) where $\mathcal{E} = \square$ and where $\Delta_T = \cdot$ since there is no resulting new thread from this step.
$\Gamma_0 \mid \Delta_0, A_1 \vdash \{\} : ![] \dashv \Delta_0, A_1$ (4)
by (T:Record), (T:Pure), and (T:Frame) with Δ_0, A_1.

If we make:
$H_0 = H, H'$ (5)
then:
$\mathbf{locs}(A_0) = \mathbf{locs}(A_1) = \overline{\rho}$ (6)
$\overline{\rho^\bullet \hookrightarrow v} \in H, \overline{\rho^\bullet \hookrightarrow v}$ (7)
$\Gamma \mid A_0 \vdash H, \overline{\rho^\bullet \hookrightarrow v}$ (8)
$\overline{\rho \hookrightarrow v} \in H, \overline{\rho \hookrightarrow v}$ (9)
$\Gamma \mid A_0 \vdash H, \overline{\rho \hookrightarrow v}$ (10)
$\Gamma \mid \Delta_0, A_1 \vdash H, H', \overline{\rho \hookrightarrow v}$ (11)
by (Store Typing Inversion) with (2).
Therefore, we conclude by (4) and (11) noting (5).

Case (T:LOCK-RELY) - We have:

$\Gamma_0 \mid \Delta_0, A_0 \Rightarrow A_1 \vdash \mathsf{lock}\ \overline{v} : ![] \dashv \Delta_0, A_0, A_1$ (1)
$\Gamma_0 \mid \Delta_0, A_0 \Rightarrow A_1, \Delta_2 \vdash H_0, \overline{\rho \hookrightarrow v}$ (2)
$H_0, \overline{\rho \hookrightarrow v}\ ;\ \mathcal{E}[\mathsf{lock}\ \overline{\rho}] \ \mapsto\ H_0, \overline{\rho^\bullet \hookrightarrow v}\ ;\ \mathcal{E}[\{\}]$ (3)
by hypothesis, with (T:LOCK-RELY) where $\mathcal{E} = \square$, and where $\Delta_T = \cdot$ since there is no resulting new thread from this step.
$\Gamma_0 \mid \Delta_0, A_0, A_1 \vdash \{\} : ![] \dashv \Delta_0, A_0, A_1$ (4)
by (T:RECORD) and (T:FRAME) with Δ_0, A_0, A_1.
$\mathbf{locs}(A_0) = \overline{\rho}$ (5)
by inversion on (T:LOCK-RELY) since $\overline{\rho} \in A_0$.
$\Gamma_0 \mid A_0 \vdash H', \overline{\rho \hookrightarrow v}$ (6)
$H_0 = H, H'$ (7)
by (Store Typing Inversion) with (2), since we know that the locations $\overline{\rho}$ (thus the rely type must be supported by the heap since protocols must be introduced via (STR:SUBSUMPTION)).
We must show that:
$\Gamma_0 \mid \Delta_0, A_0, A_1, \Delta_2 \vdash H, H', \overline{\rho^\bullet \hookrightarrow v}$
Since $A_0 \Rightarrow A_1$ is well-formed and also obeys safe protocol composition, we have that A_1 must have one of the following structures:

- $A_1 = (A_1'; A_1'')$ (8.1)
  Thus, if we have a heap such that:
  $\Gamma_0 \mid A_1' \vdash H'', \overline{\rho \hookrightarrow v}$ (8.2)
  $\Gamma_0 \mid \Delta_0, A_1'', \Delta_2 \vdash H'', H, \overline{\rho \hookrightarrow v}$ (8.3)
  by (Composition Preservation) and (2) since we know that all protocols must still conform after stepping.
  $\Gamma_0 \mid \Delta_0, A_0, A_1, \Delta_2 \vdash H, H', \overline{\rho^\bullet \hookrightarrow v}$ (8.4)
  by (STR:LOCKED) using (6), (7), (8.2) and (8.3).
- $A_1 = \forall l.A_1'$ (9.1)
  Thus, if we have a heap such that (similar to previous case):
  $\Gamma_0, l : \mathbf{loc} \mid A_1' \vdash H'', \overline{\rho \hookrightarrow v}$ (9.2)
  Then by (STR:LOCKED):
  $\Gamma_0, l : \mathbf{loc} \mid \Delta_0, A_0, A_1' \vdash H, H', \overline{\rho^\bullet \hookrightarrow v}$ (9.3)

and:

$\Gamma_0 \mid \Delta_0, A_0, \forall l.A'_1 \vdash H, H', \overline{\rho^\bullet \hookrightarrow v}$ (9.4)

by (STR:FORALLLOCS).

