# Composing Interfering Abstract Protocols*

**Filipe Militão[1], Jonathan Aldrich[2], and Luís Caires[3]**

1 **Carnegie Mellon University, Pittsburgh, USA and Universidade Nova de Lisboa, Lisboa, Portugal**
filipe.militao@cs.cmu.edu
2 **Carnegie Mellon University, Pittsburgh, USA**
aldrich@cs.cmu.edu
3 **Universidade Nova de Lisboa, Lisboa, Portugal**
lcaires@fct.unl.pt

## 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 safe shared memory interference, we show how this same protocol framework can be used to model safe, typeful message-passing concurrency idioms.

**1998 ACM Subject Classification** D.3.3 Language Constructs and Features

**Keywords and phrases** shared memory interference, protocol composition, aliasing, linearity

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2016.16

## 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* [19]. By sequencing *steps* of

* 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.


"rely⇒guarantee" actions, each protocol characterizes an alias's local, isolated perspective on interactions with a piece of shared state:

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

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 [13]. 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 that may occur on that shared state.

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 [19] to work in a system with concurrent run-time 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] This algorithm is implemented in a prototype.[2]
- 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 [16, 6]).
- 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; followed by 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 $\lambda$-calculus

[1] 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 [19].

[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) | $v$ | ::= | $\rho$ | (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. $\rho$ is not source-level.

**Figure 1** Grammar — Values ($v$) and expressions ($e$).

shown in Figure 1. For convenience, the grammar is let-expanded so that all constructs, except let, are defined over values. The language includes first-class functions ($\lambda$), 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 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 companion *Technical Report* [21]. They produce the standard evaluation of the language's constructs such as creating or deleting memory, spawning new threads, etc.

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 [10] *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 $\exists l.(\ (!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)\ )$. This type abstracts the fresh location, $l$, that was created by the memory allocation. Furthermore, we are given a reference of type "$!\mathbf{ref}\ l$" 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 "$\mathbf{rw}\ l\ A$", 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* [20]. In $\exists l.(\ (!\mathbf{ref}\ l) :: (\mathbf{rw}\ l\ A)\ )$ we see that the capability to $l$ is stacked on top of "$!\mathbf{ref}\ l$" 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 [16, 20, 19, 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

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

```
          y :!ref l, rw l int
          y :!ref l, rw l string
x :!ref l, y :!ref l, rw l string
x :!ref l, y :!ref l, rw l boolean
 x :!ref l, y :!ref l
```

**Figure 2** Tracking linear capabilities.

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

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

| | | | | | |
|---|---|---|---|---|---|
| $A ::=$ | $!A$ | (pure/persistent) | $\mid$ | $(\mathbf{rec}\ X(\overline{u}).A)[\overline{U}]$ | (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) | $\mid$ | $\mathbf{rw}\ p\ A$ | (capability to $p$) |
| $\mid$ | $\forall l.A$ | (universal location) | $\mid$ | $\mathbf{none}$ | (empty resource) |
| $\mid$ | $\exists l.A$ | (existential location) | $\mid$ | $\mathbf{top}$ | (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) |
| $\mid$ | $\mathbf{ref}\ p$ | (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** Grammar — Types ($A$) (Continued from Figure 1.).

location's contents (as shown in Figure 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 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.

(Note that rely and guarantee types will only be discussed in the next section, when we present sharing). Our types (Figure 3) follow the connectives of linear logic [10]. 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:

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

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 ($X$).

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

Note: bounded variables of a construct and of quantifiers must be fresh in the rule's conclusion.

**Figure 4** Typing rules (selected, see [21] for complete set).

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 the $*$ operator such that "$A_0 :: (A_1 * A_2)$" 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.

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 $\Gamma$ and linear resources $\Delta_0$ we assign the expression $e$ the type $A$, with effects resulting in the resources in $\Delta_1$. The typing environments are defined as follows:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $\Gamma$ | $::=$ | $\cdot$ | | (empty) | $\Delta$ | $::=$ | $\cdot$ | (empty) |
| | $\mid$ | $\Gamma,\ x : A$ | | (variable binding) | | $\mid$ | $\Delta,\ x : A$ | (linear binding) |
| | $\mid$ | $\Gamma,\ p : \mathbf{loc}$ | | (location assertion) | | $\mid$ | $\Delta,\ A$ | (linear resource) |
| | $\mid$ | $\Gamma,\ X <: A$ | | (bound assertion) | | | | |
| | $\mid$ | $\Gamma,\ X : k$ | | (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.

Figure 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 T.R.. (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 [27] 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 $\Delta$ (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) enables non-syntax-directed opening of existential location packages.

The subtyping rules are deferred to the T.R., 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 type expectations.

## 2 A Protocol for Modeling Fork-Join Interactions

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 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:

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

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

[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.

