Composing Interfering Abstract Protocols

Filipe Militão[1,2] **Jonathan Aldrich**[1]
Luís Caires[2]

April 2016
CMU-CS-16-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.
[2]Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Caparica, Portugal.

This document is a companion technical report of the paper, "Composing Interfering Abstract Protocols", to appear in the Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2016. It includes the complete technical development, detailed proofs, and additional examples and discussions that are not present in the ECOOP paper.

This material is supported in part by AFRL and DARPA under agreement #FA8750-16-2-0042, and by NSA lablet contract #H98230-14-C-0140; by NOVA LINCS UID/CEC/04516/2013; and 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 CMU. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of any of the sponsors.

Keywords: aliasing, interference control, rely-guarantee, typestates

Abstract

The undisciplined use of shared mutable state can be a source of program errors when aliases unsafely interfere with each other. While protocol-based techniques to reason about interference abound, they do not address two practical concerns: the decidability of protocol composition and its integration with protocol abstraction. We show that our composition procedure is decidable and that it ensures safe interference even when composing abstract protocols. To evaluate the expressiveness of our protocol framework for ensuring safe shared memory interference, we show how this same protocol framework can be used to model safe, typeful message-passing concurrency idioms.

Contents

1 Introduction

The interactions that can occur via shared mutable state can be a source of program errors. When different clients access the same mutable state, their actions can potentially *interfere*. For instance, the programmer may wrongly assume that a cell holds a particular type, when another part of the program has changed that cell to hold a different type. When this happens, the program may fault due to *unsafe* interference caused by unexpected actions through other aliases to that shared state. Thus, to reason about interference we must reason about how state is aliased and how the different aliases use the shared state.

Our technique builds on the use of linear capabilities [1] to track type-changing resource mutation within the framework of a linear type system. However, relying solely on linearity is often too restrictive. For instance, linearity enforces exclusive ownership of mutable state, which is incompatible with multithreading—i.e. linearity forbids sharing. To allow sharing, we extend the concept of *rely-guarantee protocols* [21]. By sequencing *steps* of "rely⇒guarantee" actions, each protocol characterizes an alias's local, isolated perspective on interactions with a piece of shared state:

$$\underbrace{\text{“what I } \textit{assume} \text{ about the state”} \Rightarrow \text{“what I } \textit{guarantee} \text{ about the state”}}_{\textit{current}\text{ step}} \;;\; \textit{next}\text{ step}$$

Since the interactions performed by an alias may change over time, a rely-guarantee protocol is formed by a sequence of steps that specify each interfering action. Each step *relies* on the shared state having some type and then, after some private actions, *guarantees* that the shared state will now have some other type, which becomes visible to other aliases. By 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 [16]. Naturally, not all protocols compose safely. While a protocol describes its own actions on a piece of shared state, protocol *composition* will ensure that those actions are safe w.r.t. the actions that can be done via other existing (and even future) protocols over that state. Composition is safe only if the set of protocols accounts for all possible run-time action interleavings.

Our main contribution is a decidable protocol composition procedure that also allows abstract protocols to be composed. We break down our contributions as follows:

- We adapt the existing constructs of rely-guarantee protocols [21] to work in a system with concurrent runtime semantics, and show that rely-guarantee protocols are useful to reason about safe interference in the concurrent setting.

- We give an axiomatic definition of protocol composition. We show that this procedure can be implemented in a sound and complete (w.r.t. the formal definition) algorithm that terminates on all legal inputs.[1] The protocol composition algorithm is implemented in a prototype.[2]

[1] Note that we have not proven the decidability of the entire type system, but only of the protocol composition algorithm which is at its core. The remainder of the type system is more conventional and we did not encounter difficulties with decidability when implementing similar rules in our prior work [21].

[2] See: http://www.cs.cmu.edu/~foliveir/protocol-composition.html

$x \in$ Variables $\quad$ $\mathtt{t} \in$ Tags $\quad$ $\mathtt{f} \in$ Fields $\quad$ $\rho \in$ Location Constants

e	::=	v	(value)			lock $\overline{v}$	(lock locations)
	|	$v.\mathtt{f}$	(field selection)		|	unlock $\overline{v}$	(unlock locations)
	|	$v\ v$	(application)		|	fork e	(spawn thread)
	|	let $x = e$ in e end	(let)				
	|	new v	(cell creation)	v	::=	ρ	(address)
	|	delete v	(cell deletion)		|	x	(variable)
	|	$!v$	(dereference)		|	$\lambda x.e$	(function)
	|	$v := v$	(assign)		|	$\{\overline{\mathtt{f} = v}\}$	(record)
	|	case v of $\overline{\mathtt{t}\#x \to e}$ end	(case)		|	$\mathtt{t}\#v$	(tagged value)

Notes: $\overline{Z}$ is a potentially empty sequence of Z elements. ρ is not source-level.

Figure 1: Values (v) and expressions (e).

- We show that our use of type abstraction and bounded quantification at the protocol level enables us to model new, and more general, polymorphic forms of safe modular shared state interactions.
- We prove our system sound through progress and preservation theorems that show the absence of unsafe interference in correctly typed programs. Our design ensures *memory safety* and *data-race freedom*, where linear resources are shared via protocol composition (a partial commutative monoid [19, 8]).
- We evaluate the expressiveness of our system by discussing how our core shared memory protocol framework is capable of expressing safe, typeful message-passing idioms.

Next, we briefly introduce the language that "hosts" our protocols, with the remaining text focused on discussing new protocol-level features. Sections 2 and 3 introduce our novel definition of protocol composition and its extensions to support abstract protocols. Section 4 discusses technical results. The paper ends with discussions of expressiveness, related work, and conclusions.

1.1 Preliminaries: Language Overview

Our language supports fork/join concurrency combined with lock-based mutual exclusion, where all threads share a common heap. We use the variant of the polymorphic λ-calculus shown in Fig. 1. For convenience, the grammar is let-expanded [31] so that all constructs, except let, are defined over values. The language includes first-class functions (λ), records ($\{\overline{\mathtt{f} = v}\}$) that label a $v$alue as $\mathtt{f}$, and tagged values ($\mathtt{t}\#v$) to mark a $v$alue with a $\mathtt{t}$ag. Standard constructs are used for field selection, application, let blocks, memory allocation, deletion, assignment, dereference, and case analysis. "lock $\overline{v}$" atomically locks a non-empty set of locations (ensuring both mutual exclusion and forbidding re-entrant uses) and analogously with "unlock $\overline{v}$". "fork e" executes the $e$xpression

```
// assume 'y' in scope                                          y : !ref l, rw l int
  y := "ok!";                                                  y : !ref l, rw l string
  let x = y in                                          x : !ref l, y : !ref l, rw l string
    x := false;                                         x : !ref l, y : !ref l, rw l boolean
    delete x;                                            x : !ref l, y : !ref l
    !y  // Type Error: missing capability to location 'l'.
```

Figure 2: Tracking linear capabilities.

in a new thread while sharing access to the common global heap. The operational semantics are standard and, as such, are only shown in the appendix. They produce the standard evaluation of the language's constructs such as creating or deleting memory, spawning new threads, etc.

Mutable References We type mutable references by following the design proposed in $\mathbf{L}^3$ [1]. Therefore, a mutable cell is decomposed into two components: a pure *reference*, which can be freely copied; and a linear [13] *capability*, a resource that is used to track the contents of that cell. To link a reference to its respective capability, we use location-dependent types. 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 "$!\textbf{ref}\ l$" to mean a pure/duplicable (!) **ref**erence to a location l, where the information about the contents of that location is stored in the linear capability for l. The permission to access (e.g. 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 a location l that currently has contents of type A (the type of the value, given in "new v", that initializes the new cell). We depart from [1] by making capabilities typing artifacts that only exist at the level of typing. Consequently, capabilities are managed implicitly by the type system rather than manually manipulated by the programmer via language constructs. However, we may still need to associate a capability with another type. For this reason, we use the notion of *stacking* [22]. In $\exists l.(\ (!\textbf{ref}\ l)\ ::\ (\textbf{rw}\ l\ A)\ )$ we see that the capability to l is stacked on top of "$!\textbf{ref}\ l$" since the capability is to the right of the "::". This allows the capability to be bundled together with the **ref** type, but no action is required to unbundle them if they are needed separately. We refer to prior work [19, 22, 21, 1] for more details on the use of capabilities, locations, and stacking, as well as convenient abbreviations. Here, it suffices to assume that they are handled automatically by the type system, as our focus here is on safely sharing the linear resources.

In the scheme above, all the variables that reference the same location also share a single linear (i.e. "exclusively owned" or "unique") capability that tracks the changes to that location's contents (as shown in Fig. 2). However, this tracking relies on a compile-time approximation of how variables alias, which constrains how state can be used. Since the linear capability must be (linearly) threaded through the program, this scheme forbids aliasing idioms that require "simultaneous" access to aliased state, such as when multiple threads share access to a cell. To enable this form of sharing, we split a linear resource into multiple protocols. Each protocol controls how an alias interacts with the shared state, without depending on precise knowledge of which variables alias

$x \in$ Variables $\quad X \in$ Type Variables $\quad \rho \in$ Location Constants $\quad l \in$ Location Variables

$\mathtt{t} \in$ Tags $\quad \mathtt{f} \in$ Fields $\quad p ::= \rho \mid l \quad u ::= l \mid X \quad U ::= p \mid A$

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

Notes: we simplify $X[]$ to X; $\oplus$, $\&$, $*$, $+$ are commutative and associative.

Figure 3: Types (A).

each other. We will continue with a brief presentation of the base language, before diving into the details of our sharing mechanism in Section 2.

Types (Note that rely and guarantee types will only be discussed in the next section, when we present sharing). Our types (Fig. 3) follow the connectives of linear logic [13]. For this reason a function type uses $\multimap$ (instead of $\rightarrow$) to denote a linear function. The linear restriction can be lifted when the type is preceded by a "bang", such as in $!A$, which denotes a pure/duplicable type. Records are typed as $[\overline{\mathtt{f} : A}]$ where each field $\mathtt{f}$ types the value of the record with some type A. $\sum_i \mathtt{t}_i \# A_i$ 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 have separate existential and universal quantification over locations and types, since locations and types are of different kinds. Note that we leave $\forall/\exists$ as typing artifacts and as such they do not have corresponding constructs in the language. Quantification over types can provide a type bound (on the right of $<:$) and where **top** is assumed by default when the bound is omitted.

Our recursive types (assumed to be non-bottom types) are equi-recursive, interpreted co-inductively, and satisfy the usual folding/unfolding principle:

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

Recursive types may include a list of type/location parameters ($\overline{u}$) that are substituted by some type/location ($\overline{U}$) on unfold, besides unfolding the recursive type variable (X).

We use $\oplus$ to denote a union of alternative types, and $\&$ to denote a linear choice of different types. **none** is the empty resource. Finally we have "$A_0 :: A_1$" for stacking resource A_1 on top of A_0. Stacking is not commutative, so that it is not guaranteed that "$A_0 :: A_1 :: A_2$" can be used in place of "$A_0 :: A_2 :: A_1$". To enable resource commutation, we use $*$ such that "$A_0 :: (A_1 * A_2)$"

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

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

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

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

Notes: bounded variables of a construct and type/location variables of quantifiers must be fresh in the rule's conclusion.

Figure 4: Typing rules (selected).

and "$A_0 :: (A_2 * A_1)$" are interchangeable via subtyping. For clarity, we will review these type annotations as we present examples further below. Note that we do not syntactically distinguish resources (such as capabilities or protocols) from value-inhabited types. However, the type system ensures that types such as **none** can never be used to type a value. Indeed, even though "wrong" types can be assumed (such as in a function's argument) they can never actually be introduced as values.

Type System To enable automatic threading of resources, we use a type-and-effect system with judgments of 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, with effects resulting in the resources in Δ_1.

The typing environments are defined as follows:

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

Recursive type variables are given an $\to$ kind, where the left hand side tracks the type/location kind of a parameter of that recursive type.

Fig. 4 includes a few selected typing rules. Additional rules are shown below as they become relevant to the discussion on sharing, with the remainder left to the appendix. (T:Pure) types a value as pure if the value does not use any resources. If a variable is of a pure type, then (T:PureElim) allows the binding to be moved to the linear context with its type explicitly "banged" with !. (T:Frame) enables framing [30] resources that are not used by an expression, just threaded through the expression. Since a function, (T:Function), can depend on the 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 resources it captures is actually empty. (T:Application) is the standard rule. As discussed above, (T:New) and (T:Delete) manipulate types that abstract the underlying location that was created or that is to be deleted. (T:Assign) updates the contents of a location with the type of the newly assigned value. (T:Let) threads the effects of e_0 to the initial linear resources of e_1, sequencing the evaluation of the expressions as usual. (T:Dereference-Linear) removes the contents of a cell, leaving the residual "unit" type behind (the semantics leave the cell unchanged but unusable through typing). (T:LocOpenBind) illustrates the non-syntax-directed opening of existential location packages.

The subtyping rules are deferred to the appendix, but it suffices to know the subtyping judgment, $\Gamma \vdash A_0 <: A_1$, which states that A_0 is a subtype of A_1, meaning that A_0 can be used anywhere A_1 is expected. An analogous judgment governs subtyping between linear environments, $\Gamma \vdash \Delta_0 <: \Delta_1$. Thus, the (T:Subsumption) rule simply states that we can type an expression while using weaker assumptions and ensuring a stronger result and effect, as these types cannot break the conclusion's types expectations.

2 A Protocol for Modeling join

We begin by describing how non-abstracted protocols compose and how rely-guarantee protocols work in the concurrent setting. Our language supports the fork/join model of concurrency, in which a join is encoded via shared state interactions. There are two participants in this interaction: the `Main` thread and the `Forked` thread. The forked thread computes 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 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.[3] A thread scheduler could

[3]Each protocol must be aware of all valid states, as an omission would leave room for unsafe interference, such as when later re-splitting that protocol.

reduce or eliminate the spinning caused by this "busy-wait", but this is beyond the scope of our discussion. We define the two protocols as follows:

$$\texttt{F} \triangleq \mathsf{Wait} \Rightarrow \mathsf{Result}\ ;\ \textbf{none}$$
$$\texttt{M} \triangleq (\,\mathsf{Wait} \Rightarrow \mathsf{Wait}\ ;\ \texttt{M}\,) \oplus (\,\mathsf{Result} \Rightarrow \mathsf{Done}\ ;\ \mathsf{Done}\,)$$

Each protocol contains a sequence of steps that control the use of locks and specify the (type) assumptions on that locked state. Since locks hide all private actions, the protocols will only need to model the changes that become visible upon unlocking. These changes are bounded by 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 the ; operator.

The 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 legally 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**. This empty resource 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 with the same type, before later retrying `M`. Otherwise, if we find the cell containing a Result, we know that `F` has already terminated and can no longer access the shared state. In that situation, we mutate the cell to Done and unlock it so that each lock always has a matching unlock. Afterwards, the protocol continues as Done, a type that is just a regular linear capability. Thus, `M` recovered ownership of the shared state and Done can continue to be used without locking since the cell is no longer shared. We can now give concrete definitions for Wait, Result, and Done as types describing a single capability to location l as follows:

$$\mathsf{Wait} \triangleq \textbf{rw}\ l\ \texttt{Wait\#![]} \qquad \mathsf{Result} \triangleq \textbf{rw}\ l\ \texttt{Result\#}\textbf{int} \qquad \mathsf{Done} \triangleq \textbf{rw}\ l\ \texttt{![]}$$

Wait is a capability to location l containing a tagged value, where `Wait` is the tag and "![]" (a pure empty record) is the type of the value. Result is a capability for l containing an integer value tagged with `Result`. The two tags will enable us to distinguish between the Wait and Result alternatives by using standard case analysis. With Done the content is an empty pure record ("unit").

Each protocol describes an alias's local, isolated view of the evolution of the shared state. Thus, we can discuss the uses of each protocol independently. Because a protocol is a linear resource, the forked thread will "consume" or "capture" `F` in its context, making it unavailable to the main thread. As with any linear resource, `F` is tracked by the linear typing environment (Δ) and is either used by an expression or threaded through to the next expression. However, the forked thread and main thread can share the enclosing lexical typing environment (Γ) because it only contains pure/duplicable assumptions. A possible use of the `F` protocol follows.

```
3  fork                                  Γ = c : ref l, l : loc      | Δ = work : ![] ⊸ int, 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 = ·
```

Γ contains a reference (c) to the location (l) that is being shared by the protocol, and Δ contains a variable with the (linear) function that computes the `work` that the thread will do. (In this example both protocols refer to a well-known common location, but our technique also allows each protocol to ∃ abstract its locations.) Line 4 consumes the function `work` by calling it, storing the result in variable `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`. To type a lock we must map the locations listed in the lock to those contained in the rely type of the protocol. Well-formedness conditions on the protocols ensure that, at each step, the rely and the guarantee types refer the same set of locations so that no lock on a location goes without a respective unlock.

$$\frac{\overline{\Gamma \mid \cdot \vdash v : \mathbf{ref}\ p \dashv \cdot} \qquad \mathbf{locs}(A_0) = \overline{p}}{\Gamma \mid \Delta,\ A_0 \Rightarrow A_1 \vdash \mathsf{lock}\ \overline{v} : ![] \dashv \Delta, A_0, A_1}\ (\textsc{t:Lock-Rely})$$

When locking (line 5), the step of `F` is broken down into its two components: the rely type (Wait) and the guarantee type (Result; **none**). While Wait describes the linear resources that are now available to use, the guarantee type is an obligation to mutate the state to fulfill the given type before unlocking. Indeed, line 7 is only valid because the shared state was modified to match the promised guarantee type (Result).

$$\frac{\overline{\Gamma \mid \cdot \vdash v : \mathbf{ref}\ p \dashv \cdot} \qquad \mathbf{locs}(A_0) = \overline{p}}{\Gamma \mid \Delta,\ A_0, (A_0; A_1) \vdash \mathsf{unlock}\ \overline{v} : ![] \dashv \Delta, A_1}\ (\textsc{t:Unlock-Guarantee})$$

(with parenthesis used for clarity). Once the guarantee is fulfilled, we can move on to the next step of the protocol (in the case of `F`, **none**; or A_1, in the case of the rule above). The **none** type is the empty resource that can be automatically discarded, leaving Δ empty (·). Thus, the uses of protocols are mapped to the (t:Lock-Rely) and (t:Unlock-Guarantee) rules that step a protocol. We now show the rest of the encoding:

```
1 let newFork = λwork.                Γ = ·                  | Δ = work : ![] ⊸ int
2   let c = new Wait#{} in            Γ = c : ref l, l : loc | Δ = rw l Wait#![], ...
3     fork ... // lines 3 to 8 shown above.  Γ = ...         | Δ = M, F, ...
```

To simplify the presentation, our term language is stripped of type annotations. However, the newFork function has type !((![] ⊸ **int**) ⊸ (![] ⊸ **int**)) where the argument of this pure function is the `work` to be done by the thread, as was shown above. The resulting function is the join (shown below) that, once called, waits for the forked thread's result. Line 2 creates the cell that will be shared by the main and forked threads. This new cell, although typed ∃l.((!**ref** l) :: (**rw** l `Wait#![]`)), is automatically opened by the type system via (t:LocOpenBind) to allow direct access to the **ref** l reference via variable `c`.

Line 3 shares the cell by splitting the capability to location l into the `M` and `F` protocols. This split is done in a non-syntax-directed way through (T:Subsumption) (of Fig. 4), combined with the following rule for subtyping on Δ's:

$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad \Gamma \vdash A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \vdash \Delta_0, A_0 <: \Delta_1, A_1, A_2} \text{ (SD:Share)}$$

Where the following resource split ($\Rrightarrow$) is used:

$$\Gamma \vdash \mathsf{Wait} \Rrightarrow \texttt{M} \parallel \texttt{F} \qquad \text{(recall: } \mathsf{Wait} \triangleq \mathbf{rw}\ l\ \texttt{Wait\#![]}\text{)}$$

This split results in the capability to location l being replaced by the two protocols, `M` and `F`, in Δ. The composition check (described in the next subsection) relies on the knowledge that `M` and `F` share the same location. Once the protocols are known to compose safely, however, we no longer need to track this sharing—each protocol can abstract the location being accessed under a different name, and they can be used independently. The fork expression is typed by consuming the resources that the fork will use (such as `F` in the fork of line 3):

$$\frac{\Gamma \mid \Delta \vdash e : ![] \dashv \cdot}{\Gamma \mid \Delta \vdash \mathsf{fork}\ e : ![] \dashv \cdot} \text{ (T:Fork)}$$

This rule is somewhat similar to (T:Function), but the result type is unit because fork does not produce a result. Thus, a fork is executed for the effects it produces on the shared state. As such, to avoid leaking resources, the final residual resources of the forked expression must be empty and the resulting value pure (note that "$!A <: ![]$").

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 standard usage. The following text will focus on the less obvious details.

```
9  λ_.rec R.                                  Δ = ( Wait ⇒ (Wait; M) )  ⊕  ( Result ⇒ (Done; Done) )
10                                            [a]Δ = Wait ⇒ (Wait; M)        [b]Δ = Result ⇒ (Done; Done)
11    lock c;                                 [a]Δ = Wait, (Wait; M)         [b]Δ = Result, (Done; Done)
12    case !c of                              [a]Δ = rw l ![], (Wait; M)     [b]Δ = rw l ![],  (Done; Done)
13      Wait#x → // must restore linear value                                [a]Δ = rw l ![], (Wait; M)
14        c := Wait#x;                                                      [a]Δ = Wait, (Wait; M)
15        unlock c;                                                         [a]Δ = M
16        R // retries
17    | Result#x →                            [b]Γ = x : int, ...           | Δ = rw l ![], (Done; Done)
18        unlock c;                           [b]Γ = x : int, ...           | Δ = rw l ![]
19        delete c;                           [b]Γ = x : int, ...           | Δ = ·
20        x
21    end
22  end
23 end
```

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 via (T:ALTERNATIVE-LEFT):

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

The breakdown of $\oplus$ (line 10) is done automatically by the type system. Thus, the body of the recursion must be typed individually under each one of those alternatives, marked as `[a]` and `[b]`. The type of the resource on each alternative contains a sum type that matches different branches in the case of line 12. Note that it is safe for this sum type to only match a subset of the branches that the case lists. The remaining branches are simply ignored when typing the case with that sum type:

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

This enables the same case to produce different effects, such as obeying incompatible guarantees, based solely on the tagged contents of v. 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. Although line 19 deletes the cell, we first unlock the cell to fulfill the "Done; Done" guarantee of the final protocol step.

A rely-guarantee protocol is a specification of each lock-unlock usage, modeled by a protocol type. Therefore, we will continue the discussion on interference by only looking at the protocols, while omitting the actual concrete programs that use them.

2.1 Checking Safe Protocol Composition

We now introduce our main contribution: a novel axiomatic definition of protocol composition, which is later extended to support abstraction. Composing protocols over some shared state requires considering all possible ways in which the use of these protocols may be interleaved. Thus, regardless of the non-deterministic way by which aliases are interleaved at run-time, a correct composition will ensure that all possible uses are safe.

Intuitively, 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, we must build a co-inductive proof of safe interference. We only consider binary splits when checking composition but since a protocol can be later re-split, there is no limit to how many protocols may share some state.

The two protocols, `M` and `F`, shown above contain a finite number of different positions. 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:

$$\langle\ \Gamma \vdash \mathit{Resources} \Rrightarrow \mathit{Protocol} \parallel \mathit{Protocol}\ \rangle$$

Thus, when we split a Wait cell into protocols M[4] and F[5], we get the following set of configurations that simulate the uses done via the protocols (seen as atomic public transitions of lock-unlock uses, corresponding to the respective rely and guarantee types):

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

Configuration ❶ represents the initial split of Wait into M and F. Starting from some 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 the last step of M is ready to recover ownership, and ❹ where the ownership of the shared resource was recovered and all protocols have terminated (i.e. all resources are empty, **none**).