- $A_1 = \forall X <: A'.A'_1$ (10.1)

  $\Gamma_0 \mid \Delta_0, A_0, \forall X <: A'.A'_1 \vdash H, H', \overline{\rho^\bullet \hookrightarrow v}$ (10.2)

  similarly to the previous case but using (STR:FORALLTYPES).

Therefore, we conclude by (4) and (8.4), (9.4), (10.2).

□

A.2.10 Progress

We define progress over closed programs (which includes an arbitrarily large but finite number of thread, T) and expressions (e).

Theorem 18 (Program Progress)**.** If we have

$$\Gamma \mid \Delta \vdash T_0 \qquad \texttt{live}(T_0)$$

and if exists H_0 such that $\Gamma \mid \Delta \vdash H_0$ then

$$H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$$

Proof. We proceed by induction on the typing derivation of $\Gamma \mid \Delta \vdash T_0$.

Case (WF:PROGRAM) - We have:

$\Gamma \mid \Delta \vdash T_0$ (1)
$\texttt{live}(T_0)$ (2)
by hypothesis.
$\Gamma \mid \Delta_i \vdash e_i : ![] \dashv \cdot$ (3)
$i \in \{0, ..., n\}$ (4)
$n \geq 0$ (5)
by inversion on (1) with (WF:PROGRAM).
$T_0 = e_0 \cdot ... \cdot e_n$ (6)
(remember that the order of the threads is not important)
$\Delta = \Delta_0, ..., \Delta_n$ (7)
by decomposing Δ into smaller typing environments.

Then for some arbitrary j such that:
$j \in \{0, ..., n\}$ (8)
$\texttt{live}(e_j)$ (9)
by (2) and the definition of $\texttt{live}(T_0)$ there must be at least one thread that is `live`.

Then by (Expression Progress) with e_j on (3), we have that either:

- e_j is a value, or (10.1)
- if exists H_0 such that $\Gamma \mid \Delta \vdash H_0$ (note that $\Delta_j \in \Delta$ by (7)) then either:
  - ⋄ (steps) $H_0 \; ; \; e_j \; \mapsto \; H_1 \; ; \; e'_j \; \cdot \; T'$ (10.2)
  - ⋄ (waits) $\texttt{Wait}(H_0, e_j)$ (10.3)

We have that cases (10.1) and (10.3) cannot occur because we picked e_j such that $\texttt{live}(e_j)$.
Therefore, we have (10.2) which by (D:THREAD) with the remaining threads of T_0 still steps.
Thus, we conclude since we have that the set of threads steps.

□

Theorem 19 (Expression Progress)**.** If we have

$$\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$$

then we have that either:

- e_0 is a value, or;
- if exists H_0 such that $\Gamma \mid \Delta, \Delta_0 \vdash H_0$ then either:
  - (steps) $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$
  - (waits) $\texttt{Wait}(H_0, e_0)$

Proof. We proceed by induction on the typing derivation of $\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$.

Case (T:REF), (T:PURE), (T:UNIT) - are values.

Case (T:PURE-READ), (T:LINEAR-READ), (T:PURE-ELIM) - not applicable due to the environment not being closed.

Case (T:NEW) - We have:

$\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta_1$ (1)
by hypothesis.

$\mathsf{new}\ v$ is not a value. If exists:
$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (2)
Then the expression steps using (D:NEW) with $\mathcal{E} = \square$, i.e.:
$\square[\mathsf{new}\ v]$ (3)
Thus, we conclude by stepping using (D:NEW).

Case (T:DELETE) - We have:

$\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v : \exists l.A \dashv \Delta_1$ (1)
by hypothesis.

$\Gamma \mid \Delta_0 \vdash v : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \Delta_1$ (2)
by inversion on (T:DELETE) with (1).

$\Gamma \mid \Delta_v \vdash v : \exists l.(!\mathbf{ref}\ l :: \mathbf{rw}\ l\ A) \dashv \cdot$ (3)
$\Gamma \vdash \Delta_0 <: \Delta', \Delta_v$ (4)
by (Values Inversion) with (2).

$\Gamma \mid \Delta_v \vdash v :!\mathbf{ref}\ \rho :: \mathbf{rw}\ \rho\ A \dashv \cdot$ (4)
by (Values Inversion) with (3).

$\Gamma \mid \Delta_v \vdash v :!\mathbf{ref}\ \rho \dashv \mathbf{rw}\ \rho\ A$ (5)
by (Values Inversion) with (4).