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 $\Rightarrow$ ( Result ; **none** )".

The `Main` protocol includes two alternative ($\oplus$) 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:

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

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 ($\Delta$) 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 ($\Gamma$) because it only contains pure/duplicable assumptions. A possible use of the `F` protocol follows.

```
3  fork
4    let r = work {} in
5      lock c;
6      c := Result#r;
7      unlock c
8    end
```

| | |
|---|---|
| $\Gamma = \texttt{c} : \mathbf{ref}\ l, l : \texttt{loc}$ | $\mid \Delta = \texttt{work} : ![] \multimap \texttt{int}, \texttt{F}$ |
| $\Gamma = \texttt{r} : \texttt{int}, \ldots$ | $\mid \Delta = \mathsf{Wait} \Rightarrow (\mathsf{Result}\ ;\ \mathbf{none})$ |
| $\Gamma = \ldots$ | $\mid \Delta = \mathsf{Wait}, (\mathsf{Result}\ ;\ \mathbf{none})$ |
| $\Gamma = \ldots$ | $\mid \Delta = \mathsf{Result}, (\mathsf{Result}\ ;\ \mathbf{none})$ |
| $\Gamma = \ldots$ | $\mid \Delta = \mathbf{none}$ |
| $\Gamma = \ldots$ | $\mid \Delta = \cdot$ |

$\Gamma$ contains a reference (`c`) to the location ($l$) that is being shared by the protocol, and $\Delta$ 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 $\exists$ abstract its locations.) Line 4 consumes the function `work` by calling it and 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 (later on).

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

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).

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

(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 $\Delta$ empty ($\cdot$). 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.
2   let c = new Wait#{} in
3     fork ... // lines 3 to 8 shown above.
```

| | |
|---|---|
| $\Gamma = \cdot$ | $\mid \Delta =$ `work` $: ![] \multimap$ `int` |
| $\Gamma =$ `c` $:$ `ref` $l, l :$ `loc` | $\mid \Delta =$ `rw` $l$ `Wait#![]`, ... |
| $\Gamma =$ ... | $\mid \Delta =$ `M`, `F`, ... |

To simplify the presentation, our term language is stripped of type annotations. However, the newFork function has type $!(\ (\ ![] \multimap$ `int` $)\ \multimap\ (\ ![] \multimap$ `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 $\exists l.(\ (!\mathbf{ref}\ l) :: (\mathbf{rw}\ l$ `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 Figure 4), combined with the following rule for subtyping on $\Delta$'s:

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

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

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

This split results in the capability to location $l$ being replaced by the two protocols, `M` and `F`, in $\Delta$. 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
```

We omit Γ to center the discussion on the contents of Δ. The alternative type (⊕) 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 ⊕ (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:

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

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):

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

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, $\langle \Gamma \vdash \mathsf{Result} \Rrightarrow \mathtt{M} \mid\mid \mathtt{F} \rangle$ 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

[4] $\mathtt{M} \triangleq (\ \mathsf{Wait} \Rightarrow (\ \mathsf{Wait}\ ;\ \mathtt{M}\ )\ ) \oplus (\ \mathsf{Result} \Rightarrow (\ \mathsf{Done}\ ;\ \mathsf{Done}\ )\ )$
[5] $\mathtt{F} \triangleq \mathsf{Wait} \Rightarrow (\ \mathsf{Result}\ ;\ \textbf{none}\ )$

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

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

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: $\Gamma \vdash R \Rrightarrow P \mid\mid Q$. Figure 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:

$$C \quad ::= \quad C \cdot C \quad \text{(union)} \qquad \mid \quad \langle\, \Gamma \vdash R \Rrightarrow P \mid\mid Q \,\rangle \quad \text{(configuration)}$$

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

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

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 Figure 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 one remains stationary, i.e.:

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

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

We use three distinct label prefixes to group the stepping rules based on whether a rule is stepping over a protocol (C-PS:*), stepping over 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

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

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

**Composition — Reduction Step, (c-rs:*)**

$$\frac{}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[\mathsf{none}] \,\rangle \mapsto \langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[\mathsf{none}] \,\rangle}\ (\textsc{c-rs:None})$$

$$\frac{\langle\, \Gamma \vdash R_0 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R_0 \& R_1 \Rrightarrow \mathcal{R}[P] \,\rangle \mapsto C}\ (\textsc{c-rs:StateIntersection}) \qquad \frac{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0] \,\rangle \mapsto C}{\langle\, \Gamma \vdash R \Rrightarrow \mathcal{R}[P_0 \oplus P_1] \,\rangle \mapsto C}\ (\textsc{c-rs:ProtocolAlternative})$$

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

**Composition — State Stepping, (c-ss:*)**

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

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

**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 6** Basic protocol composition stepping rules.

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. 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 6 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: $\exists X.(\ \texttt{rw}\ l\ X\ \Rightarrow\ (\ \texttt{rw}\ l\ \texttt{Result\#int}\ ;\ \mathsf{none}\ )\ )$. 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: $\exists X.(\ \texttt{rw}\ l\ \texttt{Wait\#}X\ \Rightarrow\ (\ \texttt{rw}\ l\ \texttt{Wait\#}X\ ;\ \texttt{M}\ )\ )\ \oplus\ (\ldots)$.