Upon sharing, the ownership of the shared resources belongs to all intervening protocols; all protocols can access the shared resources through locking. Ownership recovery means that this ownership is given back to one single protocol and "revoked" from all remaining protocols. In our protocols, recovery is modeled via protocol termination, such that a step transitions to a state rather than to another protocol step. 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 accidentally assume that that shared state is still available. The ownership recovery in ❸ transfers Done from the "pool" of shared resources to the alias that uses the last step of the M protocol. We also see by ❹ that this stepping consumes both the shared resource (leaving it as **none**) and the final "step" of M (leaving the protocol position also as **none**).

All protocol configurations shown above can take a step. (Even **none** can take a vacuous step that remains in the same configuration since **none** cannot change the shared resources.) 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 resources. For instance, $\langle \Gamma \vdash \mathsf{Result} \Rrightarrow \mathtt{M} \parallel \mathtt{F} \rangle$ cannot take a step with F since F does not rely on Result in any of its available 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.

Protocol composition ensures that a resource, R (capabilities or protocols), can be shared (split) as two protocols, P and Q, noted: $\Gamma \vdash R \Rrightarrow P \parallel Q$. Fig. 5 lists the grammatical categories (for protocols, states and resources) that we consider when composing protocols. As exemplified above we use a set of *configurations*, C, to represent the positions of each protocol as we traverse all possible interleaved uses of the two new protocols. C is defined as:

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

[4] $\mathtt{M} \triangleq (\ \mathsf{Wait} \Rightarrow (\ \mathsf{Wait}\ ;\ \mathtt{M}\ )\ ) \oplus (\ \mathsf{Result} \Rightarrow (\ \mathsf{Done}\ ;\ \mathsf{Done}\ )\ )$

[5] $\mathtt{F} \triangleq \mathsf{Wait} \Rightarrow (\ \mathsf{Result}\ ;\ \mathbf{none}\ )$

$$
\begin{aligned}
P, Q &::= (\textbf{rec}\ X(\overline{u}).P)[\overline{U_P}] \mid X[\overline{U_P}] \mid P \oplus P \mid P \,\&\, P \mid \textbf{none} \\
&\quad \mid S \Rightarrow P \mid S\ ;\ P \mid \exists l.P \mid \forall l.P \mid \exists X <: A.P \mid \forall X <: A.P \\
S &::= (\textbf{rec}\ X(\overline{u}).S)[\overline{U_S}] \mid X[\overline{U_S}] \mid S \oplus S \mid S \,\&\, S \mid \textbf{none} \mid A * A \mid \textbf{rw}\ p\ A \\
R &::= P \mid S
\end{aligned}
$$

Note: that the structure of allowed protocols is further restricted via protocol composition, beyond the syntactical categories above. Namely, abstraction is only enabled by the rules of Section 3.3.

Figure 5: Grammar restrictions for checking safe protocol composition: *P*rotocols, *S*tates, and *R*esources.

Protocol composition, applied via (SD:SHARE), ensures that all configurations reachable through stepping are themselves able to take a step, as follows:

$$
\frac{\langle\ \Gamma \vdash R \Rrightarrow P \parallel Q\ \rangle\uparrow}{\Gamma \vdash R \Rrightarrow P \parallel Q}\ (\text{WF:SPLIT}) \qquad \frac{C_0 \mapsto C_1 \qquad C_1\uparrow}{C_0\uparrow}\ (\text{WF:CONFIGURATION})
$$

Where $C\uparrow$ signals the *divergence* of stepping, consistent with the co-inductive nature of protocol composition. 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.

We now discuss the basic protocol composition definition of Fig. 6. (C:ALLSTEP) synchronously steps all existing configurations, where each configuration is stepped through (C:STEP). We use $\mathcal{R}_*$ (where $_*$ is either L or R) to specify the configuration reduction context on one of the protocols of a configuration, while the remaining protocol remains stationary, i.e.:

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

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 some state (C-SS:*), or is applicable on both kinds of resource (C-RS:*). (C-RS:NONE) "spins" a configuration since a terminated protocol cannot use the shared resources but must be stuck-free for consistency with our definition. The following (C-RS:*ALTERNATIVE) and (C-RS:*INTERSECTION) rules "dissect" a resource 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 resource to consider when stepping. State stepping, (C-SS:STEP), transitions the step of the protocol and changes the state of the shared resources to reflect the guaranteed state of the protocol. Ownership recovery, (C-SS:RECOVERY), "consumes" the shared state (leaving it as **none**) which models the transfer of ownership of that state back to the client context that uses the final step of the protocol. Protocol stepping, (C-PS:STEP), requires an exact simulation of the rely and guarantee types when stepping both the simulated protocol and the current stepping protocol.

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

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

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

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

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

$$\frac{\langle\ \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C}{\langle\ \Gamma \vdash R_0 \& R_1 \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C}\ \text{(C-RS:STATEINTERSECTION)}$$

$$\frac{\langle\ \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0]\ \rangle \mapsto C}{\langle\ \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0 \oplus P_1]\ \rangle \mapsto C}\ \text{(C-RS:PROTOCOLALTERNATIVE)}$$

$$\frac{\langle\ \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0]\ \rangle \mapsto C_0 \qquad \langle\ \Gamma \vdash R \Rrightarrow \mathcal{R}[P_1]\ \rangle \mapsto C_1}{\langle\ \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0 \& P_1]\ \rangle \mapsto C_0 \cdot C_1}\ \text{(C-RS:PROTOCOLINTERSECTION)}$$

$$\frac{\langle\ \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C_0 \qquad \langle\ \Gamma \vdash R_1 \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C_1}{\langle\ \Gamma \vdash R_0 \oplus R_1 \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C_0 \cdot C_1}\ \text{(C-RS:STATEALTERNATIVE)}$$

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

$$\frac{}{\langle\ \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[S_0 \Rightarrow S_1; P]\ \rangle \mapsto \langle\ \Gamma \vdash S_1 \Rrightarrow \mathcal{R}[P]\ \rangle}\ \text{(C-SS:STEP)}$$

$$\frac{}{\langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[S]\ \rangle \mapsto \langle\ \Gamma \vdash \textsf{none} \Rrightarrow \mathcal{R}[\textsf{none}]\ \rangle}\ \text{(C-SS:RECOVERY)}$$

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

$$\frac{}{\langle\ \Gamma \vdash S_0 \Rightarrow S_1; Q \Rrightarrow \mathcal{R}[S_0 \Rightarrow S_1; P]\ \rangle \mapsto \langle\ \Gamma \vdash Q \Rrightarrow \mathcal{R}[P]\ \rangle}\ \text{(C-PS:STEP)}$$

Figure 6: Basic protocol composition stepping rules.

Note that the rules above also enable the re-splitting of a protocol by extending an ownership recovery step. 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.

3 Polymorphic Protocol Composition

Up to this point, protocol composition does a strict stepping of protocols. Consequently, stepping requires each protocol to know the exact type representation of the shared resources. Ideally, to improve both locality and modularity, each protocol should only depend on the type information that is relevant to the actions done through that alias. For instance, the action done through the `F` protocol of page 9 does not need to know the precise type (Wait) that is initially stored in location l. Thus, we want to be able to abstract Wait as X such that the protocol only keeps the typing information that is relevant to that protocol's *local perspective* on the shared resources:

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

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

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

Since rely-guarantee protocols are first-class types, they can move outside the scope of a "module". Without this form of abstraction, such a move would either expose potentially private information or limit how clients may later re-split the shared resources. While enabling protocols to abstract part of their uses based on their perspective of the shared resources improves modularity and increases flexibility, it also brings new challenges on defining safe protocol composition and ensuring its termination. We will focus the discussion on two new aliasing idioms that this kind of abstraction enables: a) *existential-universal interaction*, how a universally quantified guarantee can safely 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) yet without the risk of introducing unsafe interference on older protocols of that state.

Section 4 approaches the decidability problem. The remaining of this section starts by introducing the basic intuition of how protocol-level abstraction works, before extending our axiomatic definition of composition to account for abstraction.

3.1 Existential-Universal Interaction

Enabling existential abstraction over the contents of the shared state will naturally allow a greater decoupling from the actions done by other aliases to that shared state. However, since a protocol encodes *sequences* of steps, ensuring safety must also account for the validity of the scope of the opaque type. For instance, consider the composition:

$$\Gamma\ \vdash\ \mathbf{rw}\ p\ \mathbf{int} \Rrightarrow \exists X.(\ \mathbf{rw}\ p\ X\ \Rightarrow\ \mathbf{rw}\ p\ X\ ;\ \mathbf{rw}\ p\ X\ \Rightarrow\ \ldots\ )\ \|\ (\ \mathbf{rw}\ p\ \mathbf{int}\ \Rightarrow\ \mathbf{rw}\ p\ \mathbf{boolean}\ ;\ \ldots\ )$$

On the left protocol, the assumption of X extends beyond a single step. Because the right protocol can change the underlying representation of X, this composition cannot be ruled safe. Indeed, if X were of a pure type, the left protocol could potentially reintroduce a type that would unsafely interfere with the right protocol's assumptions on the shared state. Thus, while the left protocol depends on an opaque type, it still requires that the "lifetime" of X extends to the next step.

We now discuss the core ideas that enable the safe composition of protocols that interact over abstractions. First the interaction will only occur via the "lifetime" of the stored type (as it changes on each step), and then we will use bounded quantification to enable types that are less opaque. Consider the following protocols that are sharing a 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, while also guaranteeing that X is restored before repeating the protocol. Thus, `Nothing` cannot publicly modify the shared state, although p 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 $\forall$ of 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. However, the "lifetime" of X in `Nothing` is restricted to a single step. Naturally, to be able to check this composition in a finite number of steps, we must check 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`**:

$$p : \mathbf{loc} \ \vdash\ \mathbf{rw}\ p\ \mathbf{int}\ \Rrightarrow\ \texttt{Full}[\mathbf{int}]\ \|\ \texttt{Nothing}$$

Protocol composition results in the following set of configurations:

$$\begin{array}{l} \{\ ❶\ \langle\ p : \mathbf{loc} \vdash \mathbf{rw}\ p\ \mathbf{int} \Rrightarrow \texttt{Full}[\mathbf{int}]\ \|\ \texttt{Nothing}\ \rangle\ , \\ \ \ ❷\ \langle\ p : \mathbf{loc}, Z : \mathbf{type} \vdash \mathbf{rw}\ p\ Z \Rrightarrow \texttt{Full}[Z]\ \|\ \texttt{Nothing}\ \rangle\ \} \end{array}$$

The use of abstraction will mean that each configuration may have different assumptions of type (and location) variables. 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` (**rw** p X). 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. 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. In this case, we used Z to represent that new type. It is straightforward 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. Perhaps 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 proof of safe composition. Although each step of `Full` must consider a fresh type due to the $\forall$, 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`'s guarantee is infinite, the number of *distinct interactions* that can legally occur through that shared state is finite if we model those interactions abstractly. Lifetime conflicts cannot occur with this technique as even if we open an existential, we must still step the new configuration. Consequently, the problematic composition above would be detected via stepping.

We can use bounded quantification to provide more expressive abstractions that go beyond the fully opaque types used above (which are equivalent to a "<: **top**" bound), and convert this example into one of more practical use. 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.

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

As before `W` is committed to preserve the representation type of X although it now has sufficient room to use that type as `B`. `C` is now more constrained than before since it is forced to guarantee a type that is compatible with `B`. 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 similar set of configurations, but with the bound assumption on Z.

This form of asymmetric interaction over shared state relates to the `full` – `pure` interaction of *access permissions* [4]. A `full` permission allows exclusive write permission to an object, but also enables read-only permissions (`pure`) to co-exists. Consequently, each `pure` permission must assume that other permissions can modify the shared object up to a certain type, the *state guarantee*. While their work focuses on the read-write distinction, and our work is centered on modeling type-changing mutations (so all aliases can write), the example shows that we are able to naturally model similar asymmetric interaction within our protocol framework.

3.2 Inner Step Extension with Specialization

Re-splitting an existing protocol while specializing its interference 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. The latter case is more interesting: when combined with abstraction it allows specialization *within* an existing step (i.e. nested re-splits), enabling new forms of shared state interaction through that abstraction.

To illustrate the expressiveness gains, we revisit the join protocol of Section 2. 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 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.

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

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 2 is a specialization of this protocol since it includes only one of the choices. In here, we 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 together and flag the state as ready, so that `Main` can proceed.

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

It is important to note that the new tags/values are nested *inside* the old `W` tag. This ensures that the new usages remain hidden from `M` and "look" just like the previous `F` usage. (There are also no lifetime conflicts since `M` does not preserve its type assumption on the abstraction beyond a single step.) 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 already finished. We can then re-split `F` as follows (note the required initial type in `F`, `E#[]`, for this split to be valid):

$$\Gamma\ \vdash\ \texttt{F}[\texttt{E\#}[]]\ \Rrightarrow\ \texttt{K}\ \|\ \texttt{K}$$

Protocol composition follows analogous principles to above, except that we are now simulating the steps of the original `F` protocol with the steps of the two new `K` protocols:

$$\{\ \langle\ \Gamma \vdash\ \texttt{F}[\texttt{E\#}[]]\ \Rightarrow\ \texttt{K}\ \|\ \texttt{K}\ \rangle,\ \langle\ \Gamma \vdash \texttt{F}[\texttt{R\#}\textbf{int}] \Rightarrow\ \texttt{K}\ \|\ \textbf{none}\ \rangle,$$
$$\langle\ \Gamma \vdash \texttt{F}[\texttt{R\#}\textbf{int}] \Rightarrow \textbf{none}\ \|\ \texttt{K}\ \rangle,\ \langle\ \Gamma \vdash\ \textbf{none} \Rightarrow \textbf{none}\ \|\ \textbf{none}\ \rangle\ \}$$

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

$$\text{(c-rs:Weakening)}\quad \frac{\langle\, \Gamma_0 \vdash R \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, \Gamma_0, \Gamma_1 \vdash R \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C} \qquad \text{(c-ss:ForallLoc)}\quad \frac{\langle\, \Gamma, l : \mathbf{loc} \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow \forall l.P] \,\rangle \mapsto C}$$

$$\text{(c-ss:OpenLoc)}\quad \frac{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P\{p/l\}] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[\exists l.P] \,\rangle \mapsto C} \qquad \text{(c-ss:ForallType)}\quad \frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow \forall X <: A.P] \,\rangle \mapsto C}$$

$$\text{(c-ss:OpenType)}\quad \frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P\{A_1/X\}] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[\exists X <: A_0.P] \,\rangle \mapsto C}$$

$$\text{(c-ps:ExistsType)}\quad \frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash P \Rrightarrow \mathcal{R}[Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash \exists X <: A.P \Rrightarrow \mathcal{R}[\exists X <: A.Q] \,\rangle \mapsto C} \qquad \text{(c-ps:ExistsLoc)}\quad \frac{\langle\, \Gamma, l : \mathbf{loc} \vdash P \Rrightarrow \mathcal{R}[Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash \exists l.P \Rrightarrow \mathcal{R}[\exists l.Q] \,\rangle \mapsto C}$$

$$\text{(c-ps:ForallType)}\quad \frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash S \Rightarrow P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall X <: A.P \Rrightarrow \mathcal{R}[S \Rightarrow \forall X <: A.Q] \,\rangle \mapsto C}$$

$$\text{(c-ps:ForallLoc)}\quad \frac{\langle\, \Gamma, l : \mathbf{loc} \vdash S \Rightarrow P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[S \Rightarrow \forall l.Q] \,\rangle \mapsto C} \qquad \text{(c-ps:LocApp)}\quad \frac{\langle\, \Gamma \vdash S \Rightarrow P\{p/l\} \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}$$

$$\text{(c-ps:TypeApp)}\quad \frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\, \Gamma \vdash S \Rightarrow P\{A_1/X\} \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall X <: A_0.P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}$$

$P\{A/X\} \triangleq$ "substitution, in P, of X for A"

Note: bound type/location variables of a type must be fresh in that rule's conclusion.

Figure 7: Protocol composition abstraction extension.

when given a choice (&) of F steps. Similarly, at least one alternative (⊕) 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 uses of the shared state.

3.3 Composing Abstract Protocols

The composition rules of Fig. 29 complement those of Fig. 6 to enable composing abstract protocols. Weakening on a configuration (up to renaming), (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 (c-ss:Forall*) rules do similar stepping to (c-ss:Step) but considering an abstracted guarantee, which results in a typing environment with the opened abstraction. (c-ss:Open*) ex-

poses the representation type/location (if it exists) before doing a regular step. (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, enabling step specialization during simulation. In the T.R, we also consider a straightforward extension to protocol composition that enables subtyping over stepping.

3.4 Discussion & Brief Examples

Above, we showed how our local, isolated protocol types can model core interference concepts over a relatively small and simple calculus. We refrained from adding support for more precise states and refined data abstractions of others (such as [18]), and focus instead on typestates [22, 33, 34]. However, this is not an intrinsic limitation of our model. If we consider more precise states, we can (for instance) model monotonic counters from prior work [28, 14] where each counter shares state symmetrically. Our local protocols model these uses solely from the perspective of a single alias as:

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

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 the core existential-universal protocol interaction discussed above (but, in our calculus, using less precise types: J and I) and where the protocol can be re-split indefinitely.

While our states are less precise, we can enforce more precise uses of that shared state. The semantics of prior work [28, 14] differed on whether the counter was forcefully used by clients, or whether the action was simply available to be used. We can model the two cases explicitly:

$$\exists p.(\ (!\textbf{ref}\ p) :: \texttt{MC}\ \multimap\ [] :: \texttt{MC}\ )$$

Enables clients to use the counter an arbitrary number of times or simply thread it through, unused.

$$\forall X.\exists p.(\ (!\textbf{ref}\ p) :: \exists J.(\ \textbf{rw}\ p\ J\ \Rightarrow \forall I.(\ \textbf{rw}\ p\ I\ ;\ X\ )\ )\ \multimap\ [] :: X\ )$$

By unfolding the protocol, the function guarantees that a single step of the protocol will be used. Since we (intentionally) abstract subsequent steps, the function cannot use the counter beyond that single use. Analogous reasoning can be used to enforce specific, finite, usages.

Adding support for dependent refinement types, and ensuring its decidability (even without interference), is beyond the scope of our work as we focus on the core composition problem. However, we believe that the underlying decidability insights made here will carry to a system with decidable dependent refinement types; even if perhaps requiring more fine-grained conditions to close the co-inductive proof of safe interference—that are only relevant once more precise typing is considered.

While we use a relatively simple calculus to keep the theory focused on the core of interference-control, we can for instance model `MVar`s [25]. Fig. 8 shows an `MVar`, a structure that contains

```
let newMVar = λ _.
  let m = new Empty#{} in
    // "shares" the new cell using MVar protocol
    Γ ⊢ (rw l Empty#[]) ⇛ MVar[l] ∥ none
    {
    putMVar = λ val.
      rec R.
        lock m;
        case !m of
          Empty#x → m := Full#val;
                    unlock m
        | Full#value → m := Full#value;
                       unlock m;
                       R // retries
        end
      end,
   splitMVar = λ _.
    Γ ⊢ MVar[l] ⇛ MVar[l] ∥ MVar[l]
    {},
   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
```

Figure 8: MVar example.

```
let x = new 0 in
 // share 'x' as via some protocols
 {
   lockMe = λ_.lock x,
   // ...
 }
```

Figure 9: Indirect locking.

a single shared cell which is either empty or contains a value of some type. Notable operations include: `putMVar`, that waits until the cell is empty before inserting the given value; and `takeMVar` which waits until the cell is full to remove the cell's value, leaving the cell empty. `MVar`s can be shared by many aliases, each assigned a protocol such as:

$$\begin{aligned}\texttt{MVar}[m] \triangleq\ & \exists Y.(\ (\ (\mathbf{rw}\ m\ \texttt{Empty\#}Y) \Rightarrow (\mathbf{rw}\ m\ \texttt{Empty\#}Y)\ ;\ \texttt{MVar}[m]\ )\ \&\\ & \qquad (\ (\mathbf{rw}\ m\ \texttt{Empty\#}Y) \Rightarrow (\mathbf{rw}\ m\ \texttt{Full\#}\mathbf{int})\ ;\ \texttt{MVar}[m]\ )\ )\\ \oplus\ & (\ (\ (\mathbf{rw}\ m\ \texttt{Full\#}\mathbf{int}) \Rightarrow (\mathbf{rw}\ m\ \texttt{Empty\#}[])\ ;\ \texttt{MVar}[m]\ )\ \&\\ & \qquad \exists Y.(\ (\mathbf{rw}\ m\ \texttt{Full\#}Y) \Rightarrow (\mathbf{rw}\ m\ \texttt{Full\#}Y)\ ;\ \texttt{MVar}[m]\ )\ )\end{aligned}$$

The appendix includes additional examples, including modeling examples of prior work with our more local protocol types. We can also model a shared pair where each alias keeps its own, local, precise knowledge on one of the two components of the pair stored in that shared state. The two aliases, `L` and `R`, share a common cell but keep part of that state private to itself. While both can do private actions over the shared cell, they are guaranteed to not interfere with the precise assumptions of the remaining alias.

$$\begin{aligned}\texttt{P}[A][B] &\triangleq \mathbf{rw}\ p\ [A,B]\\ \texttt{L}[A] &\triangleq \exists X.(\ \texttt{P}[A][X] \Rightarrow \forall Y.(\ \texttt{P}[Y][X]\ ;\ \texttt{L}[Y]\ )\ )\\ \texttt{R}[A] &\triangleq \exists X.(\ \texttt{P}[X][A] \Rightarrow \forall Y.(\ \texttt{P}[X][Y]\ ;\ \texttt{R}[Y]\ )\ )\end{aligned} \qquad \Gamma \vdash \texttt{P}[X][Y] \Rrightarrow \texttt{L}[X] \parallel \texttt{R}[Y]$$

Thus, we can use the different perspectives of each protocol to model local knowledge that is hidden from other aliases, within our core protocol framework without needing additional mechanisms.

Since our types express sharing, we can use standard techniques to abstract the components of a protocol type after safe composition is checked. This enables an abstraction to expose a type interface that indirectly manipulates the shared state, such as indirectly locking/unlocking state (Fig. 9). We can type the record in such a way to hide the type in `x` but still expose some information on sharing that is useful for later enabling other typestate functions [22]. For instance:

$$\exists \mathsf{A}.\exists \mathsf{B}.\exists \mathsf{C}.[\ \ldots,\ \texttt{lockMe} : [] :: (\mathsf{A} \Rightarrow \mathsf{B};\mathsf{C}) \multimap [] :: (\mathsf{A} * (\mathsf{B};\mathsf{C})),\ \texttt{add} : (\ \mathbf{int} :: \mathsf{A} \multimap [] :: \mathsf{A}\ ),\ \ldots\ ]$$

Clients can only call `add` once the type `A` is available. This could model, for instance, a global lock on a collection to enable more coarse-grained control over the interference to that collection—but without exposing the lock to clients. Thus, when `lockMe` returns, the client receives a type that

```
lock @a;  Γ = a : ref @a
  let b = !a;
  lock @b; // locks loc. of 'b'
unlock @a;
    let c = !b;
    lock @c;
  unlock @b;
    ...
```

Figure 10: Hand-over-hand locking example.

expresses that A is available and that a guarantee (B; C) is expected to be fulfilled. However, this fulfillment can only occur indirectly via the wrapper record as clients do not have a direct way of accessing or mutating the internals of that shared state.

While we do not guarantee dead-lock freedom, it is possible to type more fine-grained locking schemes such as *hand-over-hand* locking (Fig. 10). Consider the protocol of a list's node:

$$\mathsf{L}[q] \triangleq \exists l.(\ (\mathbf{rw}\ q\ !\mathbf{ref}\ l) * \mathsf{L}[l] \ \Rightarrow\ (\mathbf{rw}\ q\ !\mathbf{ref}\ l)\ ;\ \ldots\ )$$

L is defined over a location q that contains the (abstracted) reference to the next element of the sequence of *l*ocations to be locked. Locking will enable access to that **ref** l which can then be locked to gain access to $\mathsf{L}[l]$, the next element in the sequence of locations to lock. For brevity, we make each step simply consume the L protocol of the element in that sequence, instead of (for instance) re-splitting.

