7-2014

Rely-Guarantee Protocols

Filipe Militao
Carnegie Mellon University

Jonathan Aldrich
Carnegie Mellon University

Luis Caires
Universidade Nova de Lisboa

Published In
Lecture Notes in Computer Science, 8586, 334-359.

Rely-Guarantee Protocols

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

[1] Carnegie Mellon University, Pittsburgh, USA
[2] Universidade Nova de Lisboa, Lisboa, Portugal
{filipe.militao,jonathan.aldrich}@cs.cmu.edu lcaires@fct.unl.pt

Abstract. The use of shared mutable state, commonly seen in object-oriented systems, is often problematic due to the potential conflicting interactions between aliases to the same state. We present a substructural type system outfitted with a novel lightweight interference control mechanism, *rely-guarantee protocols*, that enables controlled aliasing of shared resources. By assigning each alias separate roles, encoded in a novel protocol abstraction in the spirit of rely-guarantee reasoning, our type system ensures that challenging uses of shared state will never interfere in an unsafe fashion. In particular, rely-guarantee protocols ensure that each alias will never observe an unexpected value, or type, when inspecting shared memory regardless of how the changes to that shared state (originating from potentially unknown program contexts) are interleaved at run-time.

1 Introduction

Shared, mutable state can be useful in certain algorithms, in modeling stateful systems, and in structuring programs. However, it can also make reasoning about a program more difficult, potentially resulting in run-time errors. If two pieces of code have references to the same location in memory, and one of them updates the contents of that cell, the update may *destructively interfere* by breaking the other piece of code's assumptions about the properties of the value contained in that cell—which may cause the program to compute the wrong result, or even to abruptly terminate. In order to mitigate this problem, static type systems conservatively associate an invariant type with each location, and ensure that every store to the location preserves this type. While this approach can ensure basic memory safety, it cannot check higher-level *protocol* properties [1, 4, 5, 13, 20] that are vital to the correctness of many programs [3].

For example, consider a `Pipe` abstraction that is used to communicate between two parts of the program. A pipe is *open* while the communication is ongoing, but when the pipe is no longer needed it is *closed*. Pipes include shared, mutable state in the form of an internal buffer, and abstractions such as Java's `PipedInputStream` also dynamically track whether they are in the open or closed state. The state of the pipe determines what operations may be performed, and invoking an inappropriate operation is an error: for example, writing to a closed pipe in Java results in a run-time exception.

Static approaches to reason about such state protocols (of which we follow the *typestate* [7, 22, 28, 29] approach) have two advantages: errors such as writing to a closed pipe can be avoided on the one hand, and defensive run-time tests of the state of an object can become superfluous on the other hand. In typestate systems, abstractions expose a more refined type that models a set of abstract states representing the internal,

changing, type of the state (such as the two states above, *open* and *closed*) enabling the static modular manipulation of stateful objects. However, sharing (such as by aliasing) these resources must be carefully controlled to avoid potentially destructive interference that may result from mixing incompatible changes to apparently unrelated objects that, in reality, are connected to the same underlying run-time object. This work aims to provide an intuitive and general-purpose extension to the typestate model by exploiting (coordination) protocols at the shared state level to allow fine-grained and flexible uses of aliased state. Therefore, by modeling the *interactions* of aliases of some shared state in a protocol abstraction, we enable complex uses of sharing to safely occur through *benign interference*, interference that the other aliases expect and/or require to occur.

Consider once more the pipe example. The next two code blocks implement simplified versions of the pipe's `put` and `tryTake` functions. Although each function operates independently of the other, internally they share nodes of the same underlying buffer:

```
// protocol: Empty ⇒ Filled; none
put = fun( v : Value ).
  // Empty shared node, oldlast, to be filled with node
  // containing tagged (#) empty record, {}, as 'Empty'
  let last = new Empty#{} in
    let oldlast = !buffer.tail in // is Empty
      // tags pair of 'v' and 'last' as 'Filled'
      oldlast := Filled#{ v , last };
      buffer.tail := last
    end // last cell is now reachable from head&tail
  end // oldlast cell unreachable from tail
```

```
// rec X.( Empty ⇒ Empty; X ⊕ Filled ⇒ none )
tryTake = fun().
  let first = !buffer.head in
    case !first of
      Empty#_ → NoResult#{}
    | Filled#[ v , next ] → // does not return
        delete first; // ownership to the protocol
        buffer.head := next;
        Result#v
    end
  end
```