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 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:

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

On the left protocol, the scope of $X$ extends beyond a single ($\Rightarrow$) 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 swap $X$'s to an $X$ of a different representation, in a way that would unsafely interfere with the right protocol's assumptions on the precise contents of the shared state. Thus, while the left protocol depends on an opaque type, that protocol still requires that the scope/"lifetime" of $X$ extends to the next step although the protocol does not impose any other type restrictions on $X$.

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`:

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

Protocol composition results in the following set of configurations:

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

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 $\Gamma$. 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.

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

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. `W` $\Rrightarrow$ `W` $||$ `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* [3].

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 interactions 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): $\Gamma \vdash \texttt{F}[\texttt{E\#}[]] \Rrightarrow \texttt{K} \parallel \texttt{K}$. 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:

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

$$\frac{\langle\ \Gamma_0 \vdash R \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C}{\langle\ \Gamma_0, \Gamma_1 \vdash R \Rrightarrow \mathcal{R}[P]\ \rangle \mapsto C}\ (\text{C-RS:WEAKENING}) \qquad \frac{\langle\ \Gamma, l : \mathbf{loc} \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow P]\ \rangle \mapsto C}{\langle\ \Gamma \vdash S \Rrightarrow \mathcal{R}[S \Rightarrow \forall l.P]\ \rangle \mapsto C}\ (\text{C-SS:FORALLLOC})$$

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

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

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

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

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

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

$P\{A/X\} \triangleq$ "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.

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

## 3.3 Composing Abstract Protocols

The composition rules of Figure 7 complement those of Figure 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 $\Gamma$, 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*) exposes 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.

```
let newMVar = λ _.
  let m = new Empty#{} in
    // splits the new cell using single 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
```

**Figure 8** MVar example.

## 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 [15]), and focus instead on typestates [20, 30, 29]. 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 [25, 11] 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 [25, 11] differ on whether the counter is forcefully used by clients, or whether the action was simply available to be used. We can model the two cases explicitly: $\exists p.(\ (!\textbf{ref}\ p) :: \texttt{MC} \multimap [] :: \texttt{MC}\ )$, which enables clients to use the counter an arbitrary number of times or simply thread it through, unused. While in:

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

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 `MVars` [24]. Figure 8 shows an `MVar`, a

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

**Figure 9** Indirect locking.

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

**Figure 10** Hand-over-hand locking example.

structure that contains 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 using the protocol:

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

The T.R. [21] 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.

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 (Figure 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 [20]. For instance: $\exists A.\exists B.\exists C.[\ \ldots,$ `lockMe` $:[] :: (A \Rightarrow B;C) \multimap [] :: (A * (B;C))$, `add` $:(\ \mathbf{int} :: A \multimap [] :: A\ ),\ \ldots\ ]$. 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 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 (Figure 10). Consider the protocol of a list's node:

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

$\mathsf{L}$ 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 $\mathsf{L}$, 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 [28, 4]. 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 T.R. 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* $(\textbf{\textit{rec}}\ X(\overline{u}).A)[\overline{U}]$ *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* $(\textbf{\textit{rec}}\ X(\overline{u}).A)[\overline{U}]$ *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* $\Gamma$*.*

### 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 T.R.. The next two lemmas show stuck freedom by properties that resemble progress and preservation but over protocols:

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

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* $\langle\ \Gamma \vdash R \Rrightarrow P \mid\mid Q\ \rangle \mapsto \langle\ \Gamma' \vdash R' \Rrightarrow P' \mid\mid Q'\ \rangle \cdot\ C$ *and* $\Gamma \vdash R \Rrightarrow P \mid\mid Q$ *then* $\Gamma' \vdash R' \Rrightarrow P' \mid\mid Q'$.

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

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 $\Delta$ or bundled using the $*$ type.

The composition algorithm is shown in the T.R. 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 $\Gamma$), the algorithm can terminate. We now state our technical lemmas on the composition algorithm but leave the proofs to the T.R..

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

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

## 4.2 Correctness Properties

Progress and preservation theorems 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 8.** *If* $\Gamma \mid \Delta \vdash T_0$ *and* $\texttt{live}(T_0)$ *and if exists* $H_0$ *such that* $\Gamma \mid \Delta \vdash H_0$ *then* $H_0 \; ; \; T_0 \; \mapsto \; H_1 \; ; \; T_1$.