4 Composition Decidability & Other Technical Results

We now show decidability of protocol composition and discuss the remaining technical results of our language. The decidability statement comes as a direct consequence of ensuring a regular type structure via syntactic well-formedness constraints on recursive types. Although applied in the context of protocol composition, we follow ideas from prior work on ensuring decidable subtyping over bounded quantification [32, 6]. The main novelty is in extending this kind of reasoning to account for recursive types with parameters, in order to ensure a regular type structure over our more flexible recursive types. To achieve this, we apply well-formedness conditions which ensure that there is only a finite number of reachable (abstract) protocol states. We focus the discussion on decidability of protocol composition, and point interested readers to appendix where these conditions are properly motivated and discussed. Crucially, these well-formedness conditions enable us to state the following:

Lemma 1 (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.*

Lemma 2 (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* ($\equiv$) *types.*

Lemma 3 (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.

4.1 Composition Properties, Algorithm, and Decidability

Informally, correctness of protocol composition is based on the two 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 but leave the proofs to the appendix. The following two lemmas show stuck freedom by properties that resemble progress and preservation but over protocols:

Lemma 4. *If* $\Gamma \vdash R \Rrightarrow P \parallel Q$ *then* $\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle \mapsto C$.

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

Lemma 5. *If* $\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash R' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C$ *and* $\Gamma \vdash R \Rrightarrow P \parallel Q$ *then* $\Gamma' \vdash R' \Rrightarrow P' \parallel Q'$.

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

Note that 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 $\Gamma \vdash R \Rrightarrow P \parallel Q$ is defined there is a single such R (defined up to subtyping and equivalent protocol/state interference specification). Since for any two protocols there may not always exist an R that can be split into P and Q, this is a partial monoid.

Lemma 6. *Protocol composition obeys the following properties:*

1. *(identity)* $\Gamma \vdash R \Rrightarrow R \parallel$ ***none***.
2. *(commutativity) If* $\Gamma \vdash R \Rrightarrow P_0 \parallel P_1$ *then* $\Gamma \vdash R \Rrightarrow P_1 \parallel P_0$.
3. *(associativity) If we have* $\Gamma \vdash R \Rrightarrow P_0 \parallel P$ *and* $\Gamma \vdash P \Rrightarrow P_1 \parallel P_2$ *then exists* Q *such that* $\Gamma \vdash R \Rrightarrow Q \parallel P_2$ *and* $\Gamma \vdash Q \Rrightarrow P_0 \parallel P_1$.

   (i.e. If $\Gamma \vdash R \Rrightarrow P_0 \parallel (P_1 \parallel P_2)$ *then* $\Gamma \vdash R \Rrightarrow (P_0 \parallel P_1) \parallel P_2$ *)*

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 (as we target sharing when that tracking is not possible), merging cannot "collapse" a protocol into a non-protocol type. In this case "merging" is equivalent to simply having the two non-merged protocols available in Δ or bundled using the $*$ type.

The composition algorithm is shown in the appendix and is a straightforward implementation of the axiomatic definitions shown above. The algorithm uses a set of *v*isited configurations to remember past configurations and ensure that once all different protocol configurations are exhausted (up to renaming and weakening of Γ), the algorithm can terminate. We now state our technical lemmas on the composition algorithm but leave the proofs to the appendix.

Lemma 7. *Given well-formed types and environment, we have that:*

1. *(soundness) if* $\mathsf{c}(\Gamma,\ R,\ P,\ Q)$ *then* $\Gamma \vdash R \Rrightarrow P \parallel Q$.
2. *(completeness) if* $\Gamma \vdash R \Rrightarrow P \parallel Q$ *then* $\mathsf{c}(\Gamma,\ R,\ P,\ Q)$.
3. *(decidability)* $\mathsf{c}(\Gamma,\ R,\ P,\ Q)$ *terminates.*

4.2 Correctness Properties

The 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 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). For 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 1. *If* $\Gamma \mid \Delta \vdash T_0$ *and* $\mathtt{live}(T_0)$ *and if exists* H_0 *such that* $\Gamma \mid \Delta \vdash H_0$ *then* $H_0\ ;\ T_0 \ \mapsto\ H_1\ ;\ T_1$.

$\mathtt{live}(T)$ means that the thread pool T contains at least one "live" thread such that the thread is neither a value nor is waiting for a lock to be released (which includes deadlocks). $\Gamma \mid \Delta \vdash H$ ensures that the *H*eap is well-defined according to Γ and Δ.

We define $\mathtt{Wait}(H, e)$ over a thread e and heap H such that the $\mathcal{E}$valuation context is reduced to evaluating the configuration: $H\ ;\ \mathcal{E}[\mathsf{lock}\ \rho, \overline{\rho'}]\ \cdot\ T$ where $\rho \hookrightarrow v \notin H$ which contains at least one location (ρ) that is currently locked or was deleted and, therefore, the thread must block waiting (potentially indefinitely) for that lock to be available before continuing. "Early" deletion of shared resources results in a pending guarantee. Since well-formed threads cannot leave residual resources, this situation is ruled out for correct programs, but may occur on the theorem below.

Progress over expressions is defined as follows:

Theorem 2. *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)* $\texttt{Wait}(H_0, e_0)$*.*

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 3. *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 pool 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 4. *If we have* $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \cdot T$ *and* $\Gamma_0 \mid \Delta_0, \Delta_T, \Delta_2 \vdash H_0$ *and* $\Gamma_0 \mid \Delta_0, \Delta_T \vdash e_0 : A \dashv \Delta$ *then, 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 to show 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.

Corollary 1. *The following program states cannot be typed:*

1. *(Data Race) Simultaneous read/modify by two thread over the same location (we also ensure read-exclusive accesses):*

$$H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[!\rho] \cdot T \qquad H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[\rho := v'] \cdot T$$

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:*

$$H; \mathcal{E}[\mathsf{delete}\, \rho] \cdot T \qquad (\text{where } \rho \notin H)$$

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. Thus, these errors are ruled out by either protocol composition or by the resource tracking of the core linear system.

```
receive(c) ≜
  rec R.
    lock c;
    case !c of
      // 1. waiting states (A..Z)
      A#n → ... // analogous to below
    | Z#n → // restore linear content
        c := Z#n;
        unlock c;
        R // retry
      // 2. desired (receive) state
    | ReadyToReceive#v →
        c := Idle#{}; // "received"
        unlock c;
        v // value received from "channel"
    end
  end
```

```
send(c,v) ≜
  rec R.
    lock c;
    case !c of
      // 1. waiting states (A..Z)
      A#n → ... // analogous to below
    | Z#n → // restore linear content
        c := Z#n;
        unlock c;
        R // retry
      // 2. desired (idle) state
    | Idle#_ →
        c := ReadyToReceive#v; // "sent"
        unlock c;
        {} // result of send is empty
    end
  end
```

Figure 11: Simple encoding of `send` and `receive` functions via a shared cell.

5 Protocol Expressiveness

We show the expressiveness of our protocols by modeling typeful message-passing concurrency, using a straightforward encoding of message-passing via shared memory interference (Fig. 13). The encoding itself should be unsurprising as it follows well-known ideas from the literature, so we defer less important details to the appendix to focus instead on the most interesting aspect of this example: how our protocol framework is able to type such uses and ensure their safety.

We encode a more primitive, "low-level" view of typeful message-passing concurrency via the causality of shared memory interference. We focus on the non-distributed setting where a channel can be precisely encoded as a low-level shared cell. Channel communication and its changing session properties are emulated indirectly via inspection of or interference over the contents of that shared cell. Thus, our functions to send/receive a value simply hides the underlying Waiting states that may be needed when the cell is not available. A receiving step can be modeled by a protocol of the form:

$$\mathsf{Wait}[A..Z] \;\oplus\; (\ \mathbf{rw}\ \mathtt{c}\ \mathtt{ReadyToReceive\#}V \Rightarrow \mathbf{rw}\ \mathtt{c}\ \mathtt{Idle\#}[]\ ;\ \mathtt{NextStep}\ )$$

where Wait is a sequence of retry steps that leave the state unmodified, until a value of type V is "received". Sending uses a similar protocol but where we must wait for an `Idle` cell before "sending".

The appendix includes the complete "`Buyer-Seller-Shipper`" example (the canonical and simple example used in session-based concurrency works) while in here we only take a look at the main aspects of the `Buyer`'s interaction with the channel (Fig. 18).

```
let c = connectSeller() in
   c : buy!(prod) ; price?(p) ; details?(d)
   send(c, GET_USER_PRODUCT() );
   let price = receive(c) in
      let details = receive(c) in
         close(c)
      end
   end
end
```

Figure 12: Buyer code.

We model a *c*hannel using a capability to location c. For brevity, we omit "**rw** c" from "**rw** c A" since all changes occur over that same location. The `Buyer`'s type uses standard π-calculus [23] notations where ! sends and ? receives a value. In our protocols, these actions are mapped to the rely type (receive) and the guarantee type (send).

$$\underbrace{\texttt{buy!}(prod)}_{\texttt{idle0\#[]} \Rightarrow \texttt{buy\#}prod} \;;\; \underbrace{\texttt{price?}(p)}_{\texttt{price\#}p \Rightarrow \texttt{idle2\#[]}} \;;\; \underbrace{\texttt{details?}(d)}_{\texttt{details\#}d \Rightarrow \texttt{[]}}$$

`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 Wait as they simply recur on that same step of the protocol (i.e. "busy-wait").

$$\texttt{idle0\#[]} \Rightarrow \texttt{buy\#}prod \;;\; \textsf{Wait} \oplus (\,\texttt{price\#}p \Rightarrow \texttt{idle2\#[]}\,) \;;\; \textsf{Wait} \oplus (\,\texttt{details\#}d \Rightarrow \texttt{[]}\,)$$

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 flexibility comes at the cost of requiring a more complex composition mechanism. Protocol composition accounts for both non-deterministic protocol interleaving and "multi-way" communication, features which are usually absent from strictly choreographed session-based concurrency (favoring instead strong liveness properties over more deterministic, linear compositions).

Naturally, more complex examples are possible. In here our focus is on showing the core insights that enable us to relate the two techniques: 1) mapping receive/send 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. (Recall that we do not guarantee deadlock freedom, nor termination.)

6 Related Work

Our work relates to prior work on *rely-guarantee protocols* [21]. We show that these protocols are useful to reason about concurrency and significantly improve the flexibility of protocol composition. Namely, we allow the composition of abstract protocols (enabling more local typing as in Section 3), show that our composition is decidable, and provide a novel axiomatic definition of composition that is straightforward to implement. Since thread-based interference is rooted in alias-related interference, the technique itself is mostly indifferent to whether sharing occurs in the sequential or concurrent setting. Still, we address all technicalities that make concurrency possible (such as adding support to arbitrarily many threads, locking locations, well-formedness changes to the rely/guarantee types to ensure matching lock/unlock of locations, etc). Furthermore, by considering the concurrent setting we are able to express and relate rely-guarantee protocols to typeful message-passing concurrency.

Our work is also related to recent work on more precise tracking of interference. *Chalice* [20] uses a simplified form of rely-guarantee to reason about shared state interference by constraining a thread's changes to a two-state invariant, relating the previous and current states. Monotonic [10, 28] uses of shared state (where all changes converge to more precise states) are less dependent on aliasing information, which simplifies checking at the expense of expressiveness. Dynamic ownership recovery mechanisms [37, 29] choose some run-time overhead and dynamic safety guarantees to enable more flexible ownership recovery than purely static approaches. *Rely-guarantee references* [14] adapt the use of rely-guarantee to individual reference cells with support for dependent refinement types in a sequential language. Although the use of refinements adds expressiveness to the description of sharing, they do not support ownership recovery, nor address decidability, and typechecking can require manual assistance in Coq. *Access permissions* [37, 4, 3] control alias interference by categorizing read-write uses into different permission kinds. 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. However, this simpler form of read-only cannot capture their multiple, simultaneous readers semantics. Still, by modeling interference in a more fundamental way, we gain additional expressiveness beyond their most permissive `share` permission as we can model uses beyond invariant-based sharing. In [7] Crafa and Pavodani introduce a high-level (actor-like) model for sharing (type)state via join patterns. We target a more low-level programming paradigm (which builds typestates through type abstraction rather than as a first-class language feature), enabling us to introduce abstraction at the level of protocols and support protocol re-splitting in ways that are not expressible in their work.

Several recent works use *partial commutative monoids* [9, 19, 8] to model sharing by leveraging the concept of fictional separation [9, 15]. Commutative monoids offer the underlying general

principle for splitting resources, enabling seemingly unrelated components to interact via aliasing under a layer of (fictional) separation. We compare more closely to [19] due to our common use of $\mathbf{L}^3$ [1] and type-based approach. In [19], Krishnaswami *et al.* define a generic sharing rule based on programmer-supplied commutative monoids for safe sharing of state in a single-threaded environment. Their work does not approach the issue of decidability of resource splitting, and requires wrapping access to shared state in an module abstraction that serves as an intermediary to access shared state. Our work focuses on a custom commutative monoid that enables first-class sharing without (necessarily) needing a wrapping module abstraction. Although our protocol splitting is a specialized monoid, we showed that this mechanism is relatively flexible, decidable, and give an algorithmic implementation. Other technical differences between our works abound such as their use of affine refinement types (enabling more fine-grained types), our use of multi-threaded semantics and allowing inconsistent states (i.e. locked cells) to be moved around as first-class, etc.

Protocol-based mechanisms for safe interference are also used by other approaches, such as in program logic-based systems (e.g. [17, 35, 36, 24, 11]). By generally targeting manual proofs (and somewhat more involved specifications) these works generally fit into a different design space than ours, although share some interesting similarities. While we make concessions on expressiveness to achieve decidable protocol composition and re-splitting, these works focus instead on the expressiveness of their concurrency specification. LRG [11] supports lock-free structures but requires a special frame-rule to support framing over rely-guarantee conditions. We simply integrate protocols into the language (as linear resources) meaning that the standard frame-rule suffices. Supporting lock-free concurrency in our system would require reinterpreting a $\Rightarrow$ step as a single-cell, `atomic`, conditional operation; with the shared resource (stored in the cell) being immediately extracted/inserted from/into the cell, rather than just accessible after locking. CaReSL [36] and Iris [17] support "islands"/regions of memory that are shared together and whose imprecise state must be considered when using. Our composition rules enforce that a protocol carries all information on imprecise states, which is then deconstructed via (T:Alternative-Left) and case analysis. Our protocols can group shared state using the $*$ operator to define shallow "regions", while their works allow for far more rich specifications of atomic regions of any depth. Iris [17] further supports a form of re-splitting via a "view shifting" mechanism, to repartition (or create) shared regions. FCSL [24] encodes protocols via auxiliary/ghost state. Although done in a compositional way, it can require checking for safe interference ("stability") after a split since a safe split does not necessarily imply safe interference in all situations. Our protocol composition mechanism is essentially a form of checking for safe interference early, at the moment a protocol is split, by checking that all possible future uses are safe resembling a form of "pre-computed" stability check.

Protocol composition itself can also be seen as a form of model checking (to check that each state has a successor) that uses abstract states to ensure a finite state space, but in a system that is more intimately integrated with the language. Our protocols are first-class resources that can be specialized by clients, even abstracting (leaving out) later steps. Thus, our protocols guide the programmer on how to reason locally about (safe) interference by mapping its uses of locks to a local protocol type that models the alias perspective on the shared state. While our work focuses on modeling the core interference phenomenon within a small calculus, rather than precisely typing

existing programs, we still showed that extensions may be used to model at least some existing programs within our model.

7 Conclusions

We defined a flexible and decidable procedure that ensures the safe composition of interfering abstract protocols that share access to mutable state. While employing a relatively small protocol framework, we are able to model the core interference principles of complex shared state interactions within our core calculus. Finally, we showed the expressiveness of our protocol framework by discussing how it can capture in a unified way both shared memory interference and typeful message-passing concurrency.

References

[1] A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, Dec. 2007.

[2] R. M. Amadio and L. Cardelli. Subtyping recursive types. In *POPL 1991*.

[3] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA 2008*.

[4] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA 2007*.

[5] L. Caires and H. T. Vieira. Conversation types. In *ESOP 2009*, 2009.

[6] G. Castagna and B. C. Pierce. Decidable bounded quantification. In *POPL 1994*.

[7] S. Crafa and L. Padovani. The chemical approach to typestate-oriented programming. In *OOPSLA 2015*.

[8] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL 2013*.

[9] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010*.

[10] M. Fähndrich and K. R. M. Leino. Heap monotonic typestate. In *IWACO 2003*.

[11] X. Feng. Local rely-guarantee reasoning. In *POPL '09*.

[12] T. Freeman and F. Pfenning. Refinement types for ml. In *PLDI '91*, pages 268–277, New York, NY, USA, 1991. ACM.

[13] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[14] C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *PLDI 2013*.

[15] J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP 2012*.

[16] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 1983*.

[17] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL 2015*.

[18] N. R. Krishnaswami, P. Pradic, and N. Benton. Integrating linear and dependent types. In *POPL 2015*.

[19] N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP 2012*.

[20] K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP 2009*.

[21] F. Militão, J. Aldrich, and L. Caires. Rely-guarantee protocols. In *ECOOP 2014*.

[22] F. Militão, J. Aldrich, and L. Caires. Substructural typestates. In *PLPV 2014*.

[23] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, Sept. 1992.

[24] A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP 2014*.

[25] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL 1996*.

[26] B. C. Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 305–315, New York, NY, USA, 1992. ACM.

[27] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[28] A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI 2011*.

[29] F. Pottier and J. Protzenko. Programming with permissions in mezzo. In *ICFP 2013*.

[30] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Logic in Computer Science*, pages 55–74, 2002.

[31] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proc. LISP and Functional Programming*, 1992.

[32] J. Seco and L. Caires. Subtyping first-class polymorphic components. In *ESOP 2005*.

[33] R. E. Strom. Mechanisms for compile-time enforcement of security. In *POPL 1983*.

[34] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[35] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP 2014*.

[36] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP '13*.

[37] R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP 2011*.

A Ensuring Regular Type Structure

The decidability statement comes as a direct consequence of ensuring a regular type structure. Although applied in the context of protocol composition, we follow ideas from prior work on ensuring decidable subtyping over bounded quantification [32, 6]. The main novelty is in extending this kind of reasoning to account for recursive types with parameters, in order to ensure a regular type structure over our more expressive unfolding. To achieve this, we apply well-formedness conditions to our recursive types. We favor simpler (as in more "natural") constraints that make our decidability argument clearer, rather than try to achieve more permissive conditions by considering lots of special-cases. We leave to compelte set of well-formedness rules to Appendix G.1.

To illustrate an irregular unfold, consider the following recursive type, R:

$$(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))[\texttt{int}] \tag{Ex.1}$$

Unfolding this type, and its resulting sub-terms, produces the following sequence of types (for clarity, we underline the "fixed" part of the recursive type as it is unfolded):

$$\begin{aligned}
&\underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [\texttt{int}] \\
\texttt{int} \multimap\ &\underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [\texttt{int} \multimap \texttt{int}] \\
\texttt{int} \multimap \texttt{int} \multimap\ &\underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [(\texttt{int} \multimap \texttt{int}) \multimap (\texttt{int} \multimap \texttt{int})] \\
&\quad\ldots
\end{aligned}$$

We see that the type that results from unfolding is not regular, as the use of the recursive variable R in "**rec** $R(X).(\ \texttt{int} \multimap R[X \multimap X]\ )$" 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). 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:

$$\frac{(X : k_0 \to \ldots \to k_n \to \textbf{type}) \in \Gamma \qquad (U_i : k_i) \in \Gamma \qquad i \in \{0, \ldots, n\}}{\Gamma \vdash X[\overline{U}]\ \textbf{type}}$$

The rule states that for a given recursive type variable X (recursive type variables have a "$\ldots \to$ **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 "$\ldots \to$ **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:

$$\frac{u_0 : k_0, ..., u_n : k_n, X : k_0 \to ... \to k_n \to \textbf{type} \vdash A\ \textbf{type} \qquad \Gamma \vdash U_i\ k_i \qquad k_i = \texttt{kind}(u_i) \qquad i \in \{0, ..., n\}}{\Gamma \vdash (\textbf{rec}\ X(\overline{u}).A)[\overline{U}]\ \textbf{type}}$$

where: $\texttt{kind}(l) = \textbf{loc}$ and $\texttt{kind}(X) = \textbf{type}$. Thus, using $\texttt{int}$ as argument in $(\textbf{rec}\ R(X).A)[\texttt{int}]$ 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:

$$(\textbf{rec}\ V(Z).(\forall X <: (A \multimap Z).V[X]))[\texttt{int}] \tag{Ex.2}$$

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):

$$\begin{aligned} \cdot &\vdash V[\texttt{int}] \\ X_0 <: A \multimap \texttt{int} &\vdash V[X_0] \\ X_1 <: A \multimap X_0,\ X_0 <: A \multimap \texttt{int} &\vdash V[X_1] \\ X_2 <: A \multimap X_1,\ X_1 <: A \multimap X_0,\ X_0 <: A \multimap \texttt{int} &\vdash V[X_2] \\ &\ldots \end{aligned}$$

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 wf. conditions:

$$\frac{\cdot \vdash A_0\ \textbf{type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1\ \textbf{type}}{\Gamma \vdash \forall X <: A_0.A_1\ \textbf{type}} \qquad \frac{\cdot \vdash A_0\ \textbf{type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1\ \textbf{type}}{\Gamma \vdash \exists X <: A_0.A_1\ \textbf{type}}$$

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:

$$(\textbf{rec}\ M(Y).(Y \multimap \forall X <: \textbf{top}.M[X]))[\texttt{int}] \tag{Ex.3}$$

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):

$$\begin{aligned} \cdot &\vdash M[\texttt{int}] \\ X_0 <: \textbf{top} &\vdash M[X_0] \\ X_1 <: \textbf{top}, X_0 <: \textbf{top} &\vdash M[X_1] \\ X_2 <: \textbf{top}, X_1 <: \textbf{top}, X_0 <: \textbf{top} &\vdash M[X_2] \\ &\dots \end{aligned}$$

By inversion on weakening of assumptions, we can consider the last context to only really require the "$X_2 <: \textbf{top}$" 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 (≡).

$$\frac{\dfrac{Z <: \textbf{top} \vdash M[Z] \quad\equiv\quad Z <: \textbf{top} \vdash M[Z]}{(X_2 <: \textbf{top})\{Z/X_2\} \vdash (M[X_2])\{Z/X_2\} \quad\equiv\quad (X_0 <: \textbf{top})\{Z/X_0\} \vdash (M[X_0])\{Z/X_0\}}}{X_2 <: \textbf{top}, X_1 <: \textbf{top}, X_0 <: \textbf{top} \vdash M[X_2] \quad\equiv\quad X_0 <: \textbf{top} \vdash M[X_0]}$$

We consider equivalence (≡) up to renaming of variables and weakening of assumptions defined as follows (for any two well-formed types, so that any premise must also obey type well-formedness):

$\Gamma \vdash A$	≡	$\Gamma \vdash A$		(equality)
$\Gamma_0, \Gamma_1 \vdash A_0$	≡	$\Gamma_2, \Gamma_3 \vdash A_1$	**if** $\Gamma_1 \vdash A_0 \equiv \Gamma_3 \vdash A_1$	(weakening)
$\Gamma_0 \vdash A_0$	≡	$\Gamma_1 \vdash A_1$	**if** $\Gamma_0\{Z/X\} \vdash A_0\{Z/X\} \equiv \Gamma_1\{Z/Y\} \vdash A_1\{Z/Y\}$ **and** Z **fresh**	(renaming type)
$\Gamma_0 \vdash A_0$	≡	$\Gamma_1 \vdash A_1$	**if** $\Gamma_0\{l/t'\} \vdash A_0\{l/t'\} \equiv \Gamma_1\{l/t\} \vdash A_1\{l/t\}$ **and** l **fresh**	(renaming loc)

The following lemmas (see appendix for proofs) state that there is a bound in the number of members of the set of types that an algorithm will recur on.