By distributing these functions between two aliases, we are able to create independent *producer* and *consumer* components of the pipe that share a common buffer (modeled as a singly-linked list). Observe how the interaction, that occurs through aliases of the buffer's nodes, obeys a well-defined protocol: the *producer* alias (through the `put` function) inserts an element into the last (empty) node of the buffer and then immediately forfeits that cell (i.e. it is no longer used by that alias); while the *consumer* alias (using `tryTake`) proceeds by testing the first node and, when it detects it has been `Filled` (thus, when the other alias is sure to no longer use it), recovers ownership of that node, which enables the alias to safely delete that cell (`first`) since it is no longer shared.

1.1 Approach in a Nutshell

Interference due to aliasing is analogous to the interference caused by thread interleaving [15, 33]. This occurs because mutable state may be shared by aliases in unknown or non-local program contexts. Such boundary effectively negates the use of static mechanisms to track exactly which other variables alias some state. Therefore, we are unable to know precisely if the shared state aliased by a local variable will be used when the execution jumps off (e.g. through a function call) to non-local program contexts. However, if that state is used, then the aliases may change the state in ways that invalidate the local alias' assumptions on the current contents of the shared state. This interference caused by "alias interleaving" occurs even without concurrency, but is analogous to how thread interleaving may affect shared state. Consequently, techniques to reason about thread interference (such as *rely-guarantee reasoning* [17]) can be useful to reason about aliasing even in our sequential setting. The core principle of rely-guarantee

reasoning that we adapt is its mechanism to make strong local assumptions in the face of interference. To handle such interference, each alias has its actions constrained to fit within a *guarantee* type and at the same time is free to assume that the changes done by other aliases of that state must fit within a *rely* type. The duality between what aliases can rely on and must guarantee among themselves yields significant flexibility in the use of shared state, when compared for instance to invariant-based sharing.

We employ rely-guarantee in a novel protocol abstraction that captures a partial view of the use of the shared state, as seen from the perspective of an alias. Therefore, each protocol models the constraints on the actions of that alias and is only aware of the resulting effects ("interference") that may appear in the shared state due to the interleaved uses of that shared state as done by other aliases. A rely-guarantee protocol is formed by a sequence of rely-guarantee steps. Each step contains a rely type, stating what an alias currently assumes the shared state contains; and a guarantee type, a promise that the changes done by that alias will fit within this type. Using these small building blocks, our technique allows strong local assumption on how the shared state may change, while not knowing when or if other aliases to that shared state will be used—only how they will interact with the shared state, if used. Since each step in a protocol can have distinct rely and guarantee types, a protocol is not frozen in time and can model different "temporal" uses of the shared state directly. A protocol is, therefore, an abstracted perspective on the actions done by each individual alias to the shared state, and that is only aware of the potential resulting effects of all the other aliases of that shared state. A *protocol conformance* mechanism ensures the sound composition of all protocols to the same shared state, at the moment of their creation. From there on, each protocol is *stable* (i.e. immune to unexpected/destructive interference) since conformance attested that each protocol, in isolation, is aware of all observable effects that may occur from all possible "alias interleaving" originated from the remaining aliases.

Our main contribution is a novel type-based protocol abstraction to reason about shared mutable state, *rely-guarantee protocols*, that captures the following features:

1. Each protocol provides a *local* type so that an alias need not know the actions that other aliases are doing, only their resulting (observable) effect on the shared state;
2. Sharing can be done *asymmetrically* so that the role of each alias in the interaction with the shared state may be distinct from the rest;
3. Our protocol paradigm is able to *scale* by modeling sharing interactions both at the reference level and also at the abstract state level. Therefore, sharing does not need to be embedded in an ADT [18], but can also work at the ADT level without requiring a wrapper reference [15];
4. State can be shared individually or simultaneously in groups of state. By enabling sharing to occur underneath a layer of apparently disjoint state, we naturally support the notion of *fictional disjointness* [9, 16, 18];
5. Our protocol abstraction is able to model complex interactions that occur through the shared state. These include invariant, monotonic and other coordinated uses. Moreover, they enable both *ownership transfer* of state between non-local program contexts and *ownership recovery*. Therefore, shared state can return to be non-shared, even allowing it to be later shared again and in such a way that is completely unrelated to its previous sharing phases;