$\Gamma \mid \Delta'_v \vdash v : \mathbf{ref}\ \rho \dashv \cdot$ (6)
$\Gamma \vdash \Delta_v <: \Delta'_v, \mathbf{rw}\ \rho\ A$ (7)
by (Values Inversion) with (5).

$\Delta'_v = \cdot$ (8)
$\rho \in \Gamma$ (9)
$v = \rho$ (10)
by (Values Inversion) on (6).

Since $\mathsf{delete}\ v$ is not a value. If exists:
$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (11)

Then we must have that:
$\rho^? \hookrightarrow v' \in H_0$ (12)
by (Store Typing Inversion) on the ρ binding and (11), (4) and (7).
Then the expression steps using (D:DELETE) with $\mathcal{E} = \square$, i.e.:
$\square[\mathsf{delete}\ v]$ (3)
Thus, we conclude by stepping using (D:DELETE).

Case (T:ASSIGN) - We have:

$\Gamma \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_1, \mathbf{rw}\ \rho\ A_0$ (1)
by hypothesis.
$\Gamma \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta_2$ (2)
$\Gamma \mid \Delta_2 \vdash v_0 : \mathbf{ref}\ \rho \dashv \Delta_1, \mathbf{rw}\ \rho\ A_1$ (3)
by inversion on (T:ASSIGN) with (1).
$v_0 = \rho$ (4)
by applying (Values Inversion) multiple times, similarly to the (T:DELETE) case, on (3).
$v_0 := v_1$ is not a value. If exists:
$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (5)
Then we must have that:
$\rho^? \hookrightarrow v' \in H_0$ (6)
by (Store Typing) definition on the ρ binding since the capability for ρ exists.
Then the expression steps using (D:ASSIGN) with $\mathcal{E} = \square$, i.e.:
$\square[v_0 := v_1]$ (7)
Thus, we conclude by stepping using (D:ASSIGN).

Case (T:DEREFERENCE-LINEAR) - We have:

$\Gamma \mid \Delta_0 \vdash !v : A \dashv \Delta_1, \mathbf{rw}\ \rho\ ![]$ (1)
by hypothesis.
$\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ \rho \dashv \Delta_1, \mathbf{rw}\ \rho\ A$ (2)
by inversion on (T:DEREFERENCE-LINEAR) with (1).
$v = \rho$ (3)
by (Values Inversion) on (2) as in previous cases.
$v_0 := v_1$ is not a value. If exists:
$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (5)
Then we must have that:
$\rho^? \hookrightarrow v' \in H_0$ (6)
by (Store Typing) definition on the ρ binding as in previous cases.
Then the expression steps using (D:DEREFERENCE) with $\mathcal{E} = \square$, i.e.:
$\square[!v]$ (7)
Thus, we conclude by stepping using (D:DEREFERENCE).

Case (T:DEREFERENCE-PURE) - similar to (T:DEREFERENCE-LINEAR).

Case (T:RECORD) - is a value.

Case (T:SELECTION) - We have:

$\Gamma \mid \Delta_0 \vdash v.\mathtt{f}_i : A_i \dashv \Delta_1$ (1)
by hypothesis.
$\Gamma \mid \Delta_0 \vdash v : [\overline{\mathtt{f} : A}] \dashv \Delta_1$ (2)
by inversion on (T:SELECTION) with (1).
$v = \{\overline{\mathtt{f} = v'}\}$ (3)
by (Values Lemma) and (Values Inversion) with (2).
$v.\mathtt{f}_i$ is not a value. If exists:
$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (5)
Then the expression steps using (D:SELECTION) with $\mathcal{E} = \Box$, i.e.:
$\Box[v.\mathtt{f}_i]$ (6)
Thus, we conclude by stepping using (D:SELECTION).

Case (T:APPLICATION) - We have:

$\Gamma \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_1$ (1)
by hypothesis.
$\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_2$ (2)
$\Gamma \mid \Delta_2 \vdash v_1 : A_0 \dashv \Delta_1$ (3)
by inversion on (T:APPLICATION) with (1).
$v_0 = \lambda x.e$ (4)
by (Values Lemma) and (Values Inversion) with (2).
$v_0\ v_1$ is not a value. If exists:
$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (5)
Then the expression steps using (D:SELECTION) with $\mathcal{E} = \Box$, i.e.:
$\Box[v_0\ v_1]$ (6)
Thus, we conclude by stepping using (D:APPLICATION).