Lemma 8 (Finite Uses)**.** *Given a well-formed recursive type* (***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.*

Lemma 9 (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.*

Lemma 10 (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*.*

B Extra Examples

We now show some additional examples that illustrate the expressiveness of the full language, how we can model some high-level synchronization mechanisms, and clarify modularity gains when compared to prior works.

B.1 Stateful Pair, revisited

We revisit the pair example from [22] to highlight the changes to that base language. Note that this particular example does not need sharing. To clarify which location belongs to which variable, we prefix a location variable's name with an `@`. So that if x is a reference then it will have location with name '`@`x' and so on.

```
let newPair :[] ⊸ PairType = λ _.
 let left = new {} in
  let right = new {} in
   {
    initL :( int::(rw @left []) ) ⊸ ( []::(rw @left int) )
      = λ i.left := i,
    initR :( int::(rw @right []) ) ⊸ ( []::(rw @right int) )
      = λ i.right := i,
    sum :( []::((rw @right int) * (rw @left int)) ) ⊸ ( int::((rw @right int) * (rw @left int)) )
      = λ _.!left+!right,
    destroy :( []::((rw @right int) * (rw @left int)) ) ⊸ []
      = λ _.delete left; delete right
   }
  end
 end
end
```

The type of the `newPair` function remains unchanged from [22]:

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

Although the existential abstraction (i.e. packing) is done via subtyping rules rather than language constructs as in [22].

B.2 Launching Arbitrarily Many Threads

There is no restriction on the number of threads that can be forked. For instance, a recursive function can launch as many threads as it sees fit to compute some result, which may also cause the computation to diverge.

```
(λx.(rec R.fork (...);R))({})
```

B.3 Local Protocol Type

The type system only needs to know that two protocol's locations are related when we are composing those two protocols. Afterwards, they can proceed to be used independently (i.e. they are fictionally disjoint [19, 15]).

Thus, we can have a function that uses the protocol $\exists X.(\ X \Rightarrow \mathbf{rw}\ l\ \mathbf{int}\ ;\ P\ )$:

```
λx.( lock x; x := 1; unlock x )
```

Be typed with the following function type:

$$\forall P.\forall l.(\ (\ (!\mathbf{ref}\ l) :: \exists X.(\ X \Rightarrow \mathbf{rw}\ l\ \mathbf{int}\ ;\ P\ )\ )\ \multimap\ (\ [] :: P\ )\ )$$

which in turn can have the left protocol of the following split be applied as its argument:

$$\Gamma\ \vdash\ \mathbf{rw}\ p\ \mathbf{int}\ \ \Rrightarrow\ \ \exists X.(\ X \Rightarrow \mathbf{rw}\ p\ \mathbf{int}\ ;\ \mathbf{none}\ )\ \|\ \mathbf{rec}\ X.(\ \mathbf{rw}\ p\ \mathbf{int} \Rightarrow \mathbf{rw}\ p\ \mathbf{int}\ ;\ X\ )$$

Thus, the location relation only needs to be know when protocols are composed. Later they can be abstracted because we know that the protocols were checked to be safe.

B.4 Abstraction and Locking

Since our types express sharing, we can use standard techniques to abstract the components of a protocol after safe composition is checked. However, our system does not enforce deadlock-freedom and abstracting components of a protocol may make it harder for the programmer to tell if there is a chance for deadlock. When compared to [21], we lose the possibility to focus/lock on abstracted states since our locks require a set of locations to lock on. Still, we enable an abstraction to allow clients to lock its internal state through the use of some "module" that exposes that operation. For instance consider the following code block:

```
let x = new 0 in
 // share x as some protocols
 {
   lockMe = λ _.lock x,
   // ...
 }
```

Although `x` is not visible outside the code that creates that cell, clients can call the function to indirectly lock the underlying abstracted state. Thus, the record can have type:

$$[\ ...,\ \texttt{lockMe} : (\ [] :: (\mathsf{A} \Rightarrow \mathsf{B};\mathsf{C})\ )\ \multimap\ (\ [] :: (\mathsf{A} * (\mathsf{B};\mathsf{C}))\ ),\ ...\ ]$$

The client sees that some, abstracted protocol is expected by `lockMe`. This protocol must have been produced by some of the other functions of the record, since the client cannot see is representation. When `lockMe` returns, the client receives a type that expresses that A is available and that a guarantee (B; C) is expected to be fulfilled. However, there this fulfillment can only occur via the wrapper record as clients have no access to its representation.

B.5 Pipe Example, revisited

We now revisit the pipe example of [21] to clarify expressiveness gains.

In this example, two aliases interact through shared state such that one alias waits for another alias to insert an element into the shared cell. The protocols are not exactly identical to those shown in [21] due to changes to support thread-based concurrency. Namely, our protocol types need to ensure that a lock-token invariant is preserved, which changes how protocols express ownership recovery. Similarly, because abstractions are now typing artifacts (to support abstractions over protocols, since protocols are not values) this enables the inspection of shared state without destructive read as the linear component of a node can be placed "on the side" while leaving the content of the cell pure.

We use the following abbreviations for the states of the protocols:

$$\begin{aligned} \mathsf{Node}[p] &\triangleq \exists q.(\ (\mathbf{rw}\ p\ \texttt{Node\#![element : int, next : ref}\ q\texttt{]})\ *\ \mathsf{H}[q]\ ) \\ \mathsf{Close}[p] &\triangleq \mathbf{rw}\ p\ \texttt{Close\#![]} \\ \mathsf{Empty}[p] &\triangleq \mathbf{rw}\ p\ \texttt{Empty\#![]} \end{aligned}$$

And finally, we define the protocols for the head and tail aliases:

$$\begin{aligned} \mathsf{T}[p] &\triangleq \mathsf{Empty}[p] \Rightarrow (\ \mathsf{Close}[p]\ \oplus\ \mathsf{Node}[p]\ )\ ;\ \mathbf{none} \\ \mathsf{H}[p] &\triangleq (\ \mathsf{Node}[p]\ \Rightarrow\ \mathsf{Node}[p]\ ;\ \mathsf{Node}[p]\ )\ \oplus \\ &\qquad (\ \mathsf{Close}[p]\ \Rightarrow\ \mathsf{Close}[p]\ ;\ \mathsf{Close}[p]\ )\ \oplus\ (\ \mathsf{Empty}[p] \Rightarrow \mathsf{Empty}[p]\ ;\ \mathsf{H}[p]\ ) \end{aligned}$$

Note that 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. We could also use $\exists$ abstraction over that step.

Possible pipe implementation:

```
let newPipe = λ _.
  let node = new Empty#{} in
   // using explicit sharing annotation to make it clear how state is split
   share (rw @node Empty#[]) as H[@node] || T[@node];
   let h = new node in
    let t = new node in
    {
    put  : int :: ∃p.((rw @t (ref p)) * T[p]) ⊸ [] :: ∃p.((rw @t (ref p)) * T[p])
     = λ e.
      let last = new Empty#{} in
        let old = !t in
          lock old;
          share (rw @last Empty#[]) as H[@last] || T[@last];
          old := Node#{ element = e , next = last };
          unlock old;
          t := last;
        end
      end,
```

```
19
20    close : int :: ∃p.((rw @t (ref p)) ∗ T[p]) ⊸ []
21     = λ _.
22      let last = !t in
23        delete t;
24        lock last;
25        last := Closed#{};
26        unlock last
27      end,
28
29    tryTake : [] :: ∃p.((rw @h (ref p)) ∗ H[p]) ⊸
30                  NoResult#([] :: ∃p.((rw @h (ref p)) ∗ H[p])) + Result#(int :: ∃p.((rw @h (ref p)) ∗ H[p])) + Depleted#[]
31     = λ _.
32      let first = = !h in
33        lock first;
34        case !first of
35          Empty#_ →
36            unlock first;
37            NoResult#{}
38        | Closed#_ →
39            unlock first;
40            delete h;
41            delete first;
42            Depleted#{}
43        | Node#n →
44            unlock first;
45            h := n.next;
46            delete first;
47            Result#n.element
48        end
49      end
50    }
51    end
52    end
53  end
54  in
55  // ...
```

Since existential types are automatically opened, we use the `@` prefix on a variable to denote its location variable that was automatically opened. Therefore, if `t` is a **ref** a then `@t` denotes location a. Also note that the function's type is used to guide type checking, meaning that the type the programmer assigns to the function will help guide the type system to move capabilities around and abstract types to match the desired target type.

We now show how protocol composition proceeds to check that the following split of location

p shares the state safely:

$$\Gamma \vdash \mathsf{Empty}[p] \Rrightarrow \mathsf{T}[p] \parallel \mathsf{H}[p]$$

Protocol composition yields the following set of configurations:

$$\begin{array}{llrcrcr}
\{ & \langle\ \Gamma \vdash & \mathsf{Empty}[p] & \Rrightarrow & \mathtt{T}[p] & \parallel & \mathtt{H}[p] \ \rangle, \\
& \langle\ \Gamma \vdash & \mathsf{Close}[p] \oplus \mathsf{Node}[p] & \Rrightarrow & \textbf{none} & \parallel & \mathtt{H}[p] \ \rangle, \\
& \langle\ \Gamma \vdash & \mathsf{Close}[p] & \Rrightarrow & \textbf{none} & \parallel & \mathsf{Close}[p] \ \rangle, \\
& \langle\ \Gamma \vdash & \mathsf{Node}[p] & \Rrightarrow & \textbf{none} & \parallel & \mathsf{Node}[p] \ \rangle, \\
& \langle\ \Gamma \vdash & \textbf{none} & \Rrightarrow & \textbf{none} & \parallel & \textbf{none} \ \rangle\ \}
\end{array}$$

B.6 Futures

Our encoding of a `future` follows that of `fork`/`join` but wrapped in an "object" interface with `get` and `isDone` methods. A thread will compute some result that is then assigned to a shared cell (i.e. a thunk, a function that takes no argument and computes an expression that was to be evaluated lazily). If the object representing the future invokes `get` then the calling thread will block until the computing thread finishes the result, while `isDone` enables inspection on whether the computation has finished. To enable the uses of `isDone`, the `M` protocol shown in the paper is extended with an additional choice to allow non-destructive inspections on the `Done` tag. Because our protocols are first-class, we can have thunks that capture part of the shared state on its body. Although the code is interference-free, it may still deadlock.

$$\begin{array}{rcl}
\mathsf{E} & \triangleq & \textbf{rw}\ t\ \mathtt{Empty\#[]} \\
\mathsf{D} & \triangleq & \textbf{rw}\ t\ \mathtt{Done\#}\textbf{int} \\
\mathsf{D}[X] & \triangleq & \textbf{rw}\ t\ \mathtt{Done\#}X
\end{array}$$

$$\begin{array}{rcl}
\mathtt{Join} & \triangleq & \mathsf{E} \Rightarrow \mathsf{D}; \textbf{none} \\
\mathtt{End} & \triangleq & (\ \mathsf{E} \Rightarrow \mathsf{E}; \mathtt{End}\ ) \oplus (\ (\mathsf{D} \Rightarrow (\textbf{rw}\ t\ \textbf{int}); (\textbf{rw}\ t\ \textbf{int})) \ \&\ \exists Y.(\ \mathsf{D}[Y] \Rightarrow \mathsf{D}[Y]; \mathtt{End}\ )\ )
\end{array}$$

```
1  future e ≜
2   let f = λ_.e in
3   let s = new Empty#{} in
4    share s as Join || End;
5
6    fork // computes result
7      (let r = Done#(f {}) in
8        lock s;
9        s := r;
10       unlock s
11     end);
12
13   // methods:
14   {
```

```
  get = λ_.( rec R.
      lock s;
      case !s of
        Empty#x →
          s := Empty#x;
          unlock s;
          R // retry
      | Done#a →
          s := {};
          unlock s;
          delete s;
          a // result of computation
      end
    end),

   isDone = λ_.(
      lock s;
      case !s of
        Empty#x →
          s := Empty#x;
          unlock s;
          NotDone#{}
      | Done#a →
          s := Done#a;
          unlock s;
          Done#{}
      end
    end)
 }
end
end
```

Client code:

```
let f = future fact(300) in
 ...
 // get value computed by future
 f.get()
end
```

B.7 Barrier

A barrier enables a fixed number of threads to wait for all those threads to reach the barrier. Only after all threads have reached the barrier will any thread be allowed to continue.

To implement this barrier, we need two phases: one to signal that a thread has reached the barrier, and another phase to make sure that all threads have seen that the barrier synchronization

was completed (so that the barrier's state can be safely destroyed). Our core language does not support reasoning about integers, being limited to a finite number of tags of a sum type. Because of this, the implementation in the core language will enforce a strict order in changing the state of the barrier—but which does not limit the barrier's use since this ordering is not observable to clients.

In the core language. To create a barrier for N threads, we will essentially have two kinds of protocols: those that join the barrier at a particular stage, and the last thread that will recover uniqueness of the barrier's state. Note that the `await` function will both wait for the thread's correct position in the scheme to be reached and, once it is reached, to change the barrier's tag appropriately to acknowledge that the thread saw that all threads had reach the barrier.

For clarity of the example, we will show how this protocol can be constructed when $N = 3$, where the state starts with the `Thread0` tag. For brevity, we only list the tags contained in some cell and not the full type of the capability to that location.

```
Thread1 ≜ Thread0 ⇒ Thread1 ; rec X.( Thread1 ⇒ Thread1; X ⊕
                                      Thread2 ⇒ Thread2; X ⊕
                                      Ack0 ⇒ Ack1 ; none )

Thread2 ≜ rec X.( Thread0 ⇒ Thread0; X ⊕
                  Thread1 ⇒ Thread2; rec Y.( Thread2 ⇒ Thread2; Y ⊕
                                             Ack0 ⇒ Ack0 ; Y ⊕
                                             Ack1 ⇒ Ack2 ; none ) )

Thread3 ≜ rec X.( Thread0 ⇒ Thread0; X ⊕
                  Thread1 ⇒ Thread1; X ⊕
                  Thread2 ⇒ Ack0; rec Y.( Ack0 ⇒ Ack0; Y ⊕
                                          Ack1 ⇒ Ack1; Y ⊕
                                          // all done, recovers shared state
                                          Ack2 ⇒ rw t [] ; rw t [] ) )
```

Naturally, the underlying pattern used to create these protocols could be generalized to facilitate the creation of similar protocols for an arbitrary number of threads. However, our core language is limited to only list a finite number of constants that are used to tag the different values of a sum type. Ideally, we would instead use integers to reason more abstractly about these tags so that we do not need to check each individual protocol but could instead consider a more abstract protocol representation that scales better to higher number of threads.

While in this version we must treat each tag individually, with "abstract" integers we can logically collapse arbitrary large number of tags into a single case as will be exemplified in the next paragraph.

Client code:

```
1 let await = barrier(3) in
2   fork ( ...; await(); ... );
```

```
3   fork ( ...; await(); ... );
4   ...
5   await();
6   ...
7 end
```

Integers instead of tags We now sketch how we can model barriers in a language where integers can be reasoned with abstractly, for instance using type refinements. In this case, we only have one kind of protocol and the value found in the shared cell distinguishes the allowed behavior of the thread. Unlike previously, this also means that we can join without statically assigning threads a particular order of joining (although that does not affect how the barrier can be used).

We assume that the cell to be shared initially has type "**rw** p 0" and the integer value will be used to count how many threads have reached the barrier and seen the barrier's been reached by all involved threads.

```
Barrier(N) ≜ // signaling phase
             ∃{x : int}.( rw p x ⇒ rw p x+1 ) ;
             // "waiting" point
             rec R.(
              // retry in case value is less than N
              ∃{x : int | x < N}.( rw p x ⇒ rw p x ; R ) ⊕
              // acknowledge, in case greater than N but not the last
              ∃{x : int | x >= N ∧ x < (2N − 1)}.( rw p x ⇒ rw p x+1 ; none ) ⊕
              // recover ownership, in case it is the last thread to acknowledge
              ∃{x : int | x == (2N − 1)}.( rw p x ⇒ rw p null ; rw p null )
             )
```

B.8 Merge Task

While in the `fork`/`join` case we statically know which thread will compute the value and which one will wait for that value, in here we design a more dynamic interaction where both threads are computing some result and the last to complete its task will merge the two results together. We can use the following protocol for the case where there are only two participants:

$$(\ \mathsf{Zero} \Rightarrow \mathsf{One}\ ;\ \textbf{none}\ )\ \oplus\ (\ \mathsf{One} \Rightarrow \mathsf{Two}\ ;\ \mathsf{Two}\ )$$

The shared cell goes over the state: Zero, when no result is ready, One when one of the results is done, and Two when the last thread completed its task. Note that by using a finite number of labels to model these states, we are limited on how many times this cell can be shared. In this case, limited to only two aliases as the protocol above can only be split twice. However, we could use iterative re-splitting to continue to aliases further more by essentially extending the termination step that recovers ownership.

Using Extensions Next, we can sketch the protocol that we would use if our language was able to reason about integers. We use an auxiliary cell to work as the reference counter, since we will be sharing the state without bounds. Likewise, we do not impose any formal conditions on how the result of each thread should be merged together, only that it is of type **int**.

```
Merger ≜
  ∃{n : int | n > 0}.
    ( (rw p int) * (rw c n) ⇒ (rw p int) * (rw c n-1) ; none )
  ⊕
  ∃{n : int | n == 0}.
    ( (rw p int) * (rw c 0) ⇒ (rw p int) * (rw c 0) ; (rw p int) * (rw c 0) )
```

To ensure that only one will recover the shared state, we see that the cell c must contain a integer value with the number of aliases to that state (i.e. if 10 aliases share the state then c should contain 9 for the last one to match the 0 value).

B.9 Shared Pair

This example shows the case where neither alias sees the "full picture" of the contents of the shared state. The example is meant to illustrate the expressiveness of our system, since the example itself is somewhat artificial. However, it shows that 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:

$$\mathbf{rw}\ p\ [A, B]$$

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 composition will have to proceed. 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, composition will ensure that this interference is accounted for although it is abstracted. We write $\forall X$ to abbreviate $\forall X <: \mathbf{top}$ and $\exists X$ for $\exists X <: \mathbf{top}$:

$$\begin{array}{lcl} \mathtt{P}[A][B] & \triangleq & \mathbf{rw}\ p\ [A, B] \\ \mathtt{L}[A] & \triangleq & \exists X.(\ \mathtt{P}[A][X] \Rightarrow \forall Y.(\ \mathtt{P}[Y][X]\ ;\ \mathtt{L}[Y]\ )\ ) \\ \mathtt{R}[A] & \triangleq & \exists X.(\ \mathtt{P}[X][A] \Rightarrow \forall Y.(\ \mathtt{P}[X][Y]\ ;\ \mathtt{R}[Y]\ )\ ) \end{array}$$

$$\Gamma\ \vdash\ \mathtt{P}[X][Y]\ \Rrightarrow\ \mathtt{L}[X] \parallel \mathtt{R}[Y]$$

Note that the existential cannot last longer than the interval where no interference from the other alias may "appear" in the pair. Thus, the recursion variable X must include a fresh existential 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:

$$
\begin{array}{lllllllll}
\{ & ❶ & \langle & \Gamma & \vdash & \mathtt{P}[A][B] & \Rrightarrow & \mathtt{L}[A] & \| \quad \mathtt{R}[B] \quad \rangle, \\
& ❷ & \langle & \Gamma, X : \textbf{type} & \vdash & \mathtt{P}[A][X] & \Rrightarrow & \mathtt{L}[A] & \| \quad \mathtt{R}[X] \quad \rangle, \\
& ❸ & \langle & \Gamma, Y : \textbf{type} & \vdash & \mathtt{P}[Y][B] & \Rrightarrow & \mathtt{L}[Y] & \| \quad \mathtt{R}[B] \quad \rangle, \\
& ❹ & \langle & \Gamma, X : \textbf{type}, \ Y : \textbf{type} & \vdash & \mathtt{P}[X][Y] & \Rrightarrow & \mathtt{L}[X] & \| \quad \mathtt{R}[Y] \quad \rangle \ \}
\end{array}
$$

C Examples using Informal Extensions

C.1 Monotonic Counter

In this example, we make use of an informal extension to allow types akin to *refinement types* [12] to model more precise states. With this extension we allow more fine-grained interactions than what was shown in the paper. While prior work already showed how a simple two-states counter could be modeled, we show here how a more precise and flexible typing can be built using the protocol type constructs we discussed above.

The example encodes a monotonic counter, similar to what is done in [28, 14], but encoded in a rely-guarantee protocol. The type will share a single location, q, using the protocol below:

$$\texttt{MCounter}[m] \triangleq \exists\{j : \textbf{int} \mid j \geq m\}.(\ \textbf{rw}\ q\ j\ \Rightarrow \forall\{i : \textbf{int} \mid i \geq j\}.(\ \textbf{rw}\ q\ i\ ;\ \texttt{MCounter}[i]\ )\ )$$

where `MCounter`$[m]$ is indexed by a type m that represents the lower bound of the counter. The guarantee of the step ensures that the change to that counter will be greater or equal than the base of the counter. Thus, we have that the types m, j, i represent integers.

Since the protocol is not depending on a specific value in the state (the value on each step is existentially quantified), the protocol can be shared arbitrarily. However, through the universal quantifier, the protocol is also able to store a more precise lower bound on the value contained in q. This protocol ensures that the value stored in q, although it can undergo intermediate uses, will keep increasing. Effectively, the interference allowed on that cell is iteratively/monotonically being narrowed down on each step of the protocol. The protocol itself is usable without restriction, meaning that it never terminates and can be freely split into other protocols that obey the same initial behavior over that shared cell.