$\texttt{live}(T)$ 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 $\Gamma$ and $\Delta$.

We define $\texttt{Wait}(H, e)$ 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 ($\rho$) 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:

▶ **Theorem 9.** *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* $\Delta$ *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 10.** *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* $\Delta_1$ *and* $\Gamma_1$, *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 ($\Delta_T$) that may be consumed by a newly spawned thread ($T$):

▶ **Theorem 11.** *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* $\Delta_1$ *and* $\Gamma_1$, *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* $\Gamma_0, \Gamma_1 \mid \Delta_T \vdash T$.

The following "Error Freedom" corollary complements the main results 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 12.** *The following program states cannot be typed:*

1. *(Data Race)* $H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[!\rho] \cdot T \qquad H; \mathcal{E}_0[\rho := v] \cdot \mathcal{E}_1[\rho := v'] \cdot T$
2. *(Memory Fault)* $H; \mathcal{E}[\rho := v] \cdot T \qquad H; \mathcal{E}[!\rho] \cdot T \qquad$ *(where* $\rho \notin H$*)*
3. *(Ownership Fault)* $H; \mathcal{E}[\mathsf{delete}\ \rho] \cdot T \qquad$ *(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.

```
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
```

**Figure 12** Buyer code.

## 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 (Figure 11). The encoding itself should be unsurprising as it follows well-known ideas from the literature, so we defer less important details to the T.R. 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 hide the underlying Waiting states, for instance to receive:

$$\mathsf{Wait}[A_0..A_n] \oplus \ (\ \mathtt{rw\ c\ ReadyToReceive}\#V \Rightarrow \mathtt{rw\ c\ Idle}\#[]\ ;\ \mathtt{NextStep}\ )$$

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 T.R. 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 (Figure 12).

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 $\pi$-calculus [22] notations where ! sends and ? receives a value. These actions are mapped to the rely type (receive) and the guarantee type (send).

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

`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 to 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:

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

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

This work is based on results collected in Militão's PhD thesis [18]. Our work relates to prior work on *rely-guarantee protocols* [19]. 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), 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 for threads and locking of locations; which then enables us to express typeful message-passing concurrency in our protocols.

Our work is also related to recent work on more precise tracking of interference. *Chalice* [17] 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 [8, 25] 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 [33, 26] choose some run-time overhead and dynamic safety guarantees to enable more flexible ownership recovery than

purely static approaches. *Rely-guarantee references* [11] 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* [33, 3, 2] 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 case. 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 [5] 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* [7, 16, 6] to model sharing by leveraging the concept of fictional separation [7, 12]. 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 [16] due to our common use of $\mathbf{L}^3$ [1] and type-based approach. In [16], 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 sharing. 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. [14, 31, 32, 23, 9]). 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 [9] 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 [32] and Iris [14] support "islands"/regions of memory that are shared together and whose imprecise state must be considered on use. 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 richer specifications of atomic regions of any depth. Iris [14] further supports a form of re-splitting via a "view shifting" mechanism, to repartition (or create) shared regions. FCSL [23] 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 composition mechanism is essentially a form of checking for safe interference early, at the moment of the split, by checking that *all* possible future uses are safe (like 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 also model typeful message-passing concurrency.

**Acknowledgements.** We thank the anonymous reviewers for their helpful feedback.

## References

1 A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 77(4):397–449, December 2007.
2 N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA 2008*.
3 K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA 2007*.
4 G. Castagna and B. C. Pierce. Decidable bounded quantification. In *POPL 1994*.
5 S. Crafa and L. Padovani. The chemical approach to typestate-oriented programming. In *OOPSLA 2015*.
6 T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL 2013*.
7 T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP 2010*.
8 M. Fähndrich and K. Rustan M. Leino. Heap monotonic typestate. In *IWACO 2003*.
9 X. Feng. Local rely-guarantee reasoning. In *POPL '09*.
10 J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
11 C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *PLDI 2013*.
12 J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP 2012*.
13 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 1983*.

14 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*.

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

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

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

18 F. Militão. *Rely-Guarantee Protocols for Safe Interference over Shared Memory*. PhD thesis, Carnegie Mellon University and Universidade Nova de Lisboa, 2015.

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

20 F. Militão, J. Aldrich, and L. Caires. Substructural typestates. In *PLPV 2014*.

21 F. Militão, J. Aldrich, and L. Caires. Composing interfering abstract protocols (technical report). CMU-CS-16-103, 2016.

22 R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Inf. Comput.*, September 1992.

23 A. Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP 2014*.

24 S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL 1996*.

25 A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI 2011*.

26 F. Pottier and J. Protzenko. Programming with permissions in mezzo. In *ICFP 2013*.

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

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

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

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

31 K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP 2014*.

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

33 R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP 2011*.