Case (T:FUNCTION) - is a value.

Case (T:CAP-ELIM) - not applicable due to the environment not being closed.

Case (T:CAP-STACK), (T:CAP-UNSTACK) - immediate by direct application of the induction hypothesis on the inversion of the typing rule.

Case (T:FRAME) - We have:

$\Gamma \mid \Delta_0, \Delta_2 \vdash e_0 : A_0 \dashv \Delta_1, \Delta_2$ (1)
by hypothesis.
$\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1$ (2)
by inversion on (T:FRAME) with (1).
Then, by induction hypothesis on (2), we have that either:

- e_0 is a value, or; (3)
- if exists H_0 such that $\Gamma \mid \Delta, \Delta_0 \vdash H_0$ then either:
  - $\diamond$ (steps) $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$ (4)
  - $\diamond$ (waits) $\texttt{Wait}(H_0, e_0)$ (5)

Therefore, by making $\Delta_2 \in \Delta$ by (3), (4), and (5) we conclude.

Case (T:SUBSUMPTION) - We have:

$\Gamma \mid \Delta_0 \vdash e_0 : A_1 \dashv \Delta_3$ (1)

by hypothesis.

$\Gamma \vdash \Delta_0 <: \Delta_1$ (2)

$\Gamma \mid \Delta_1 \vdash e_0 : A_0 \dashv \Delta_2$ (3)

$\Gamma \vdash A_0 <: A_1$ (4)

$\Gamma \vdash \Delta_2 <: \Delta_3$ (5)

by inversion on (T:SUBSUMPTION) with (1).

Then, by induction hypothesis on (3), we have that either:

- e_0 is a value, or; (6)
- if exists H_0 such that $\Gamma \mid \Delta, \Delta_1 \vdash H_0$ then either:
  - $\diamond$ (steps) $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$ (7)
  - $\diamond$ (waits) $\texttt{Wait}(H_0, e_0)$ (8)

Furthermore, we know that if exists H_0' such that:

$\Gamma \mid \Delta, \Delta_0 \vdash H_0'$ (9)

then:

$\Gamma \mid \Delta, \Delta_1 \vdash H_0'$ (10)

by (Subtyping Store Typing) with (2) and (6).

Therefore, we conclude by (6), (7) and (8) (using H_0').

Case (T:TAG) - is a value.

Case (T:CASE) - We have:

$\Gamma \mid \Delta_0 \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_1$ (1)

by hypothesis.

$\Gamma \mid \Delta_0 \vdash v : \sum_i \mathtt{t}_i \# A_i \dashv \Delta_1$ (2)

$\overline{\Gamma \mid \Delta_1, x_i : A_i \vdash e_i : A \dashv \Delta_2}$ (3)

$i \leq j$ (4)

by inversion on (T:CASE) with (1).

$v = \mathtt{t}_i \# v_i$ (5)

by (Values Lemma) and (Values Inversion) with (2).

$\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end}$ is not a value. If exists:

$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (6)

Then the expression steps using (D:CASE) with $\mathcal{E} = \square$, i.e.:

$\square[\mathsf{case}\ v\ \mathsf{of}\ \overline{\mathtt{t}_j \# x_j \to e_j}\ \mathsf{end}]$ (7)

Thus, we conclude by stepping using (D:CASE).

Case (T:ALTERNATIVE-LEFT) - We have:

$\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1$ (1)
by hypothesis.
$\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1$ (2)
$\Gamma \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1$ (3)
by inversion on (T:ALTERNATIVE-LEFT) with (1).

If exists H such that:
$\Gamma \mid \Delta', \Delta_0, A_0 \oplus A_1 \vdash H$ (4)
By (Store Typing Inversion) due to (ST:ALTERNATIVE) on (4), we have that either (from $\oplus$ being commutative):
$\diamond$ $\Gamma \mid \Delta', \Delta_0, A_0 \vdash H$ (5)
Then by induction hypothesis on (2) we conclude.
$\diamond$ $\Gamma \mid \Delta', \Delta_0, A_1 \vdash H$ (6)
Then by induction hypothesis on (3) we conclude.
Therefore, we conclude.

Case (T:INTERSECTION-RIGHT) - immediate by applying the induction hypothesis on the inversion of the typing rule.

Case (T:FORALL-LOC-VAL), (T:FORALL-TYPE-VAL) - are values.

Case (T:LOCOPENCAP) - We have:

$\Gamma \mid \Delta_0, \exists l.A_1 \vdash e : A_2 \dashv \Delta_1$ (1)
by hypothesis.
$\Gamma, l : \mathbf{loc} \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1$ (2)
by inversion on (T:LOCOPENCAP) with (1).