Client code We now exemplify some client code that uses the protocol above. Since our formal system does not have refinement types, their use in this code is somewhat informal. Still the intention is to show how a more precise monotonic counter can be modeled by a rely-guarantee protocol:

```
1  Γ = x : ref p , p : loc
2  Δ = ∃n.( rw p n ⇒ ∀m : m ≥ n.( rw p m ; ... ) )
3  // opens existential (note language construct hidden)
4  Γ = x : ref p , p : loc , n : type
5  Δ = rw p n ⇒ ∀m : m ≥ n.( rw p m ; ... )
6    lock x; // locks cell
7  Γ = x : ref p , p : loc , n : type
8  Δ = rw p n , ∀m : m ≥ n.( rw p m ; ... )
9    x := x-1;
10 Δ = rw p (n − 1) , ∀m : m ≥ n.( rw p m ; ... )
11   // would be type error to unlock at this point
12   x := x+40;
13 Δ = rw p (n + 39) , ∀m : m ≥ n.( rw p m ; ... )
14   // by subtyping we can apply "(n+39)" as m:
```

```
15 Δ = rw p (n + 39) , rw p (n + 39) ; ...
16   // thus, we can now unlock
17   unlock x;
18 Δ = ∃o ≥ (n + 39).( rw p o ⇒ ... )
```

Composition We now discuss the protocol composition configurations:

$$
\begin{array}{lllllllll}
\{ & ❶ & \langle\, m : \mathbf{int} & \vdash & \mathbf{rw}\ q\ m & \Rrightarrow & \texttt{MCounter}[m] & \| & \texttt{MCounter}[m] \ \rangle, \\
& ❷ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m & \vdash & \mathbf{rw}\ q\ n & \Rrightarrow & \texttt{MCounter}[n] & \| & \texttt{MCounter}[m] \ \rangle, \\
& ❸ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m & \vdash & \mathbf{rw}\ q\ n & \Rrightarrow & \texttt{MCounter}[m] & \| & \texttt{MCounter}[n] \ \rangle, \\
& ❹ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m\ ,\ o : o \geq n & \vdash & \mathbf{rw}\ q\ o & \Rrightarrow & \texttt{MCounter}[n] & \| & \texttt{MCounter}[o] \ \rangle, \\
& ❺ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m\ ,\ o : o \geq n & \vdash & \mathbf{rw}\ q\ o & \Rrightarrow & \texttt{MCounter}[o] & \| & \texttt{MCounter}[n] \ \rangle\ \}
\end{array}
$$

To close the set of configurations, we must extend our definition of equi-variance to consider states up to the transitivity of $\geq$. Thus, continuing to step either protocol will just generate equivalent configurations.

$$
\begin{array}{lllllllll}
\{ & ❶ & \langle\, m : \mathbf{int} & \vdash & \texttt{MCounter}[m] & \Rrightarrow & \texttt{MCounter}[m] & \| & \texttt{MCounter}[m] \ \rangle, \\
& ❷ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m & \vdash & \texttt{MCounter}[n] & \Rrightarrow & \texttt{MCounter}[n] & \| & \texttt{MCounter}[m] \ \rangle, \\
& ❸ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m & \vdash & \texttt{MCounter}[n] & \Rrightarrow & \texttt{MCounter}[m] & \| & \texttt{MCounter}[n] \ \rangle, \\
& ❹ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m\ ,\ o : o \geq n & \vdash & \texttt{MCounter}[o] & \Rrightarrow & \texttt{MCounter}[n] & \| & \texttt{MCounter}[o] \ \rangle, \\
& ❺ & \langle\, m : \mathbf{int}\ ,\ n : n \geq m\ ,\ o : o \geq n & \vdash & \texttt{MCounter}[o] & \Rrightarrow & \texttt{MCounter}[o] & \| & \texttt{MCounter}[n] \ \rangle\ \}
\end{array}
$$

The case where there is a mismatch on the lower bound of the protocol to be split and the resulting protocol is not completely trivial. It is important to note the subtlety on enabling a protocol to work with a lower bound on the value of the cell. Indeed, this just amounts to weakening the rely type since the guarantee type is dependent on an abstracted type that, consequently, remains unchanged—we just have a less precise bound on that value. Therefore, we have that:

$$\texttt{MCounter}[n] <: \texttt{MCounter}[m] \qquad n \geq m$$

since we are weakening the rely type but leave the effective guarantee type unchanged.

Discussion Prior work in the area makes a distinction in the uses of the monotonic counter based on whether the action is enforced or simply available to clients if they choose to use it. In our system, these uses can be modeled by the function type depending on whether the linear protocol resource is used or not.

For instance, the type:

$$\exists q.\exists n.(\ (!\mathbf{ref}\ q) :: \texttt{MCounter}[n]\ ) \multimap \exists m.(\ [] :: \texttt{MCounter}[m]\ )$$

Simply allows the counter to be used an arbitrary number of times, i.e. it allows the action of the counter but does not enforce it. If the client so chooses it can use the monotonic counter zero or more times.

On the other hand, if we use the type that unfolds a single step of the protocol:

$$\exists q.\exists m.\forall X.(\ (!\mathbf{ref}\ q) :: \exists\{j : \mathbf{int} \mid j \geq m\}.(\ \mathbf{rw}\ q\ j\ \Rightarrow \forall\{i : \mathbf{int} \mid i \geq j\}.(\ \mathbf{rw}\ q\ i\ ;\ X\ )\ ) \multimap (\ [] :: X\ )\ )$$

Our system can enforce that a single action of the monotonic counter is used by the function, thus that the counter is really incremented once. Alternatively, X could instead just be the monotonic counter's type, thus encoding that the counter is used once or more times. (Note: the use of abstractions over refined types would need an improved kind system but we simplify this by using plain X : **type** here).

Consequently, we are able to have a more precise type than prior works and model different kinds of uses within the same protocol framework.

C.2 Monotonically Growing List

Encoding Read-Only A read-only capability (**ro** p A) implies that the cell will only be read, never written to. Interestingly, since we are employing a model where all accesses to some shared state must be protected by a lock (granting exclusive access to that location), we can encode a meaning similar to (exclusive) read-only access without explicitly requiring a read-only capability. Essentially, if there will not be any concurrent accesses to the shared cell, a shared read-only usage can be modeled as the following step in a protocol:

$$\exists X.(\ \mathbf{rw}\ p\ X \Rightarrow \mathbf{rw}\ p\ X\ ;\ \ldots\ )$$

since it will enforce that the cell has to remain publicly unchanged (since any temporary write is not observable), although that may only occur after some private uses. Thus, the use "looks" like read-only to any other alias of that state.

Naturally, if concurrent read-only accesses were possible then we would need a proper **ro** capability to ensure that no intermediate, private values are placed in that shared cell. Otherwise, we could also define a different $\Rightarrow$ stepping that forbade private uses and requires a *direct* transition from the rely to the guarantee type.

In this example we are modeling a monotonically increasing singly linked list, similar to what is done in [14]. All the nodes of the list are technically immutable, so that the pointer to the next node of the list cannot be changed. However, the list contains a head pointer which can *prepend* new nodes to that list. Since each of those nodes is immutable, each node could also be shared without bounds which would effectively model an immutable spaghetti stack if multiple shared nodes could point to one same (immutable) next cell.

The interesting aspect of modeling this example is how we can constrain the actions of the head pointer of the list. To illustrate this point, we use a list that already contains one element which simplifies our protocol representation. With this protocol, we can enforce a more specific change to the shared state: the head pointer is required to add one (and only one) new node that points to the *exact* same list node that was previously in that shared state, if the shared state is used at all. The list protocol will ensure that, regardless of how many aliases to that list exist, if any alias uses the list then it must forcefully prepend just a single node to the list as its sole action.

$$\begin{aligned}
\texttt{N}[p] &\triangleq \texttt{Node\#[element} : \mathbf{int}\ ,\ \texttt{next} : \mathbf{ref}\ p\ \texttt{]} \\
\texttt{Nil}[p] &\triangleq \mathbf{ro}\ p\ \texttt{Nil\#[]} \\
\texttt{Node}[p] &\triangleq \exists q.(\ (\ \mathbf{ro}\ p\ \texttt{N}[q]\ )\ *\ (\ \texttt{Node}[q]\ \oplus\ \texttt{Nil}[q]\ )\ ) \\
\\
\texttt{RootNode}[q] &\triangleq (\ \mathbf{rw}\ p\ \texttt{N}[q]\ )\ *\ (\ \texttt{Node}[q]\ \oplus\ \texttt{Nil}[q]\ ) \\
\texttt{L}[p] &\triangleq \exists q.(\ (\ \mathbf{rw}\ p\ \texttt{N}[q]\ )\ *\ (\ \texttt{Node}[q]\ \oplus\ \texttt{Nil}[q]\ )\ \Rightarrow \\
&\qquad \forall t.(\ (\ \mathbf{rw}\ p\ \texttt{N}[t]\ )\ *\ (\ \mathbf{ro}\ t\ \texttt{N}[q]\ )\ *\ (\ \texttt{Node}[q]\ \oplus\ \texttt{Nil}[q]\ )\ )\ );\ \texttt{L}[p]
\end{aligned}$$

The important aspect is the constraint on the location that the new guaranteed cell must point to. This means that we ensure the new cell that was added correctly points to the previous cell of the list, as we had at the moment of focus. The protocol explicitly requires a prepend to occur. We

could also include (through the use of &) additional uses that could, for instance, enable the client to chose whether there should or not be changes to the list, so that iterating would be possible. Since our focus is on only showing the prepend function, we omit that extra case.

We now show the implementation of the `prepend` function, that is defined as a closure with access to the list's `head` pointer. The `h`(ead) pointer is a reference to a location (`@h`) that is shared using the `L[@h]` protocol for that location.

```
1  Γ = @h : loc , h : ref @h
2  Δ = L[@h]
3  prepend = λ x.
4      Γ = x : int , @h : loc , h : ref @h
5      Δ = L[@h]
6      // automatic open of existential of L[@h]
7      Γ = t : loc , x : int , @h : loc , h : ref @h
8      Δ = ( ( rw @h N[t] ) ∗ ( Node[t] ⊕ Nil[t] ) ⇒ ... )
9      lock h; // locks location of head, i.e. '@h'
10     Δ = rw @h N[t] , Node[t] ⊕ Nil[t] , ...
11     ... , ∀q.( ( ( rw @h N[q] ) ∗ ( ro q N[t] ) ∗ ( Node[t] ⊕ Nil[t] ) ) ; L[@h] )
12     let n = new Node#{ element = !h.element, next = !h.next } in
13       Δ = n : ∃w.( rw w N[t] )
14       // automatic open
15       Γ = ... , w : loc , n : ref w
16       Δ = ... , rw w N[t]
17       // subtype to readonly, this cannot be reversed back to writable!
18       Δ = ... , ro w N[t]
19       Δ = ... , rw @h N[t] , ro w N[t]
20       h := Node#{ element = x , next = n };
21       Δ = ... , rw @h N[w] , ro w N[t]
22       // automatically apply location 'w' to guarantee:
23       Δ = ... , ( ( ( rw @h N[w] ) ∗ ( ro w N[t] ) ∗ ( Node[t] ⊕ Nil[t] ) ) ; L[p] )
24     end;
25     // it is ok to unlock since we are at the desired guarantee
26     unlock h
27     Δ = L[@h]
```

Protocol composition becomes easier to grasp once the following subtyping relation between states is observed:

$$
\begin{aligned}
&\forall t.(\,(\,\mathbf{rw}\ p\ \mathtt{N}[t]\,)\ *\ \underline{(\,\mathbf{ro}\ t\ \mathtt{N}[q]\,)\ *\ (\,\mathtt{Node}[q]\ \oplus\ \mathtt{Nil}[q]\,)}\,)\\
&<:\\
&\forall t.(\,(\,\mathbf{rw}\ p\ \mathtt{N}[t]\,)\ *\ \underline{\exists q.(\,(\,\mathbf{ro}\ t\ \mathtt{N}[q]\,)\ *\ (\,\mathtt{Node}[q]\ \oplus\ \mathtt{Nil}[q]\,)\,)}\,)\\
&<:\\
&\forall t.(\,(\,\mathbf{rw}\ p\ \mathtt{N}[t]\,)\ *\ \underline{\mathtt{Node}[t]}\,)\\
&<:\\
&\forall t.(\,(\,\mathbf{rw}\ p\ \mathtt{N}[t]\,)\ *\ (\,\mathtt{Node}[t]\ \oplus\ \mathtt{Nil}[t]\,)\,)
\end{aligned}
$$

Therefore, we have that the rely and guaranteed states are very similar except for referring different location variables. Once this is considered, protocol composition is straightforward.

If the split state is $\texttt{Node}[p]$, then we have the following set of configurations:

$$\begin{array}{lllllllll}
\{ & \textbf{❶} & \langle \cdot & \vdash & \exists q.\texttt{RootNode}[q] & \Rrightarrow & \texttt{L}[p] & \| & \texttt{L}[p] \quad \rangle, \\
& \textbf{❷} & \langle\, t : \textbf{loc} & \vdash & \texttt{RootNode}[t] & \Rrightarrow & \texttt{L}[p] & \| & \texttt{L}[p] \quad \rangle\ \}
\end{array}$$

Which is the result of the steps:

$$\dfrac{\dfrac{\langle q : \textbf{loc} \vdash (\textbf{rw}\ p\ \texttt{N}[q]) * (\ldots) \Rrightarrow (\textbf{rw}\ p\ \texttt{N}[q]) * (\ldots) \Rightarrow \forall t.(\ldots)\ ;\ \texttt{L}\rangle \mapsto \langle q : \textbf{loc}, t : \textbf{loc} \vdash \texttt{Node}[t] \Rrightarrow \texttt{L}\rangle}{\langle \cdot \vdash \exists q.((\textbf{rw}\ p\ \texttt{N}[q]) * (\ldots)) \Rrightarrow \exists q.((\textbf{rw}\ p\ \texttt{N}[q]) * (\ldots) \Rightarrow \forall t.(\ldots))\ ;\ \texttt{L}\rangle \mapsto \langle t : \textbf{loc} \vdash \texttt{Node}[t] \Rrightarrow \texttt{L}\rangle}}{\textbf{❶}\langle \cdot \vdash \exists q.\texttt{RootNode}[q] \Rrightarrow \texttt{L}\rangle \mapsto \langle t : \textbf{loc} \vdash \texttt{Node}[t] \Rrightarrow \texttt{L}\rangle\ \textbf{❷}}$$

$$\dfrac{\dfrac{\langle t : \textbf{loc} \vdash (\textbf{rw}\ p\ \texttt{N}[t]) * (\ldots) \Rrightarrow ((\textbf{rw}\ p\ \texttt{N}[q]) * (\ldots) \Rightarrow \forall l.(\ldots)\ ;\ \texttt{L})\{q/t\}\rangle \mapsto \langle l : \textbf{loc}, t : \textbf{loc} \vdash \texttt{Node}[l] \Rrightarrow \texttt{L}\rangle}{\langle t : \textbf{loc} \vdash (\textbf{rw}\ p\ \texttt{N}[t]) * (\ldots) \Rrightarrow \exists q.((\textbf{rw}\ p\ \texttt{N}[q]) * (\ldots) \Rightarrow \forall l.(\ldots))\ ;\ \texttt{L}\rangle \mapsto \langle l : \textbf{loc} \vdash \texttt{Node}[l] \Rrightarrow \texttt{L}\rangle}}{\textbf{❷}\langle t : \textbf{loc} \vdash \texttt{RootNode}[t] \Rrightarrow \texttt{L}\rangle \mapsto \langle l : \textbf{loc} \vdash \texttt{Node}[l] \Rrightarrow \texttt{L}\rangle\ \textbf{❷}}$$

D 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, by using an auxiliary cell to store values and just move that pointer (and capability) between the threads via the channel (modeled as a shared cell).

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.

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 negates 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 13. The underlying principle of these idioms is to hide the waiting states of the channel. Client code will then look identical to what

```
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 13: Possible encoding of `send` and `receive` functions.

you would normally see on 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 sharing 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 on 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:

$$\mathsf{WaitSteps}[A, B, ...] \oplus \ (\ \mathbf{rw}\ \mathtt{c}\ \mathtt{ReadyToReceive\#}V \Rightarrow \mathbf{rw}\ \mathtt{c}\ \mathtt{idle\#}[]\ ;\ \ldots\ )$$

where WaitSteps 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 send, we have:

$$\mathsf{WaitSteps}[A, B, ...] \oplus \ (\ \mathbf{rw}\ \mathtt{c}\ \mathtt{idle\#}[] \Rightarrow \mathbf{rw}\ \mathtt{c}\ \mathtt{ReadyToReceive\#}V\ ;\ \ldots\ )$$

Note that, expanding $\mathsf{WaitSteps}[A, B]$ above yields the protocol:

$$\mathbf{rec}\ X.(\ (\ A \Rightarrow A; X\ ) \oplus (\ B \Rightarrow B; X\ ) \oplus (\ \mathbf{rw}\ \mathtt{c}\ \mathtt{idle\#}[] \Rightarrow \mathbf{rw}\ \mathtt{c}\ \mathtt{ReadyToReceive\#}V\ ;\ \ldots\ )\ )$$

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.

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 phase is hidden from the programmer, our receive and send may cause the current thread to "busy-waiting" when an old value is still in the shared cell. (Although overwrites may be allowed in certain situations, so that waiting is unnecessary but may cause the interaction to be somewhat non-deterministic which is not usual for message-passing.) 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 approach the problem of finding better ways to schedule threads to reduce excessive spinning.

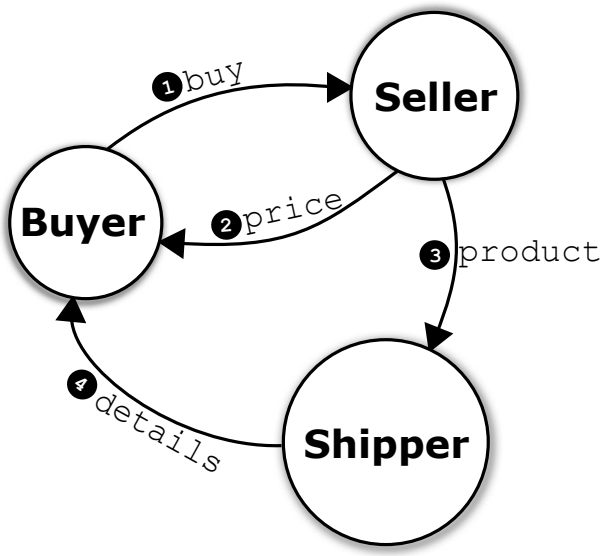

Figure 14: Buyer-Seller-Shipper message-passing example.

D.1 Buyer-Seller-Shipper Example

The example of Figure 14 (adapted from [5]) 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 listing the shipping details back to the buyer.

We follow the original example in using "session-delegation" instead of opening a private communication channel between the shipper and the seller, although our system can also model that. Similarly, to simplify the presentation, we do not abstract a channel's internal states even though such 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 "$\rightsquigarrow$" to connect or delegate the communication.

$$\begin{array}{rcl} \mathsf{Buyer} & : & \mathsf{Seller} \triangleleft \mathsf{buy}!(\mathit{prod}) \,;\, \mathtt{price}?(p) \,;\, \mathtt{details}?(d) \\ \mathsf{Seller} & \rightsquigarrow & \mathsf{buy}?(\mathit{prod}) \,;\, \mathtt{price}!(p) \,;\, \mathsf{Shipper} \triangleleft \mathtt{product}!(\mathit{prod}) \\ \mathsf{Shipper} & \rightsquigarrow & \mathtt{product}?(\mathit{prod}) \,;\, \mathtt{details}!(d) \end{array}$$

The types above define a Buyer type that initializes a communication ($\triangleleft$) 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 ($\rightsquigarrow$), the seller reads the product that is to be bought, sends back the price, and then connects to Shipper to send the *prod*uct information while delegating to Shipper the remainder of the communication. Finally, Shipper,

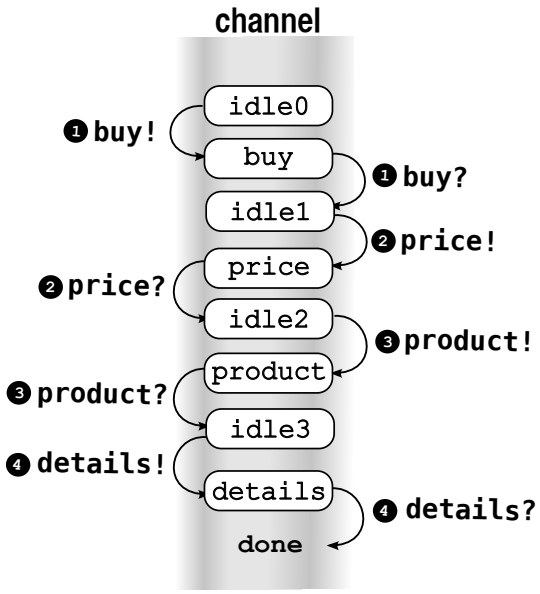

Figure 15: Buyer-Seller-Shipper shared channel's changing session state.

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 15 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" step, although our $\overline{A}$ will be limited to listing tags and not the full type for simplicity. 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 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:

$$\begin{array}{rl}
\texttt{Buyer}: & \mathbf{rw}\ c\ \texttt{idle0\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{buy\#}\mathit{prod}\ ; \\
& \mathbf{wait}\ \texttt{buy}, \texttt{idle1}\ \mathbf{else}\ \ \mathbf{rw}\ c\ \texttt{price\#}p \Rightarrow \mathbf{rw}\ c\ \texttt{idle2\#}[]\ ; \\
& \mathbf{wait}\ \texttt{idle2}, \texttt{idle3}, \texttt{product}\ \mathbf{else}\ \ \mathbf{rw}\ c\ \texttt{details\#}d \Rightarrow \mathbf{rw}\ c\ []\ ; \\
& \mathbf{rw}\ c\ [] \\
\\
\texttt{Seller}: & \mathbf{wait}\ \texttt{idle0}\ \mathbf{else}\ \ \mathbf{rw}\ c\ \texttt{buy\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{idle1\#}[]\ ; \\
& \mathbf{rw}\ c\ \texttt{idle1\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{price\#}p\ ; \\
& \mathbf{wait}\ \texttt{price}\ \mathbf{else}\ \ \mathbf{rw}\ c\ \texttt{idle2\#}[] \Rightarrow \mathbf{rw}\ c\ \texttt{product\#}\mathit{prod}\ ; \\
& \mathbf{none} \\
\\
\texttt{Shipper}: & \mathbf{wait}\ \texttt{idle0}, \texttt{idle1}, \texttt{idle2}, \texttt{buy}, \texttt{price}\ \mathbf{else} \\
& \mathbf{rw}\ c\ \texttt{product\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{idle3\#}[]\ ; \\
& \mathbf{rw}\ c\ \texttt{idle3\#}\mathit{prod} \Rightarrow \mathbf{rw}\ c\ \texttt{details\#}d\ ; \\
& \mathbf{none}
\end{array}$$

In our scheme, all participants must be aware of *all* (public) states that may appear on the shared state to enable safe re-splittings later on (even after subtyping). Therefore, each party

```
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 16: Seller code.

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 18, 16, and 17 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 17: 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 18: Buyer code.

E Complete Technical Development

$x \in$ Variables $\quad$ $\mathtt{t} \in$ Tags $\quad$ $\mathtt{f} \in$ Fields $\quad$ $\rho \in$ Location Constants

$l \in$ Location Variables $\quad$ $X \in$ Type Variables

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

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 selection)
	|	$v\ v$	(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 19: Values (v), expressions (e), evaluation contexts ($\mathcal{E}$).

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

Pairs, **rec**ursion, and other constructs are definable in the language as was done in [22].

Figure 20: Types (A).

$\boxed{H_0 \,;\, T_0 \;\mapsto\; H_1 \,;\, T_1}$ **Dynamics, (D:*)**

$$\text{(D:New)}\ \frac{\rho \textbf{ fresh}}{H \,;\, \mathcal{E}[\mathsf{new}\ v] \;\mapsto\; H, \rho \hookrightarrow v \,;\, \mathcal{E}[\rho]}$$

$$\text{(D:Delete)}\ \frac{}{H, \rho^{?} \hookrightarrow v \,;\, \mathcal{E}[\mathsf{delete}\ \rho] \;\mapsto\; H \,;\, \mathcal{E}[v]}$$

$$\text{(D:Dereference)}\ \frac{}{H, \rho^{?} \hookrightarrow v \,;\, \mathcal{E}[!\rho] \;\mapsto\; H, \rho^{?} \hookrightarrow v \,;\, \mathcal{E}[v]}$$

$$\text{(D:Assign)}\ \frac{}{H, \rho^{?} \hookrightarrow v_0 \,;\, \mathcal{E}[\rho := v_1] \;\mapsto\; H, \rho^{?} \hookrightarrow v_1 \,;\, \mathcal{E}[v_0]}$$

$$\text{(D:Application)}\ \frac{}{H \,;\, \mathcal{E}[(\lambda x.e)\ v] \;\mapsto\; H \,;\, \mathcal{E}[e\{v/x\}]}$$

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

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

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

$$\text{(D:Lock)}\ \frac{}{H, \overline{\rho \hookrightarrow v} \,;\, \mathcal{E}[\mathsf{lock}\ \overline{\rho}] \;\mapsto\; H, \overline{\rho^{\bullet} \hookrightarrow v} \,;\, \mathcal{E}[\{\}]}$$

$$\text{(D:Unlock)}\ \frac{}{H, \overline{\rho^{\bullet} \hookrightarrow v} \,;\, \mathcal{E}[\mathsf{unlock}\ \overline{\rho}] \;\mapsto\; H, \overline{\rho \hookrightarrow v} \,;\, \mathcal{E}[\{\}]}$$

$$\text{(D:Fork)}\ \frac{}{H \,;\, \mathcal{E}[\mathsf{fork}\ e] \;\mapsto\; H \,;\, \mathcal{E}[\{\}] \cdot e}$$

$$\text{(D:Thread)}\ \frac{H_0 \,;\, \mathcal{E}[e_0] \;\mapsto\; H_1 \,;\, \mathcal{E}[e_1] \cdot T_1}{H_0 \,;\, \mathcal{E}[e_0] \cdot\ T_0 \;\mapsto\; H_1 \,;\, \mathcal{E}[e_1] \cdot T_1 \cdot\ T_0}$$

Notes: $\rho^{?}$ ranges over ρ (not locked) and $\rho^{\bullet}$ (locked) with lock token unchanged. We use $\mathtt{f}_i$ to denote access to some position, i, in the set of labels $\overline{\mathtt{f}}$ (and similarly for other sets, such as $\overline{v}$, $\overline{\mathtt{t}}$, etc.).

Figure 21: Operational semantics.

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

$$\frac{}{\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\, \rho \dashv \cdot}\ (\text{T:Ref}) \qquad \frac{\Gamma \mid \cdot \vdash v : A \dashv \cdot}{\Gamma \mid \cdot \vdash v : !A \dashv \cdot}\ (\text{T:Pure}) \qquad \frac{}{\Gamma \mid \cdot \vdash v : ![] \dashv \cdot}\ (\text{T:Unit}) \qquad \frac{\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1}\ (\text{T:Pure-Elim})$$

$$\frac{}{\Gamma, x : A \mid \cdot \vdash x : !A \dashv \cdot}\ (\text{T:Pure-Read}) \qquad \frac{}{\Gamma \mid x : A \vdash x : A \dashv \cdot}\ (\text{T:Linear-Read}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v : [\overline{\mathtt{f} : A}] \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash v.\mathtt{f}_i : A_i \dashv \Delta_1}\ (\text{T:Selection}) \qquad \frac{\overline{\Gamma \mid \Delta \vdash v : A \dashv \cdot}}{\Gamma \mid \Delta \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \cdot}\ (\text{T:Record})$$

$$\frac{\Gamma \mid \Delta_0 \vdash v : \exists l.((!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)) \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v : \exists l.A \dashv \Delta_1}\ (\text{T:Delete}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v : \exists l.((!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)) \dashv \Delta_1}\ (\text{T:New})$$

$$\frac{\Gamma \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \quad \Gamma \mid \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1}{\Gamma \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_0}\ (\text{T:Assign}) \qquad \frac{\Gamma \vdash \Delta_0 <: \Delta_1 \quad \Gamma \mid \Delta_1 \vdash e : A_0 \dashv \Delta_2 \quad \Gamma \vdash A_0 <: A_1 \quad \Gamma \vdash \Delta_2 <: \Delta_3}{\Gamma \mid \Delta_0 \vdash e : A_1 \dashv \Delta_3}\ (\text{T:Subsumption})$$

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

$$\frac{\Gamma \mid \Delta, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta \vdash \lambda x.e : A_0 \multimap A_1 \dashv \cdot}\ (\text{T:Function}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1 \quad \Gamma \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2}\ (\text{T:Application})$$

$$\frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A}{\Gamma \mid \Delta_0 \vdash !v : A \dashv \Delta_1, \mathbf{rw}\ p\ ![]}\ (\text{T:Dereference-Linear}) \qquad \frac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ !A}{\Gamma \mid \Delta_0 \vdash !v : !A \dashv \Delta_1, \mathbf{rw}\ p\ !A}\ (\text{T:Dereference-Pure})$$

$$\frac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \quad \Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_2}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \& A_2}\ (\text{T:Intersection-Right}) \qquad \frac{\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1 \quad \Gamma \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1}\ (\text{T:Alternative-Left})$$

Figure 22: Typing rules (1/2).

(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. We use the notation $\mathtt{f}_i$ to denote access to some position, i, in the set of labels $\overline{\mathtt{f}}$ (and similarly for $\overline{A}$).

Figure 23: Typing rules (2/2).

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	::=	$\mathbf{type} \mid \mathbf{type} \to k \mid \mathbf{loc} \to k$	(kinds)

Figure 24: Typing environments.

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

$$\frac{}{\Gamma \vdash A <: A}\ (\textsc{st:Symmetry}) \qquad \frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash\ !A_0 <: A_1}\ (\textsc{st:ToLinear}) \qquad \frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash\ !A_0 <: !A_1}\ (\textsc{st:Pure}) \qquad \frac{}{\Gamma \vdash\ !A <: ![]}\ (\textsc{st:PureTop}) \qquad \frac{\Gamma_0 \vdash A <: B}{\Gamma_0, \Gamma_1 \vdash A <: B}\ (\textsc{st:Weakening})$$

$$\frac{\Gamma \vdash A_1 <: A_3 \qquad \Gamma \vdash A_2 <: A_0}{\Gamma \vdash A_0 \multimap A_1 <: A_2 \multimap A_3}\ (\textsc{st:Function}) \qquad \frac{\Gamma \vdash A_0 <: A_2}{\Gamma \vdash A_0 <: A_2 \oplus A_1}\ (\textsc{st:Alternative}) \qquad \frac{\Gamma \vdash A_0 <: A_2}{\Gamma \vdash A_0 \& A_1 <: A_2}\ (\textsc{st:Intersection})$$

$$\frac{}{\Gamma \vdash A <: \mathbf{top}}\ (\textsc{st:Top}) \qquad \frac{\overline{\Gamma \vdash A_i <: B_i} \qquad n \leq m}{\Gamma \vdash \sum_i^n \mathtt{t}_i \# A_i <: \sum_i^m \mathtt{t}_i \# B_i}\ (\textsc{st:Sum}) \qquad \frac{\Gamma \vdash [\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad \overline{\mathtt{f} : A} \neq \emptyset}{\Gamma \vdash [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}' : A'] <: [\overline{\mathtt{f} : B}]}\ (\textsc{st:Discard})$$

$$\frac{X <: A_0 \in \Gamma \qquad \Gamma \vdash A_0 <: A_1}{\Gamma \vdash X <: A_1}\ (\textsc{st:TypeVar}) \qquad \frac{\Gamma \vdash [\overline{\mathtt{f} : A}] <: [\overline{\mathtt{f} : B}] \qquad A' <: B'}{\Gamma \vdash [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}' : A'] <: [\overline{\mathtt{f} : B}\ ,\ \mathtt{f}' : B']}\ (\textsc{st:Record})$$

$$\frac{\Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_0 :: A_2 <: A_1 :: A_3}\ (\textsc{st:Stack}) \qquad \frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash \mathbf{rw}\ p\ A_0\ <:\ \mathbf{rw}\ p\ A_1}\ (\textsc{st:Cap}) \qquad \frac{\Gamma \vdash A_0 <: A_2 \qquad \Gamma \vdash A_1 <: A_3}{\Gamma \vdash A_0 * A_1 <: A_2 * A_3}\ (\textsc{st:Star})$$

$$\frac{\Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_0 \oplus A_2 <: A_1 \oplus A_3}\ (\textsc{st:Alternative-Cong}) \qquad \frac{\Gamma \vdash A_0 <: A_1 \qquad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_0 \& A_2 <: A_1 \& A_3}\ (\textsc{st:Intersection-Cong}) \qquad \frac{\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1}{\Gamma \vdash \forall l.A_0 <: \forall l.A_1}\ (\textsc{st:Loc-Forall})$$

$$\frac{\Gamma \vdash A_0 <: A_1}{\Gamma \vdash A_0 <: \exists l.A_1\{l/p\}}\ (\textsc{st:PackLoc}) \qquad \frac{\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1}{\Gamma \vdash \exists l.A_0 <: \exists l.A_1}\ (\textsc{st:Loc-Exists}) \qquad \frac{\Gamma, l : \mathbf{loc} \vdash A_0 <: A_1}{\Gamma \vdash \forall l.A_0 <: A_1\{p/l\}}\ (\textsc{st:LocApp})$$

$$\frac{\Gamma \vdash A_2 <: A_1 \qquad \Gamma \vdash A_0 <: A_0'}{\Gamma \vdash\ A_0\ <:\ \exists X <: A_1.A_0'\{X/A_2\}}\ (\textsc{st:PackType}) \qquad \frac{\Gamma \vdash A_2 <: A_1 \qquad \Gamma, X : \mathbf{type}, X <: A_1 \vdash A_0 <: A_0'}{\Gamma \vdash\ \forall X <: A_1.A_0\ <:\ A_0'\{A_2/X\}}\ (\textsc{st:TypeApp})$$

$$\frac{\Gamma, X : \mathbf{type}, X <: A_3 \vdash A_0 <: A_1}{\Gamma \vdash \exists X <: A_3.A_0 <: \exists X <: A_3.A_1}\ (\textsc{st:Type-Exists}) \qquad \frac{\Gamma, X : \mathbf{type}, X <: A_3 \vdash A_0 <: A_1}{\Gamma \vdash \forall X <: A_3.A_0 <: \forall X <: A_3.A_1}\ (\textsc{st:Type-Forall})$$

Note: recall that $\oplus$ and $\&$ are commutative; thus we only have one subtyping rule for (ST:ALTERNATIVE) and (ST:INTERSECTION).

Figure 25: Subtyping on types (co-inductive).

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

$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad A_0 <: A_1}{\Gamma \vdash \Delta_0, x : A_0 <: \Delta_1, x : A_1}\ (\textsc{sd:Var}) \qquad \frac{}{\Gamma \vdash \Delta <: \Delta}\ (\textsc{sd:Symmetry}) \qquad \frac{\Gamma \vdash \Delta_0 <: \Delta_1, A_0 \qquad \Gamma \vdash A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \vdash \Delta_0 <: \Delta_1, A_1, A_2}\ (\textsc{sd:Share})$$

$$\frac{\Gamma \vdash \Delta, A_0, A_1 <: \Delta, A_1, A_2}{\Gamma \vdash \Delta, A_0, A_1 <: \Delta, A_1 * A_2}\ (\textsc{sd:Star-R}) \qquad \frac{\Gamma \vdash \Delta, A_0 * A_1 <: \Delta, A_1 * A_2}{\Gamma \vdash \Delta, A_0 * A_1 <: \Delta, A_1, A_2}\ (\textsc{sd:Star-L}) \qquad \frac{\Gamma \vdash \Delta_0 <: \Delta_1 \qquad \Gamma \vdash A_0 <: A_1}{\Gamma \vdash \Delta_0, A_0 <: \Delta_1, A_1}\ (\textsc{sd:Type})$$

$$\frac{\Gamma \vdash \Delta_0 <: \Delta_1}{\Gamma \vdash \Delta_0 <: \Delta_1, \mathbf{none}}\ (\textsc{sd:None-R}) \qquad \frac{\Gamma \vdash \Delta_0 <: \Delta_1}{\Gamma \vdash \Delta_0, \mathbf{none} <: \Delta_1}\ (\textsc{sd:None-L}) \qquad \frac{\Gamma \vdash \Delta_0, A_0 <: \Delta_1 \qquad \Gamma \vdash \Delta_0, A_1 <: \Delta_1}{\Gamma \vdash \Delta_0, A_0 \oplus A_1 <: \Delta_1}\ (\textsc{sd:Alternative-L}) \qquad \frac{\Gamma \vdash \Delta_0 <: \Delta_1, A_1 \qquad \Gamma \vdash \Delta_0 <: \Delta_1, A_2}{\Gamma \vdash \Delta_0 <: \Delta_1, A_1 \& A_2}\ (\textsc{sd:Intersection-R})$$

Figure 26: Subtyping on linear typing environments.

E.1 Protocol Composition

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

$$
\frac{\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle\!\uparrow}{\Gamma \vdash R \Rrightarrow P \parallel Q}\ (\textsc{wf:Split}) \qquad \frac{C_0 \mapsto C_1 \qquad C_1\!\uparrow}{C_0\!\uparrow}\ (\textsc{wf:Configuration})
$$

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

Figure 27: Grammar restrictions for checking composition: *P*rotocols, *S*tates, and *R*esources (either a protocol or a state).

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

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

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

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

$$\frac{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \qquad \langle\, \Gamma \vdash R_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_1}{\langle\, \Gamma \vdash R_0 \oplus R_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C_0 \cdot C_1}\ (\textsc{C-RS:StateAlternative}) \qquad \frac{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0 \oplus P_1] \,\rangle \mapsto C}\ (\textsc{C-RS:ProtocolAlternative})$$

$$\frac{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R_0 \& R_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}\ (\textsc{C-RS:StateIntersection}) \qquad \frac{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C_0 \qquad \langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C_1}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0 \& P_1] \,\rangle \mapsto C_0 \cdot C_1}\ (\textsc{C-RS:ProtocolIntersection})$$

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

$$\frac{}{\langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[S_0 \Rightarrow S_1; P] \,\rangle \mapsto \langle\, \Gamma \vdash S_1 \Rrightarrow \mathcal{R}[P] \,\rangle}\ (\textsc{C-SS:Step})$$

$$\frac{}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[S] \,\rangle \mapsto \langle\, \Gamma \vdash \textsf{none} \Rrightarrow \mathcal{R}[\textsf{none}] \,\rangle}\ (\textsc{C-SS:Recovery})$$

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

$$\frac{}{\langle\, \Gamma \vdash S_0 \Rightarrow S_1; Q \Rrightarrow \mathcal{R}[S_0 \Rightarrow S_1; P] \,\rangle \mapsto \langle\, \Gamma \vdash Q \Rrightarrow \mathcal{R}[P] \,\rangle}\ (\textsc{C-PS:Step})$$

Figure 28: Basic protocol composition stepping rules.

$$\text{(c-rs:Weakening)}\quad \frac{\langle\, \Gamma_0 \vdash R \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, \Gamma_0, \Gamma_1 \vdash R \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C} \qquad \text{(c-ss:ForallLoc)}\quad \frac{\langle\, \Gamma, l : \mathbf{loc} \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow \forall l.P] \,\rangle \mapsto C}$$

$$\text{(c-ss:OpenLoc)}\quad \frac{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P\{p/l\}] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[\exists l.P] \,\rangle \mapsto C} \qquad \text{(c-ss:ForallType)}\quad \frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow \forall X <: A.P] \,\rangle \mapsto C}$$

$$\text{(c-ss:OpenType)}\quad \frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[P\{A_1/X\}] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rrightarrow \mathcal{R}[\exists X <: A_0.P] \,\rangle \mapsto C}$$

$$\text{(c-ps:ExistsType)}\quad \frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash P \Rrightarrow \mathcal{R}[Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash \exists X <: A.P \Rrightarrow \mathcal{R}[\exists X <: A.Q] \,\rangle \mapsto C} \qquad \text{(c-ps:ExistsLoc)}\quad \frac{\langle\, \Gamma, l : \mathbf{loc} \vdash P \Rrightarrow \mathcal{R}[Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash \exists l.P \Rrightarrow \mathcal{R}[\exists l.Q] \,\rangle \mapsto C}$$

$$\text{(c-ps:ForallType)}\quad \frac{\langle\, \Gamma, X : \mathbf{type}, X <: A \vdash S \Rightarrow P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall X <: A.P \Rrightarrow \mathcal{R}[S \Rightarrow \forall X <: A.Q] \,\rangle \mapsto C}$$

$$\text{(c-ps:ForallLoc)}\quad \frac{\langle\, \Gamma, l : \mathbf{loc} \vdash S \Rightarrow P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[S \Rightarrow \forall l.Q] \,\rangle \mapsto C}$$

$$\text{(c-ps:LocApp)}\quad \frac{\langle\, \Gamma \vdash S \Rightarrow P\{p/l\} \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall l.P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}$$

$$\text{(c-ps:TypeApp)}\quad \frac{\Gamma \vdash A_1 <: A_0 \qquad \langle\, \Gamma \vdash S \Rightarrow P\{A_1/X\} \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}{\langle\, \Gamma \vdash S \Rightarrow \forall X <: A_0.P \Rrightarrow \mathcal{R}[S \Rightarrow Q] \,\rangle \mapsto C}$$

$P\{A/X\} \triangleq$ "substitution, in P, of X for A"

Note: bound type/location variables of a type must be fresh that rule's conclusion.

Figure 29: Protocol composition abstraction extension.

(C-RS:SUBSUMPTION)

$$\frac{\Gamma \vdash R_1 <: R_0 \qquad \langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C \qquad \Gamma \vdash P_0 <: P_1}{\langle\, \Gamma \vdash R_1 \Rrightarrow \mathcal{R}[P_1] \,\rangle \mapsto C}$$

(C-SS:RECOVERY)

$$\frac{\Gamma \vdash S_0 <: S_1}{\langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[S_1] \,\rangle \mapsto \langle\, \Gamma \vdash \textbf{none} \Rrightarrow \mathcal{R}[\textbf{none}] \,\rangle}$$

(C-SS:STEP)

$$\frac{\Gamma \vdash S_0 <: S_1}{\langle\, \Gamma \vdash S_0 \Rrightarrow \mathcal{R}[S_1 \Rightarrow S_2; P] \,\rangle \mapsto \langle\, \Gamma \vdash S_2 \Rrightarrow \mathcal{R}[P] \,\rangle}$$

(C-PS:STEP)

$$\frac{\Gamma \vdash S_0 <: S_1 \qquad \Gamma \vdash S_3 <: S_2}{\langle\, \Gamma \vdash S_0 \Rightarrow S_2; Q \Rrightarrow \mathcal{R}[S_1 \Rightarrow S_3; P] \,\rangle \mapsto \langle\, \Gamma \vdash Q \Rrightarrow \mathcal{R}[P] \,\rangle}$$

Figure 30: Protocol composition subtyping extension.

F Algorithms

F.1 Protocol Composition

$\boxed{\mathtt{c}(\Gamma,\, R,\, P,\, Q)}$ **Composition Algorithm, (c)**

(1) $\mathtt{c}(\Gamma,\, R,\, P,\, Q) \triangleq \mathtt{cf}(\Gamma,\, R,\, P,\, Q,\, \emptyset)$

(2) $\mathtt{c}(\Gamma,\, R,\, P,\, Q,\, v) \triangleq$ /* (c:Step) and (c:AllStep) */
$\forall(\Gamma' \vdash R' \Rrightarrow P' \parallel Q') \in (\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P],\, v) \cup \mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[Q],\, v)).(\, \mathtt{c}(\Gamma',\, R',\, P',\, Q',\, v \cup \langle \Gamma \vdash R \Rrightarrow P \parallel Q \rangle)\,)$

$\boxed{\mathtt{stp}(\Gamma,\, R,\, P,\, v)}$ **Step, (stp)**

(3) $\mathtt{stp}((\Gamma', \Gamma),\, R,\, \mathcal{R}[P],\, v) \triangleq$ /* (c-rs:Weakening) */
$\emptyset$ **if** $\langle \Gamma \vdash R \Rrightarrow \mathcal{R}[P] \rangle \in v$ /* considering equality up to (eq:Rec) */

(4) $\mathtt{stp}(\Gamma,\, (\mathbf{rec}\ X(\overline{u}).R)[\overline{U}],\, \mathcal{R}[P],\, v) \triangleq \mathtt{stp}(\Gamma,\, R\{\mathbf{rec}\ X(\overline{u}).R/X\}\{\overline{U}/\overline{u}\},\, \mathcal{R}[P],\, v)$ /* (eq:Rec) */

(5) $\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[(\mathbf{rec}\ X(\overline{u}).P)[\overline{U}]],\, v) \triangleq \mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P\{\mathbf{rec}\ X(\overline{u}).P/X\}\{\overline{U}/\overline{u}\}],\, v)$ /* (eq:Rec) */

(6) $\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[\mathbf{none}],\, v) \triangleq \{\Gamma \vdash R \Rrightarrow \mathcal{R}[\mathbf{none}]\}$ /* (c-rs:None) */

(7) $\mathtt{stp}(\Gamma,\, S_0,\, \mathcal{R}[S_0 \Rightarrow S_1; P],\, v) \triangleq \{\Gamma \vdash S_1 \Rrightarrow \mathcal{R}[P]\}$ /* (c-ss:Step) */

(8) $\mathtt{stp}(\Gamma,\, (S_0 \Rightarrow S_1; Q),\, \mathcal{R}[S_0 \Rightarrow S_1; P],\, v) \triangleq \{\Gamma \vdash Q \Rrightarrow \mathcal{R}[P]\}$ /* (c-ps:Step) */

(9) $\mathtt{stp}(\Gamma,\, R_0 \oplus R_1,\, \mathcal{R}[P],\, v) \triangleq \mathtt{stp}(\Gamma,\, R_0,\, \mathcal{R}[P],\, v) \cup \mathtt{stp}(\Gamma,\, R_1,\, \mathcal{R}[P],\, v)$ /* (c-rs:StateAlternative) */

(10) $\mathtt{stp}(\Gamma,\, R_0 \& R_1,\, \mathcal{R}[P],\, v) \triangleq \mathtt{stp}(\Gamma,\, R_0,\, \mathcal{R}[P],\, v)$ /* (c-rs:StateIntersection) */
(11) $\mathtt{stp}(\Gamma,\, R_0 \& R_1,\, \mathcal{R}[P],\, v) \triangleq \mathtt{stp}(\Gamma,\, R_1,\, \mathcal{R}[P],\, v)$

(12) $\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P_0 \oplus P_1],\, v) \triangleq \mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P_0],\, v)$ /* (c-rs:ProtocolAlternative) */
(13) $\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P_0 \oplus P_1],\, v) \triangleq \mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P_1],\, v)$

(14) $\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P_0 \& P_1],\, v) \triangleq \mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P_0],\, v) \cup \mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P_1],\, v)$ /* (c-rs:ProtocolIntersection) */

(15) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[S\,],\, v) \triangleq \{\Gamma \vdash \mathbf{none} \Rrightarrow \mathcal{R}[\mathbf{none}]\}$ /* (c-ss:Recovery) */

(16) $\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[\exists l.P],\, v) \triangleq \mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P\{p/l\}],\, v)$ /* (c-ss:OpenLoc) */

(17) $\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[\exists X <: A_0.P],\, v) \triangleq$ /* (c-ss:OpenType) */
$\mathtt{stp}(\Gamma,\, R,\, \mathcal{R}[P\{A_1/X\}],\, v)$ **if** $\mathtt{sbt}(\Gamma,\, A_1,\, A_0)$

(18) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[S \Rightarrow \forall l.P],\, v) \triangleq \mathtt{stp}((\Gamma, l : \mathbf{loc}),\, S,\, \mathcal{R}[S \Rightarrow P],\, v)$ /* (c-ss:ForallLoc) */

(19) $\mathtt{stp}(\Gamma,\, S,\, \mathcal{R}[S \Rightarrow \forall X <: A.P],\, v) \triangleq \mathtt{stp}((\Gamma, X : \mathbf{type}, X <: A),\, S,\, \mathcal{R}[S \Rightarrow P],\, v)$ /* (c-ss:ForalType) */

Figure 31: Protocol composition algorithm (1/2).