If e is not a value, then if exists:
$\Gamma \mid \Delta, \Delta_0, \exists l.A_1 \vdash H_0$ (3)
$\Gamma \vdash A_1\{\rho/l\} <: \exists l.A_1$ (3)
by (Store Typing Inversion) on (3) and (ST:PACKLOC) with $\exists l.A_1$.
$\Gamma \mid \Delta_0, A_1\{\rho/l\} \vdash e : A_2 \dashv \Delta_1$ (4)
by (Substitution Lemma) on (2) with ρ.

Thus, we conclude by induction hypothesis on (4).

Case (T:TYPEOPENCAP) - similar to (T:LOCOPENCAP).

Case (T:TYPEOPENBIND), (T:LOCOPENBIND) - not applicable due to the environment not being closed.

Case (T:FORK) - We have:

$\Gamma \mid \Delta_0 \vdash \mathsf{fork}\ e : ![] \dashv \cdot$ (1)
by hypothesis.

$\mathsf{fork}\ e$ is not a value. If exists:

$\Gamma \mid \Delta, \Delta_0 \vdash H_0$ (2)
Then the expression steps using (D:Fork) with $\mathcal{E} = \square$, i.e.:
$\square[\mathsf{fork}\ e]$ (3)
Thus, we conclude by stepping using (D:Fork).

Case (T:Lock-Rely) - We have:

$\Gamma \mid \Delta_0, A_0 \Rightarrow A_1 \vdash \mathsf{lock}\ \overline{v} : ![] \dashv \Delta_1, A_0, A_1$ (1)
by hypothesis.

$\mathsf{lock}\ \overline{v}$ is not a value. If exists:
$\Gamma \mid \Delta, \Delta_0, A_0 \Rightarrow A_1 \vdash H_0$ (2)
By (Store Typing) definition, we have that either:
• All the locks of the locations contained in A_0 are available. Then the expression steps by (D:Lock) with $\mathcal{E} = \square$.
• Some of the locks of the locations of A_0 are unavailable. Then we have that the expression waits, i.e. we have $\texttt{Wait}(H_0, \mathsf{lock}\ \overline{v})$.
Thus, we conclude.

Case (T:Unlock-Guarantee) - We have:

$\Gamma \mid \Delta_0, A_0, A_0; A_1 \vdash \mathsf{unlock}\ \overline{v} : ![] \dashv \Delta_0, A_1$ (1)
by hypothesis.

$\mathsf{unlock}\ \overline{v}$ is not a value. If exists:
$\Gamma \mid \Delta_0, A_0, A_0; A_1 \vdash H_0$ (2)
Then the expression steps using (D:Unlock) with $\mathcal{E} = \square$, i.e.:
$\square[\mathsf{unlock}\ \overline{v}]$ (3)
Thus, we conclude by stepping using (D:Unlock).

Case (T:Let) - We have:

$\Gamma \mid \Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_2\ \mathsf{end} : A \dashv \Delta_2$ (1)
by hypothesis.
$\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1$ (2)
$\Gamma \mid \Delta_1, x : A_0 \vdash e_2 : A_1 \dashv \Delta_2$ (3)
by inversion on (T:Let) with (1).
By induction hypothesis on (2), we have that either:
• e_0 is a value; (6)
then by (D:Let) the expression transitions, with $\mathcal{E} = \square$.
• if exists H_0 such that $\Gamma \mid \Delta, \Delta_0 \vdash H_0$ then either:
◇ (steps) $H_0\ ;\ e_0 \mapsto H_1\ ;\ e_1 \cdot T$ (7)
then the expression steps by (D:Thread) (with an empty initial thread pool) and $\mathcal{E} = (\mathsf{let}\ x = \mathcal{E}'[e_1]\ \mathsf{in}\ e_2\ \mathsf{end})$ and $\mathcal{E}' = \square$.
◇ (waits) $\texttt{Wait}(H_0, e_0)$ (8)
then $\texttt{Wait}(H_0, \mathsf{let}\ x = e_0\ \mathsf{in}\ e_2\ \mathsf{end})$ by the definition of `Wait` since we have: $\texttt{Wait}(H_0, \mathcal{E}[e_0])$
where $\mathcal{E} = (\mathsf{let}\ x = \mathcal{E}'[e_0]\ \mathsf{in}\ e_2\ \mathsf{end})$ where $\mathcal{E}' = \square$.

Thus, we conclude.

□