(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) */
$\texttt{stp}((\Gamma, X : \textbf{type}, X <: A),\ P,\ \mathcal{R}[Q],\ v)$

(22) $\texttt{stp}(\Gamma,\ S \Rightarrow \forall l.P,\ \mathcal{R}[S \Rightarrow \forall l.Q],\ v) \triangleq$ /* (C-PS:FORALLLOC) */
$\texttt{stp}((\Gamma, l : \textbf{loc}),\ S \Rightarrow P,\ \mathcal{R}[S \Rightarrow Q],\ v)$

(23) $\texttt{stp}(\Gamma,\ S \Rightarrow \forall X <: A.P,\ \mathcal{R}[S \Rightarrow \forall X <: A.Q],\ v) \triangleq$ /* (C-PS:FORALLTYPE) */
$\texttt{stp}((\Gamma, X : \textbf{type}, X <: A),\ S \Rightarrow P,\ \mathcal{R}[S \Rightarrow Q],\ v)$

(24) $\texttt{stp}(\Gamma,\ S \Rightarrow \forall l.P,\ \mathcal{R}[S \Rightarrow Q],\ v) \triangleq \texttt{stp}(\Gamma,\ S \Rightarrow P\{l/p\},\ \mathcal{R}[S \Rightarrow Q],\ v)$ /* (C-PS:LOCAPP) */

(25) $\texttt{stp}(\Gamma,\ S \Rightarrow \forall X <: A_0.P,\ \mathcal{R}[S \Rightarrow Q],\ v) \triangleq$ /* (C-PS:TYPEAPP) */
$\texttt{stp}(\Gamma,\ S \Rightarrow P\{A_1/X\},\ \mathcal{R}[S \Rightarrow Q],\ v)\ \textbf{if}\ \texttt{sbt}(\Gamma,\ A_1,\ A_0)$

Subtyping extension:

(7) $\texttt{stp}(\Gamma,\ S_0,\ \mathcal{R}[S_1 \Rightarrow S_2; P],\ v) \triangleq \{\Gamma \vdash S_2 \Rrightarrow \mathcal{R}[P]\}\ \textbf{if}\ \texttt{sbt}(\Gamma,\ S_0,\ S_1)$ /* (C-SS:STEP) */

(8) $\texttt{stp}(\Gamma,\ (S_0 \Rightarrow S_2; Q),\ \mathcal{R}[S_1 \Rightarrow S_3; P],\ v) \triangleq \{\Gamma \vdash Q \Rrightarrow \mathcal{R}[P]\}\ \textbf{if}\ \texttt{sbt}(\Gamma,\ S_0,\ S_1) \wedge \texttt{sbt}(\Gamma,\ S_3,\ S_2)$ /* (C-PS:STEP) */

(15) $\texttt{stp}(\Gamma,\ S_0,\ \mathcal{R}[S_1],\ v) \triangleq \{\Gamma \vdash \textbf{none} \Rrightarrow \mathcal{R}[\textbf{none}]\}\ \textbf{if}\ \texttt{sbt}(\Gamma,\ S_0,\ S_1)$ /* (C-SS:RECOVERY) */

Note: Recall that (C-RS:SUBSUMPTION) is admissible.

Figure 32: Protocol composition algorithm (2/2).

F.2 Subtyping

Our subtyping algorithm follows the approach of [32, 2] so that `sbt` includes a *t*rail to track cycles and close the co-inductive proof.

$\boxed{\texttt{sbt}(\ \Gamma,\ A,\ B\ )}$ (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) $\texttt{sbt}(\ \Gamma,\ (A * B),\ (C * D),\ t\ ) \triangleq$ /* (ST:STAR) */
$(\texttt{sbt}(\ \Gamma,\ A,\ C,\ t\ ) \wedge \texttt{sbt}(\ \Gamma,\ B,\ D,\ t\ )) \vee (\texttt{sbt}(\ \Gamma,\ A,\ D,\ t\ ) \wedge \texttt{sbt}(\ \Gamma,\ B,\ C,\ t\ ))$

(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) */

(18) $\texttt{sbt}(\ \Gamma,\ (\textbf{rec}\ X(\overline{u}).A)[\overline{U}],\ (\textbf{rec}\ Y(\overline{u'}).B)[\overline{U'}],\ t\ ) \triangleq$ /* (EQ:REC) */
$\texttt{sbt}(\ \Gamma,\ A\{\textbf{rec}\ X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\},\ B\{\textbf{rec}\ Y(\overline{u'}).B/Y\}\{\overline{U'}/\overline{u'}\},\ (t \cup (\Gamma \vdash (\textbf{rec}\ X(\overline{u}).A)[\overline{U}] <: (\textbf{rec}\ Y(\overline{u'}).B)[\overline{U'}]))\ )$

(19) $\texttt{sbt}(\ \Gamma,\ (\textbf{rec}\ X(\overline{u}).A)[\overline{U}],\ B,\ t\ ) \triangleq$ /* (EQ:REC) */
$\texttt{sbt}(\ \Gamma,\ A\{\textbf{rec}\ X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\},\ B,\ (t \cup (\Gamma \vdash (\textbf{rec}\ X(\overline{u}).A)[\overline{U}] <: B))\ )$

(20) $\texttt{sbt}(\ \Gamma,\ A,\ (\textbf{rec}\ Y(\overline{u'}).B)[\overline{U'}],\ t\ ) \triangleq$ /* (EQ:REC) */
$\texttt{sbt}(\ \Gamma,\ A,\ B\{\textbf{rec}\ Y(\overline{u'}).B/Y\}\{\overline{U'}/\overline{u'}\},\ (t \cup (\Gamma \vdash A <: (\textbf{rec}\ Y(\overline{u'}).B)[\overline{U'}]))\ )$

Figure 33: Subtyping algorithm (1/2).

(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) $\mathtt{sbt}(\,\Gamma,\,A \oplus B,\,C \oplus D,\,t\,) \triangleq$ /* (ST:ALTERNATIVE-CONG) */
$(\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,)) \vee (\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,))$

(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) $\mathtt{sbt}(\,\Gamma,\,A\&B,\,C\&D,\,t\,) \triangleq$ /* (ST:INTERSECTION-CONG) */
$(\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,)) \vee (\mathtt{sbt}(\,\Gamma,\,A,\,C,\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,B,\,D,\,t\,))$

(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) $\mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A},\,\mathtt{f}' : A'],\,[\overline{\mathtt{f} : B}],\,t\,) \triangleq$ /* (ST:DISCARD) */
$[\overline{\mathtt{f} : A}] \neq \emptyset \wedge \mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A}],\,[\overline{\mathtt{f} : B}],\,t\,)$

(27) $\mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A},\,\mathtt{f}' : A'],\,[\overline{\mathtt{f} : B},\,\mathtt{f}' : B'],\,t\,) \triangleq$ /* (ST:RECORD) */
$\mathtt{sbt}(\,\Gamma,\,A',\,B',\,t\,) \wedge \mathtt{sbt}(\,\Gamma,\,[\overline{\mathtt{f} : A}],\,[\overline{\mathtt{f} : B}],\,t\,)$

(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 34: Subtyping algorithm (2/2).

G Auxiliary Definitions

G.1 Well-Formed Types and Environments

Well-formed conditions are not explicitly mentioned, but are assumed to be present whenever they are relevant.

Definition 1 (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)**

$$\frac{}{\Gamma \vdash \mathbf{none}\ \mathbf{type}} \qquad \frac{\Gamma \vdash A\ \mathbf{type}}{\Gamma \vdash\ !A\ \mathbf{type}} \qquad \frac{\overline{\Gamma \vdash A_i\ \mathbf{type}}}{\Gamma \vdash [\overline{\mathtt{f} : A}]\ \mathbf{type}} \qquad \frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash A_0 \multimap A_1\ \mathbf{type}}$$

$$\frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash\ A_0 :: A_1\ \mathbf{type}} \qquad \frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash\ A_0 * A_1\ \mathbf{type}}$$

$$\frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash A_0 \oplus A_1\ \mathbf{type}} \qquad \frac{\Gamma \vdash A_0\ \mathbf{type} \qquad \Gamma \vdash A_1\ \mathbf{type}}{\Gamma \vdash A_0 \& A_1\ \mathbf{type}}$$

$$\frac{\Gamma \vdash p\ \mathbf{loc} \qquad \Gamma \vdash A\ \mathbf{type}}{\Gamma \vdash\ \mathbf{rw}\ p\ A\ \mathbf{type}} \qquad \frac{\Gamma \vdash p\ \mathbf{loc}}{\Gamma \vdash\ \mathbf{ref}\ p\ \mathbf{type}} \qquad \frac{\Gamma, l : \mathbf{loc} \vdash A\ \mathbf{type}}{\Gamma \vdash \forall l.A\ \mathbf{type}} \qquad \frac{\Gamma, l : \mathbf{loc} \vdash A\ \mathbf{type}}{\Gamma \vdash \exists l.A\ \mathbf{type}}$$

$$\frac{\cdot \vdash A_0 \ \textbf{type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1 \ \textbf{type}}{\Gamma \vdash \forall X <: A_0.A_1 \ \textbf{type}}$$

$$\frac{\cdot \vdash A_0 \ \textbf{type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1 \ \textbf{type}}{\Gamma \vdash \exists X <: A_0.A_1 \ \textbf{type}}$$

$$\frac{\overline{\Gamma \vdash A_i \ \textbf{type}}}{\Gamma \vdash \sum_i \mathtt{t}_i \# A_i \ \textbf{type}} \qquad \frac{\Gamma \vdash A_0 \ \textbf{type} \qquad \Gamma \vdash A_1 \ \textbf{type}}{\Gamma \vdash A_0; A_1 \ \textbf{type}}$$

$$\frac{\Gamma \vdash A_0 \ \textbf{type} \qquad \Gamma \vdash A_1 \ \textbf{type} \qquad \textbf{locs}(A_0) = \textbf{locs}(A_1) \neq \emptyset}{\Gamma \vdash A_0 \Rightarrow A_1 \ \textbf{type}}$$

$$\frac{\begin{array}{c} u_0 : k_0, ..., u_n : k_n, X : k_0 \to ... \to k_n \to \textbf{type} \vdash A \ \textbf{type} \\ \Gamma \vdash U_i \ k_i \qquad k_i = \mathtt{kind}(u_i) \qquad i \in \{0, ..., n\} \end{array}}{\Gamma \vdash (\textbf{rec} \ X(\overline{u}).A)[\overline{U}] \ \textbf{type}}$$

$$\frac{(X : k_0 \to ... \to k_n \to \textbf{type}) \in \Gamma \qquad (U_i : k_i) \in \Gamma \qquad i \in \{0, ..., n\}}{\Gamma \vdash X[\overline{U}] \ \textbf{type}}$$

where:

$$\begin{array}{lcl} \mathtt{kind}(l) & = & \textbf{loc} \\ \mathtt{kind}(X) & = & \textbf{type} \end{array}$$

G.2 Set of Locations of a Type

Definition 2 (Locations of a Type)**.**

$$\begin{array}{lcl}
\textbf{locs}(\textbf{rw} \ p \ A) & = & \{p\} \\
\textbf{locs}(A_0 * A_1) & = & \textbf{locs}(A_0) \cup \textbf{locs}(A_1) \\
\textbf{locs}(A_0 \& A_1) & = & \textbf{locs}(A_0) \cup \textbf{locs}(A_1) \\
\textbf{locs}(A_0 \oplus A_1) & = & \textbf{locs}(A_0) \cup \textbf{locs}(A_1) \\
\textbf{locs}(\exists l.A) & = & \textbf{locs}(A) \\
\textbf{locs}(\forall l.A) & = & \textbf{locs}(A) \\
\textbf{locs}(\exists X <: A_0.A_1) & = & \textbf{locs}(A_1\{A_0/X\}) \\
\textbf{locs}(\forall X <: A_0.A_1) & = & \textbf{locs}(A_1\{A_0/X\}) \\
\textbf{locs}(A_0 \Rightarrow A_1) & = & \textbf{locs}(A_0) \\
\textbf{locs}(A_0; A_1) & = & \textbf{locs}(A_0) \\
\textbf{locs}(\textsf{none}) & = & \emptyset \\
\textbf{locs}(P) & = & \emptyset \\
\textbf{locs}(X[\overline{U}]) & = & \emptyset \\
\textbf{locs}((\textbf{rec} \ X(\overline{u}).A)[\overline{U}]) & = & \textbf{locs}(A\{\textbf{rec} \ X(\overline{u}).A/X\}\{\overline{U}/\overline{u}\})
\end{array}$$

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.

G.3 Store Typing

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

$$\frac{\Gamma \mid \Delta \vdash H}{\Gamma, \rho : \mathbf{loc} \mid \Delta \vdash H}\ (\text{STR:LOC}) \qquad \frac{\Gamma \mid \Delta_0 \vdash H \qquad \Gamma \vdash \Delta_0 <: \Delta_1}{\Gamma \mid \Delta_1 \vdash H}\ (\text{STR:SUBSUMPTION}) \qquad \frac{\Gamma \mid \Delta, \Delta_v \vdash H \qquad \Gamma \mid \Delta_v \vdash v : A \dashv \cdot}{\Gamma \mid \Delta, \mathbf{rw}\ \rho\ A \vdash H, \rho \hookrightarrow v}\ (\text{STR:BINDING})$$

$$\frac{}{\cdot \mid \cdot \vdash \cdot}\ (\text{STR:EMPTY}) \qquad \frac{\mathbf{locs}(A_0) = \mathbf{locs}(A_1) = \overline{\rho} \qquad \Gamma \mid A_0 \vdash H', \overline{\rho \hookrightarrow v} \qquad \Gamma \mid A_1 \vdash H'', \overline{\rho \hookrightarrow v'} \qquad \Gamma \mid \Delta, A_2 \vdash H, H'', \overline{\rho \hookrightarrow v'}}{\Gamma \mid \Delta, A_0,\ A_1; A_2\ \vdash H, H', \overline{\rho^\bullet \hookrightarrow v}}\ (\text{STR:LOCKED})$$

$$\frac{\overline{\rho \hookrightarrow v} \notin H \qquad \mathbf{locs}(A_1) = \overline{\rho} \qquad \Gamma \mid A_1 \vdash H'', \overline{\rho \hookrightarrow v'} \qquad \Gamma \mid \Delta, A_2 \vdash H, H'', \overline{\rho \hookrightarrow v'}}{\Gamma \mid \Delta,\ A_1; A_2\ \vdash H}\ (\text{STR:DEAD-LOCKED})$$

$$\frac{\Gamma, l : \mathbf{loc} \mid \Delta, A \vdash H}{\Gamma \mid \Delta, \forall l.A \vdash H}\ (\text{STR:FORALLLOCS}) \qquad \frac{\Gamma, X : \mathbf{type}, X <: A_0 \mid \Delta, A_1 \vdash H}{\Gamma \mid \Delta, \forall X <: A_0.A_1 \vdash H}\ (\text{STR:FORALLTYPES})$$

(where l, X are fresh in the conclusion)

G.4 Substitution

Definition 3 (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:

$\boxed{e_0\{v/x\} = e_1}$

(vs:1)	$\rho\{v/x\}$	$=$	ρ	
(vs:2)	$x\{v/x\}$	$=$	v	
(vs:3)	$x_0\{v/x_1\}$	$=$	x_0	$(x_0 \neq x_1)$
(vs:4)	$(\lambda x_0.e_0)\{v/x_1\}$	$=$	$\lambda x_0.e_0\{v/x_1\}$	$(x_0 \neq x_1)$
(vs:5)	$\{\overline{\mathtt{f} = e}\}\{v/x\}$	$=$	$\{\overline{\mathtt{f} = e\{v/x\}}\}$	
(vs:6)	$(e.\mathtt{f})\{v/x\}$	$=$	$e\{v/x\}.\mathtt{f}$	
(vs:7)	$(e_0\ e_1)\{v/x\}$	$=$	$e_0\{v/x\}\ e_1\{v/x\}$	
(vs:8)	$(\mathsf{new}\ e)\{v/x\}$	$=$	$\mathsf{new}\ e\{v/x\}$	
(vs:9)	$(\mathsf{delete}\ e)\{v/x\}$	$=$	$\mathsf{delete}\ e\{v/x\}$	
(vs:10)	$(!e)\{v/x\}$	$=$	$!e\{v/x\}$	
(vs:11)	$(e_0 := e_1)\{v/x\}$	$=$	$e_0\{v/x\} := e_1\{v/x\}$	
(vs:12)	$(\mathtt{t}\#e)\{v/x\}$	$=$	$\mathtt{t}\#e\{v/x\}$	
(vs:13)	$(\mathsf{case}\ e\ \mathsf{of}\ \overline{\mathtt{t}_i\#x_i \to e_i}\ \mathsf{end})\{v/x\}$	$=$	$\mathsf{case}\ e\{v/x\}\ \mathsf{of}\ \overline{\mathtt{t}_i\#x_i \to e_i\{v/x\}}\ \mathsf{end}$	$(x_i \neq x)$
(vs:14)	$(\mathsf{let}\ x_0 = e_0\ \mathsf{in}\ e_1\ \mathsf{end})\{v/x_1\}$	$=$	$\mathsf{let}\ x_0 = e_0\{v/x_1\}\ \mathsf{in}\ e_1\{v/x_1\}\ \mathsf{end}$	$(x_0 \neq x_1)$
(vs:15)	$(\mathsf{lock}\ \overline{e})\{v/x\}$	$=$	$\mathsf{lock}\ \overline{e\{v/x\}}$	
(vs:16)	$(\mathsf{unlock}\ \overline{e})\{v/x\}$	$=$	$\mathsf{unlock}\ \overline{e\{v/x\}}$	
(vs:17)	$(\mathsf{fork}\ e)\{v/x\}$	$=$	$\mathsf{fork}\ e\{v/x\}$	

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}
$$

$\boxed{\Gamma_0\{A/X\} = \Gamma_1}$

$$
\begin{array}{lrcll}
(\text{TS:3.1}) & \cdot\{A/X\} & = & \cdot & \\
(\text{TS:3.2}) & (\Gamma, x : A_0)\{A_1/X\} & = & \Gamma\{A_1/X\}, x : A_0\{A_1/X\} & \\
(\text{TS:3.3}) & (\Gamma, l : \mathbf{loc})\{A/X\} & = & \Gamma\{A/X\}, l : \mathbf{loc} & \\
(\text{TS:3.4}) & (\Gamma, X_0 <: A_0)\{A_1/X_1\} & = & \Gamma\{A_1/X_1\}, X_0 <: A_0\{A_1/X_1\} & (X_0 \neq X_1) \\
(\text{TS:3.5}) & (\Gamma, X_0 : k)\{A_1/X_1\} & = & \Gamma\{A_1/X_1\}, X_0 : k & (X_0 \neq X_1)
\end{array}
$$

$\boxed{\Delta_0\{A/X\} = \Delta_1}$

$$
\begin{array}{lrcl}
(\text{TS:4.1}) & \cdot\{A/X\} & = & \cdot \\
(\text{TS:4.2}) & (\Delta, x : A_0)\{A_1/X\} & = & \Delta\{A_1/X\}, x : A_0\{A_1/X\} \\
(\text{TS:4.3}) & (\Delta, A_0)\{A_1/X\} & = & \Delta\{A_1/X\}, A_0\{A_1/X\}
\end{array}
$$

H Main Theorems

H.1 Subtyping Lemmas

Lemma 11 (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) $A = [\overline{\mathtt{f} : A}, \mathtt{f}'' : A'']$ 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) $A = [\overline{\mathtt{f} : A}, \mathtt{f}'' : A_0]$ and $A' = [\overline{\mathtt{f} : A}, \mathtt{f}'' : B_0]$ 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 $\Delta = \Delta_0, \textsf{none}$ then $\Delta' = \Delta_1$ and $\Gamma \vdash \Delta_0 <: \Delta_1$.
  7. $\Delta' = \Delta_0, \textsf{none}$ 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 12 (Subtyping 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 by induction on the derivation of subtyping. Note that transitivity for (SD:SHARE) requires the subtyping extension on composition. □

H.2 Store Typing Lemmas

Lemma 13 (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 14 (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.:

$$A_0 \in \mathtt{rely}(A_1) \qquad A_0 \in \mathtt{rely}(A_2)$$

where:

$$\begin{array}{lcll} \mathtt{rely}(\ P_0 \& P_1\ ) & = & \mathtt{rely}(P_0) & \text{if } \mathtt{rely}(P_0) = \mathtt{rely}(P_1) \\ \mathtt{rely}(\ P_0 \oplus P_1\ ) & = & \mathtt{rely}(P_0) \cup \mathtt{rely}(P_1) & \\ \mathtt{rely}(\ A \Rightarrow P\ ) & = & \{A\} & \end{array}$$

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. □

H.3 Values Inversion Lemma

Lemma 15 (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] $A_1 =$ **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)] $A_0 = [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}' : A']$ and
$A_1 = [\overline{\mathtt{f} : A}\ ,\ \mathtt{f}' : A'']$ (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)] $A_0 = [\overline{\mathtt{f} : A}, \mathtt{f}' : A]$ 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 $A_0 = \textbf{top}$ then $A_1 = \textbf{top}$ 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.

□

H.4 Free Variables Lemma

Lemma 16 (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. $x \in \texttt{fv}(v_0) \wedge x \in \texttt{fv}(v_1)$

   $x \notin \Delta_1$ (3.1)

   by induction hypothesis on (3) and case assumption.

   We reach a contradiction since v_0 is well-typed by (4) but $x \in \texttt{fv}(v_1)$ 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)
$x \in \texttt{fv}(!v)$ (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).
[$\texttt{fv}(!v) = \texttt{fv}(v)$]

$x \in \texttt{fv}(v)$ (4)

by definition of 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)
$x \in \texttt{fv}(!v)$ (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).
[$\texttt{fv}(!e) = \texttt{fv}(v)$]

$x \in \texttt{fv}(v)$ (4)

by definition of 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:

$\Gamma \mid \Delta, x : A_0 \vdash \{\overline{\texttt{f} = v}\} : [\overline{\texttt{f} : A}] \dashv \cdot$ (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. $x \in \mathtt{fv}(e_1) \wedge x \notin \mathtt{fv}(e_0)$

   $(x : A) \in \Delta_1$ (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. $x \in \mathtt{fv}(e_0) \wedge x \in \mathtt{fv}(e_1)$

   $x \notin \Delta_1$ (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. $x \in \mathtt{fv}(e_0) \wedge x \notin \mathtt{fv}(e_1)$

   $x \notin \Delta_1$ (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.

□

H.5 Well-Form Lemmas

Lemma 17 (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 18 (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. □

H.6 Substitution Lemma

Lemma 19 (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 "lock $\overline{v}$" nor "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).

□

H.7 Values Lemma

Lemma 20 (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.

□

H.8 Protocol Lemmas

Lemma 21 (Composition Progress)**.** If $\Gamma \vdash R \Rrightarrow P \parallel Q$ then $\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle \mapsto C$.

In other words, if two protocols compose safely, then the configuration can always take a step to some other set of configurations.

Proof. By induction on the derivation of $\Gamma \vdash R \Rrightarrow P \parallel Q$:

$\Gamma \vdash R \Rrightarrow P \parallel Q$ (1)

by hypothesis.

$\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle\!\uparrow$ (2)

by inversion on (1) with (WF:SPLIT).

$\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle \mapsto C$ (3)

$C\!\uparrow$ (4)

by inversion on (2) with (WF:CONFIGURATION).

Thus, we conclude by (3).

□

Lemma 22 (Composition Preservation)**.** If $\Gamma \vdash R \Rrightarrow P \parallel Q$ and $\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash R' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C$ then $\Gamma' \vdash R' \Rrightarrow P' \parallel Q'$.

If two protocols compose safely then the respective configuration can take a step by (Composition Progress). We now show that the resulting configuration still composes safely. 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 compose 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 $\Gamma \vdash R \Rrightarrow P \parallel Q$:

$\Gamma \vdash R \Rrightarrow P \parallel Q$ (1)

$\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash R' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C$ (2)

by hypothesis.

$\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle\!\uparrow$ (3)

by inversion on (1) with (WF:SPLIT).

$\langle\, \Gamma \vdash R \Rrightarrow P \parallel Q \,\rangle \mapsto \langle\, \Gamma' \vdash R' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C$ (4)

$\langle\, \Gamma' \vdash R' \Rrightarrow P' \parallel Q' \,\rangle \cdot\ C\!\uparrow$ (5)

by inversion on (2) with (WF:CONFIGURATION).

$\langle\, \Gamma' \vdash R' \Rrightarrow P' \parallel Q' \,\rangle\!\uparrow$ (6)

by (WF:CONFIGURATION) and (CF:ALLSTEP) on (5).

$\Gamma' \vdash R' \Rrightarrow P' \parallel Q'$ (7)

by (WF:SPLIT) on (6).

Thus, we conclude by (7).

□

Lemma 23 (Protocol Properties)**.** Protocol Composition obeys the follow properties:

$$\Gamma \vdash R \Rrightarrow R \parallel \textbf{none} \qquad \text{(Identity)}$$
$$\text{If } \Gamma \vdash R \Rrightarrow P_0 \parallel P_1 \text{ then } \Gamma \vdash R \Rrightarrow P_1 \parallel P_0 \qquad \text{(Commutativity)}$$
$$\text{If } \Gamma \vdash R \Rrightarrow P_0 \parallel (P_1 \parallel P_2) \text{ then } \Gamma \vdash R \Rrightarrow (P_0 \parallel P_1) \parallel P_2 \qquad \text{(Associativity)}$$

Showing identity and that protocol composition is commutative are immediate. We now show that protocol composition is associative.

Lemma 24 (Associativity)**.** If we have:

$$\Gamma \vdash R \Rrightarrow P \parallel Q \qquad \Gamma \vdash P \Rrightarrow P_0 \parallel P_1$$

then, exists W such that:

$$\Gamma \vdash R \Rrightarrow P_0 \parallel W \qquad \Gamma \vdash W \Rrightarrow P_1 \parallel Q$$

(i.e. if $\Gamma \vdash R \Rrightarrow \underbrace{(P_0 \parallel P_1)}_{P} \parallel Q$ then $\Gamma \vdash R \Rrightarrow P_0 \parallel \underbrace{(P_1 \parallel Q)}_{W}$).

Proof. We proceed by induction on the structure of S. Note that we omit cases related to the use of (CF-RS:SUBSUMPTION) and (CF-RS:WEAKENING) since they are straightforward and detract from the core aspects of the proof.

1. Case $R = \textbf{none}$. Immediate since neither protocol can take a step.

2. Case $R = R_0 \oplus R_1$, we have:

   $\Gamma \vdash R_0 \oplus R_1 \Rrightarrow P \parallel Q$ (1)
   $\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)
   by hypothesis.
   $\Gamma \vdash R_0 \Rrightarrow P \parallel Q$ (3)
   $\Gamma \vdash R_1 \Rrightarrow P \parallel Q$ (4)
   by inversion with (CF-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):

   $\Gamma \vdash R_0 \Rrightarrow P_0 \parallel W$ (5)
   $\Gamma \vdash W \Rrightarrow P_1 \parallel Q$ (6)
   $\Gamma \vdash R_1 \Rrightarrow P_0 \parallel W$ (7)
   by induction hypothesis on (2) and (3), and (2) and (4).
   $\Gamma \vdash R_0 \oplus R_1 \Rrightarrow P_0 \parallel W$ (8)
   by (CF-RS:STATEALTERNATIVE) with (5) and (7).

   Thus, we conclude.

3. Case $R = R_0 \& R_1$, we have:

   $\Gamma \vdash R_0 \& R_1 \Rrightarrow P \parallel Q$ (1)
   $\Gamma \vdash P \Rrightarrow P_0 \parallel P_1$ (2)
   by hypothesis.
   $\Gamma \vdash R_0 \Rrightarrow P \parallel Q$ (3)
   by inversion with (CF-RS:StateIntersection) on (1).
   Then, by induction hypothesis we have that exists W such that:
   $\Gamma \vdash R_0 \Rrightarrow P_0 \parallel W$ (4)
   $\Gamma \vdash W \Rrightarrow P_1 \parallel Q$ (5)
   by induction hypothesis with (2) and (3).
   $\Gamma \vdash R_0 \& R_1 \Rrightarrow P_0 \parallel W$ (6)
   by (CF-RS:StateIntersection) on (4).
   Thus, we conclude.

4. Case $R = S$ (a non-protocol type), we have:

   $\Gamma \vdash S \Rrightarrow P \parallel Q$ (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 $P =$ **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 (CF-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 $P = A \Rightarrow \forall X <: A''.P'$ are straightforward by inversion and application of the induction hypothesis. Similarly, the sub-case where P_0 (or P_1) transition by (CF-PS:TypeApp) or by (CF-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 $P = \exists X <: A''.P'$ and $P = \exists l.P'$ are straightforward by inversion on (CF-SS:Open*) and (CF-PS:Exists*), induction hypothesis, and re-applying the rule.

(d) Case $P = S$ (ownership recovery), we have:

$$\Gamma \vdash S \Rrightarrow S \parallel Q \tag{1}$$
$$\Gamma \vdash S \Rrightarrow P_0 \parallel P_1 \tag{2}$$

by hypothesis.

Then, by inversion on (2) we have that either:

- P_0 or P_1 are also S. Then, since the protocols compose safely, 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 S, i.e. they are appending some protocol to what was previously ownership recovery. Thus, we have:

  $$P_0 = S \Rightarrow S'; P'_0 \tag{3}$$
  $$P_1 = S \Rightarrow S'; P'_1 \tag{4}$$

  by inversion on (2) with (CF-PS:Step).

  But since Q composes with S, Q cannot use the shared state and must be equivalent of **none**. Therefore, we can build $W = P_1$ and immediately conclude.

5. Case $R = A \Rightarrow A'; P''$ (a protocol type), we have:

$$\Gamma \vdash (A \Rightarrow A'; P'') \Rrightarrow P \parallel Q \tag{1}$$
$$\Gamma \vdash P \Rrightarrow P_0 \parallel P_1 \tag{2}$$

by hypothesis.

$$P = A \Rightarrow A'; P' \tag{3}$$
$$P_0 = A \Rightarrow A'; P'_0 \tag{4}$$
$$P_1 = A \Rightarrow A'; P'_1 \tag{5}$$

by inversion on (2) with (CF-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 $R = \exists l.R'$ (case $R = \exists X <: A''.R'$ is analogous), we have:

$\Gamma \vdash \exists l.R' \Rrightarrow P \parallel Q$ (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 compose with R and P.
$\Gamma, l : \mathbf{loc} \vdash R' \Rrightarrow P' \parallel Q'$ (7)
$\Gamma, l : \mathbf{loc} \vdash P' \Rrightarrow P'_0 \parallel P'_1$ (8)
by inversion on (2) with (CF-PS:ExistsLoc).
Then, by induction hypothesis there exists a W such that:
$\Gamma, l : \mathbf{loc} \vdash R' \Rrightarrow P'_0 \parallel W$ (9)
$\Gamma, l : \mathbf{loc} \vdash W \Rrightarrow P'_1 \parallel Q'$ (10)
by induction hypothesis on (7) and (8).
$\Gamma \vdash \exists l.R' \Rrightarrow \exists l.P'_0 \parallel \exists l.W$ (11)
$\Gamma \vdash \exists l.W \Rrightarrow \exists l.P'_1 \parallel \exists l.Q'$ (12)
by (CF-PS:ExistsLoc) on (9) and (10).
Thus, we conclude.

7. Case $R = A \Rightarrow \forall l.R''$ (case $R = S \Rightarrow \forall X <: A''.R''$ is analogous), we have:

$\Gamma \vdash A \Rightarrow \forall l.R' \Rrightarrow P \parallel Q$ (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.

□

H.9 Preservation

Note that preservation is only defined over closed programs and expressions so that they can step.

Theorem 5 (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 6 (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 $\mathcal{E} = \square$. Also note that we have $\Delta_T = \cdot$ since there is no resulting new thread from this step.

$\Gamma_0 \mid \Delta_0 \vdash \{\overline{\mathtt{f} = v}\} : [\overline{\mathtt{f} : A}] \dashv \Delta$ (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 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 compose 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).

□

H.10 Progress

We define progress over closed programs (which includes an arbitrarily large but finite number of thread, T) and expressions (e).

Theorem 7 (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 $\texttt{live}$.
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 8 (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 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 $\mathcal{E} = \square$, i.e.:
$\square[v.\mathtt{f}_i]$ (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 $\mathcal{E} = \square$, i.e.:
$\square[v_0\ v_1]$ (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:
  - ⋄ (steps) $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$ (4)
  - ⋄ (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:
  - ⋄ (steps) $H_0 \; ; \; e_0 \; \mapsto \; H_1 \; ; \; e_1 \; \cdot \; T$ (7)
  - ⋄ (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 $\mathcal{E} = \Box$, i.e.:

$\Box[\mathsf{fork}\ e]$ (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 $\mathcal{E} = \Box$.
- Some of the locks of the locations of A_0 are unavailable. Then we have that the expression waits, i.e. we have $\mathtt{Wait}(H_0, \mathsf{lock}\ \overline{v})$.

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) $\mathtt{Wait}(H_0, e_0)$ (8)
  then $\mathtt{Wait}(H_0, \mathsf{let}\ x = e_0\ \mathsf{in}\ e_2\ \mathsf{end})$ by the definition of `Wait` since we have: $\mathtt{Wait}(H_0, \mathcal{E}[e_0])$
  where $\mathcal{E} = (\mathsf{let}\ x = \mathcal{E}'[e_0]\ \mathsf{in}\ e_2\ \mathsf{end})$ where $\mathcal{E}' = \square$.

Thus, we conclude.

□

I Algorithms Theorems

We now discuss the algorithmic implementations of protocol composition (shown in Section F.1) and subtyping (shown in Section F.2). Namely, we discuss the decidability of these algorithms. The decidability of the full system will likely require annotated terms to guide the type system in less obvious choices, such as in what to split a resource and when. Although they did not do a formal discussion of decidability, prior work on rely-guarantee protocols [21] includes a prototype implementation that uses additional type annotations to make typing practically syntax directed. Note that more practical implementations may reduce some of the annotation overhead of [21] by pushing additional verification complexity to the type checker (i.e. trade annotations for "type-searching" algorithms). In here we focus on the core issue of decidability of protocol composition *without* the need for additional type annotations beyond the initial resource to be split and the two resulting protocols.

Our subtyping algorithm uses a variant of the algorithm described in [2] to ensure decidability of subtyping on recursive types. The main distinction is that our recursive types include parameters, thus our recursive types may include arguments that introduce subtle decidability issues. For this reason, our focus is on treating the new problems in a clear way. For instance, to avoid the general undecidability of subtyping over bounded quantification [26] we use the subtyping rule of $\mathtt{F}_{<:}$ [27], although other more flexible subtyping rules exists [6].

Note that both both (ST:PACK*) and (ST:*APP) rules use a *unification algorithm* that searches for a single type/location match to be used in the rule. Since we at most traverse the type structure to find such a type/location, the algorithm naturally terminates through the same reasoning as with subtyping. Type equality follows similar reasoning. Therefore, our reasoning assumes that there exists a decidable, sound, and complete unification ($A\{B/X\} = C$) and equality ($A \equiv B$) algorithms. Finally, also note that we do not consider types (such as $\bot = \mathbf{rec}\ X.X$) to be well-formed types. Since we use the same cycle detection technique of [2], allowing these types could yield wrong results.

I.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 $\mathtt{int} \multimap \mathtt{bool} <: \mathtt{string} \multimap \mathtt{double}$ is not valid requires checking $\mathtt{string} <: \mathtt{int}$ and $\mathtt{bool} <: \mathtt{double}$, 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.

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):

$$\begin{array}{r} \underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [\texttt{int}] \\ \texttt{int} \multimap \underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [\texttt{int} \multimap \texttt{int}] \\ \texttt{int} \multimap \texttt{int} \multimap \underline{(\textbf{rec}\ R(X).(\ \texttt{int} \multimap R[X \multimap X]\ ))}\ [(\texttt{int} \multimap \texttt{int}) \multimap (\texttt{int} \multimap \texttt{int})] \\ \ldots \end{array}$$

for clarity, we replace the underlined type with just R to highlight the irregular set of sub-terms:

$$\begin{array}{r} R[\texttt{int}] \\ \texttt{int} \multimap R[\texttt{int} \multimap \texttt{int}] \\ \texttt{int} \multimap \texttt{int} \multimap R[(\texttt{int} \multimap \texttt{int}) \multimap (\texttt{int} \multimap \texttt{int})] \\ \ldots \end{array}$$

We see that the type that results from unfolding is not regular, as the use of the recursive variable R in "**rec** $R(X).(\ \texttt{int} \multimap R[X \multimap X]\ )$" 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 G.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 "... → **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:

$$\frac{u_0 : k_0, ..., u_n : k_n, X : k_0 \to ... \to k_n \to \textbf{type} \vdash A\ \textbf{type} \qquad \Gamma \vdash U_i\ k_i \qquad k_i = \texttt{kind}(u_i) \qquad i \in \{0, ..., n\}}{\Gamma \vdash (\textbf{rec}\ X(\overline{u}).A)[\overline{U}]\ \textbf{type}}$$

where: $\texttt{kind}(l) = \textbf{loc}$ and $\texttt{kind}(X) = \textbf{type}$. Thus, using $\texttt{int}$ as argument in $(\textbf{rec}\ R(X).A)[\texttt{int}]$ 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:

$$(\textbf{rec}\ V(Z).(\forall X <: (A \multimap Z).V[X]))[\texttt{int}] \tag{Ex.2}$$

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):

$$\begin{aligned} \cdot &\vdash V[\texttt{int}] \\ X_0 <: A \multimap \texttt{int} &\vdash V[X_0] \\ X_1 <: A \multimap X_0,\ X_0 <: A \multimap \texttt{int} &\vdash V[X_1] \\ X_2 <: A \multimap X_1,\ X_1 <: A \multimap X_0,\ X_0 <: A \multimap \texttt{int} &\vdash V[X_2] \\ &\dots \end{aligned}$$

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:

$$\frac{\cdot \vdash A_0 \ \textbf{type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1 \ \textbf{type}}{\Gamma \vdash \forall X <: A_0 . A_1 \ \textbf{type}} \qquad \frac{\cdot \vdash A_0 \ \textbf{type} \qquad \Gamma, X : \textbf{type}, X <: A_0 \vdash A_1 \ \textbf{type}}{\Gamma \vdash \exists X <: A_0 . A_1 \ \textbf{type}}$$

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:

$$(\textbf{rec}\ M(Y).(Y \multimap \forall X <: \textbf{top}.M[X]))[\texttt{int}] \tag{Ex.3}$$

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):

$$\begin{aligned} \cdot &\vdash M[\texttt{int}] \\ X_0 <: \textbf{top} &\vdash M[X_0] \\ X_1 <: \textbf{top}, X_0 <: \textbf{top} &\vdash M[X_1] \\ X_2 <: \textbf{top}, X_1 <: \textbf{top}, X_0 <: \textbf{top} &\vdash M[X_2] \\ &\dots \end{aligned}$$

By inversion on weakening of assumptions, we can consider the last context to only really require the "$X_2 <: \textbf{top}$" 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.

$$\frac{\frac{Z <: \textbf{top} \vdash M[Z] \quad \equiv \quad Z <: \textbf{top} \vdash M[Z]}{(X_2 <: \textbf{top})\{Z/X_2\} \vdash (M[X_2])\{Z/X_2\} \quad \equiv \quad (X_0 <: \textbf{top})\{Z/X_0\} \vdash (M[X_0])\{Z/X_0\}}}{X_2 <: \textbf{top}, X_1 <: \textbf{top}, X_0 <: \textbf{top} \vdash M[X_2] \quad \equiv \quad X_0 <: \textbf{top} \vdash M[X_0]}$$

Thus, we consider equivalence up to renaming of variables and weakening of assumptions—besides the other restrictions discussed above.

I.1.1 Finite Sub-terms

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.

$$
\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 35: Computing the (infinite) set of sub-terms of a type (**sts**).

Figure 35 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**.

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):

$$
\begin{array}{lcll}
\Gamma \vdash A & \equiv & \Gamma \vdash A & & \text{(equality)} \\
\Gamma_0, \Gamma_1 \vdash A_0 & \equiv & \Gamma_2, \Gamma_3 \vdash A_1 & \textbf{if}\ \ \Gamma_1 \vdash A_0 \ \equiv\ \Gamma_3 \vdash A_1 & \text{(weakening)} \\
\Gamma_0 \vdash A_0 & \equiv & \Gamma_1 \vdash A_1 & \textbf{if}\ \ \Gamma_0\{Z/X\} \vdash A_0\{Z/X\} \ \equiv\ \Gamma_1\{Z/Y\} \vdash A_1\{Z/Y\}\ \textbf{and}\ Z\ \textbf{fresh} & \text{(renaming type)} \\
\Gamma_0 \vdash A_0 & \equiv & \Gamma_1 \vdash A_1 & \textbf{if}\ \ \Gamma_0\{l/t'\} \vdash A_0\{l/t'\} \ \equiv\ \Gamma_1\{l/t\} \vdash A_1\{l/t\}\ \textbf{and}\ l\ \textbf{fresh} & \text{(renaming loc)}
\end{array}
$$

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.

$$
\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 \ \mathtt{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 36: Computing the set of sub-terms of a type, up to equivalence ($\equiv$).

Lemma 25 (Finite Uses)**.** *Given a well-formed recursive type* (***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 26 (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 $\forall$ or $\exists$ 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 $\forall$ and $\exists$ 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 27 (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 36) terminates and that **st** produces a set that is equivalent to the set produced by **sts** (Figure 35), 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.

□

I.2 Protocol Composition Lemmas

The proofs of the following lemmas are generally straightforward, after all that was the point in using an axiomatic definition of composition.

Lemma 28 (`stp` soundness)**.** For some v and well-formed Γ, R, $\mathcal{R}[P]$ if $\texttt{stp}(\Gamma, R, \mathcal{R}[P], v)$ then $\Gamma \vdash R \Rrightarrow \mathcal{R}[P]$, if the step is not in v.

Proof. Straightforward to show by induction on the cases of `stp`. Note that the algorithm (in Section F.1) includes a comment with the equivalent axiomatic rule to clarify which case the algorithm is implementing. We highlight cases (4) and (5) that match (EQ:REC) since the axiomatic definition is defined up to unfolding of recursive types, thus these two cases are implicit in the axiomatic definition. Similarly, cases (10) and (11) (also for (12) and (13)) make explicit that either case of & (and $\oplus$) suffices. Although they match a single rule in the axiomatic definition this is because we assume that both & are $\oplus$ are commutative but made that fact explicit in the algorithm. □

Lemma 29 (c soundness)**.** Given well-formed Γ, R, P, Q if $\mathtt{c}(\Gamma,\ R,\ P,\ Q)$ then $\Gamma \vdash R \Rrightarrow P \parallel Q$.

Proof. The proof is straightforward by induction on the cases of the algorithm since each case of the algorithm directly matches a specific rule of the axiomatic definition. The use of a *v*isited set simply tracks the derivation of composition by keeping the set of visited configurations as we go conclusion to premise on the co-inductive proof. Thus, the lemma can be restated as: "For some v and well-formed Γ, R, P, Q if $\mathtt{c}(\Gamma,\ R,\ P,\ Q,\ v)$ then $\Gamma \vdash R \Rrightarrow P \parallel Q$, where all members of v step." We have that case (1) we conclude by induction hypothesis with an empty v. Case (2) we apply (stp soundness) on each step of the configuration, apply the induction hypothesis on the recursive call to c using the new v, and then use the steps of (stp soundness) and the induction hypothesis to conclude by (c:Step) and (c:AllStep). □

Lemma 30 (stp completeness)**.** For well-formed $\Gamma, R, \mathcal{R}[P]$ if $\Gamma \vdash R \Rrightarrow \mathcal{R}[P]$ then $\mathtt{stp}(\Gamma, R, \mathcal{R}[P], v)$ for some v that does not contain the step.

Proof. Straightforward to show following similar reasoning to soundness. □

Lemma 31 (c completeness)**.** Given well-formed Γ, R, P, Q if $\Gamma \vdash R \Rrightarrow P \parallel Q$ then $\mathtt{c}(\Gamma,\ R,\ P,\ Q)$.

Proof. Straightforward to show following similar reasoning to soundness. □

Lemma 32 (c decidability)**.** Given well-formed Γ, R, P, Q then $\mathtt{c}(\ \Gamma,\ R,\ P,\ Q)$ terminates.

Proof. The proof follows immediately by the (Finite Sub-Terms) lemma as it ensure that R, P, and Q must contain a finite set of distinct sub-terms. Thus, stepping of a protocol must therefore be bounded in the number of protocol configurations that the algorithm can visit and consequently c must terminate.

□

Subtyping Extension Further below we show that sbt is sound, complete, and decidable so it should be straightforward to show that even when we add the subtyping conditions, c remains at least sound and decidable. The problem is whether protocol composition remains complete when we consider the (c-rs:Subsumption) stepping rule since the 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 compose safely without that rule being present, i.e. (c-rs:Subsumption) is admissible. Instead, (c-rs:Subsumption) is only important to close the proof "earlier" by reducing the number of configurations that we need to visit to check composition. Thus, c remains complete without the need for an explicit (c-rs:Subsumption).

Lemma 33 ((c-rs:Subsumption) admissible)**.** If $\Gamma \vdash R_1 <: R_0$ and $\langle\ \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P_0]\ \rangle \mapsto C$ and $\Gamma \vdash P_0 <: P_1$ then $\langle\ \Gamma \vdash R_1 \Rrightarrow \mathcal{R}[P_1]\ \rangle \mapsto C$, without using (c-rs:Subsumption).

Proof. The proof is straightforward as composition enforces breaking the $\oplus$ and & cases, and the changes to (c-ss:Step) and (c-ps:Step) ensure congruence of subtyping is still derivable. Note that there are no rules for congruence on subtyping over protocols. □

I.3 Subtyping Lemmas

Lemma 34 (`sbt` soundness)**.** Given well-formed Γ, A, B if $\texttt{sbt}(\Gamma,\ A,\ B)$ then $\Gamma \vdash A <: B$.

Proof. The lemma can be restated as: "For some t and given well-formed Γ, A, B if $\texttt{sbt}(\Gamma,\ A,\ B,\ t)$ then $\Gamma \vdash A <: B$, where all members of t obey the same equivalence". We proceed by case analysis on each case of the algorithm. Case (1) is proven by induction hypothesis where t is the empty set. The following cases are straightforward as we include the comment on the corresponding subtyping rule being applied. We have that by case (2) then we conclude by (ST:SYMMETRY); case (3) we conclude by induction hypothesis and (ST:TOLINEAR); etc. Cases (17), (18), (19), and (20) match the unfolding of recursive types and the set of visited subtyping configurations to close the co-inductive proof (which is left implicit in the co-inductive proof of the axiomatic definition). □

Lemma 35 (`sbt` completeness)**.** Given well-formed Γ, A, B if $\Gamma \vdash A <: B$ then $\texttt{sbt}(\Gamma,\ A,\ B)$.

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash A <: B$ and is straightforward as each case of the derivation matches a `sbt` case directly. We include a comment on the definition of `sbt` to highlight the axiomatic rule that is being modeled in the algorithm. □

Lemma 36 (`sbt` decidability)**.** Given well-formed types, A and B, and a well-formed environment, Γ, $\texttt{sbt}(\Gamma,\ A,\ B)$ terminates.

Proof. The proof follows immediately by the (Finite Sub-Terms) lemma. It suffices to see that each recursive call to `sbt` will necessarily be "smaller" in the sense of either considering a type that is *structurally* smaller or the recursive call including a larger *t*rail of already seen unfolds, even if the unfolded type may not be smaller. This trail represents the potentially infinite but regular (up to renaming of variables and weakening of Γ) "tree" of types modeled by a recursive type. Since by (Finite Sub-Terms) we know that unfolding a type will result in a finite number of sub-terms, we know that the sub-terms that `sbt` will visit are necessarily shrinking since there is a finite number of combinations to visit—even when considering the product of sub-terms of types A and B. By (17) we check whether a type was already "seen", which ensures termination after at most visiting all those combinations of sub-terms. Therefore there is an upper bound on the set of types to visit, which ensures that `sbt` cannot recur indefinitely. Thus, `sbt` must terminate. □