6. Although protocol conformance is checked in pairs, *arbitrary aliasing* is possible (if safe) by further sharing a protocol in ways that do not conflict with the initial sharing. Therefore, global conformance in the use of the shared state by multiple aliases is assured by the combination of individual binary protocol splits, with each split sharing the state without breaking what was previously assumed on that state;
7. We allow *temporary inconsistencies*, so that the shared state may undergo intermediate (private) states that cannot be seen by other aliases. Using an idea similar to (static) mutual exclusion, we ensure that the same shared state cannot be inspected while it is inconsistent. Such kind of critical section (that does not incur in any run-time overhead) is sufficiently flexible to support multiple simultaneously inconsistent states, when they are sure to not be aliasing the same shared state.

With this technique we are able to model challenging uses of aliasing in a lightweight substructural type system, where all sharing is centered on a simple and intuitive protocol abstraction. We believe that by specializing our system to typestate and aliasing [1, 27] properties we can offer a useful intermediate point that is simpler than the full functional verification embodied in separation logic [6,25] yet more expressive than conventional type systems. Our proofs of soundness use standard progress and preservation theorems. We show that all allowed interference is benign (i.e. that all changes to the shared state are expected by each alias) by ensuring that a program cannot get stuck, while still allowing the shared state to be legally used in complex ways. Besides expressing the programmer's intent in the types, our technique also enables a program to be free of errors related to destructive interference. For instance, the programmer will not be able to wrongly attempt to use a shared cell as if it were no longer shared, or leave values in that shared cell that are not expected by the other aliases of that cell.

Section 2 introduces the language but leaves its sharing mechanisms to Section 4, after an overview of the type system. Section 5 discusses technical results, and Section 6 additional examples. The paper ends with Sections for related work and conclusions.

2 Pipe Example

Our language is based on the polymorphic λ-calculus with mutable references, immutable records, tagged sums and recursive types. Technically, we build on [22] (a variant of $\mathbf{L}^3$ [1] adapted for usability) by supporting sharing of mutable state through rely-guarantee protocols. As in $\mathbf{L}^3$, a cell is decomposed in two components: a pure *reference* (that can be freely copied), and a linear [14] *capability* used to track the contents of that cell. Unlike $\mathbf{L}^3$, by extending [22] our language implicitly threads capabilities through the code, reducing syntactic overhead. To support this separation of references and capabilities, our language uses location-dependent types to relate a reference to its respective capability. Therefore, a reference has a type "**ref** *t*" to mean a **ref**erence to a location *t*, where the information about the contents of that location is stored in the capability for *t*. Our capabilities follow the format "**rw** *t A*" meaning a **r**ead-**w**rite capability to location *t* which, currently, has contents of type *A* stored in it. The permission to access, such as by dereference, the contents of a cell requires both the reference and the capability to be available. Capabilities are typing artifacts that do not exist at run-

time and are moved implicitly through the code. Locations (such as t) must be managed explicitly, leading to constructs dedicated to abstracting and opening locations.

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

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

The function creates a pipe by allocating an initial `node` for the internal buffer, a cell to be shared by the `head` and `tail` pointers. The newly allocated cell (line 2) contains a tagged (as `Empty`) empty record ({}). In our language, aliasing information is correlated through static names, *locations*, such that multiple references to the same location must imply that these references are aliases of the same cell. Consequently, the `new` construct (line 2) must be assigned a type that abstracts the concrete location that was created, $\exists t.($ **ref** t :: **rw** t `Empty#[]` $)$, which means that there exists some fresh location t, and the new expression evaluates to a reference to t ("**ref** t"). We associate this reference with a capability to access it, using a *stacking* operator ::. In this case the capability is **rw** t `Empty#[]`, representing a **r**ead and **w**rite capability to the location t, which currently contains a value of type `Empty#[]` as initially mentioned. On the same line, we then `open` the existential by giving it a location variable n and a regular variable `node` to refer that reference. From there on, the capability (a typing artifact which has no actual value) is automatically *unstacked* and moved implicitly as needed through the program. For clarity, we will manually stack capabilities (such as on line 4, using the construct e :: A where A is the stacked capability), although the type system does not require it. On line 3, the type system initially carries the following assumptions:

$$\Gamma \;=\; \_ : [] \,,\; \texttt{node} : \textbf{ref}\; n \,,\; n : \textbf{loc} \qquad | \qquad \Delta \;=\; \textbf{rw}\; n\; \texttt{Empty\#[]}$$

where Γ is the lexical environment (of persistent/pure resources), and Δ is a linear typing environment that contains all linear resources (such as capabilities). Each linear capability must either be used up or passed on through the program (e.g. by returning it from a function). The contents of the reference `node` are known statically by looking up the capability for the **loc**ation n to which `node` refers (i.e. "**rw** n `Empty#[]`").

Capabilities are linear (cannot be duplicated), but aliasing in local contexts is still possible by copying references. All copies link back to the same capability using the location contained in the reference. However, when aliases operate in non-local contexts, this location-based link is lost. Thus, if we were to pack `node`'s capability before sharing it, it would become unavailable to other aliases of that location. For instance, by writing $\langle n, \texttt{node} :: \textbf{rw}\ n\ \texttt{Empty\#}[]\rangle$ we pack the location n by abstracting it in an existential type for that location. The packed type now refers a fresh location, unrelated to its old version. Instead, we share that capability (line 3) by splitting it in two rely-guarantee protocols, H and T [3]. Each protocol is then assigned to the `head` and `tail` pointers (lines 4 and 5, respectively), since they encode the specific uses of each of those aliases. The protocols and sharing mechanisms will be introduced in Section 4.

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

$$\texttt{newPipe}\ :\ !(\ [] \multimap \exists \mathsf{C}.\exists \mathsf{P}.(\ ![\ \ldots\ ] :: \mathsf{C} * \mathsf{P}\ )\ )$$

where the *separate* capabilities for the Consumer and Producer are stacked together in a commutative group ($*$). In this type, C abstracts the capability **rw** $h\ \exists p.$(**ref** p :: H[p]), and P abstracts **rw** $t\ \exists p.$(**ref** p :: T[p]). Finally, although we have not yet shown the implementation, the type of the elided record ([...]) contains function types that should be unsurprising noting that each argument and return type has the respective capabilities for the `head`/`tail` cells stacked on top (similarly to pre/post conditions, but directly expressed in the types). Therefore, those functions are closures that use the knowledge about the reference to the `head`/`tail` pointers from the surrounding context, but do not capture the capability to those cells and instead require them to be supplied as argument.

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

Therefore, `put` preserves the producer's capability, but `close` destroys it; while the result of `tryTake` is a sum type of either `Result` or `NoResult` depending on whether the still open pipe has or not contents available, or `Depleted` to signal that the pipe was closed (and therefore that the capability to C vanished). Observe that the state that the functions depend on is, apparently, disjoint although underneath this layer the state is actually shared (but coordinated through a protocol) so that (benign) interference must occur for the pipe to work properly—i.e. it is *fictionally disjoint* [9, 16, 18].

3 Type System Overview

We now present the type system. Non-essential details are relegated to [21, 22]. For consistency, we include all sharing mechanisms but leave their discussion to Section 4.

[3] As a brief glimpse, T is "**rw** n `Empty#`[] $\Rightarrow$ (**rw** n `Node#`R $\oplus$ **rw** n `Closed#`[]); **none**" which relies on n containing `Empty#`[], ensures n then contains either `Node#`R or `Closed#`[], and then loses access to n. Both "$\Rightarrow$" and ";" (and R) will be discussed in detail in Section 4.

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

$\mathtt{l} \in$ Labels (Tags) $\quad \mathtt{f} \in$ Fields $\quad x \in$ Variables $\quad X \in$ Type Variables

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

Note: ρ is not source-level. $\overline{Z}$ for a possibly empty sequence of Z. Tuples, recursion, etc. are encoded as idioms, see [22].

Fig. 1. Values (v) and expressions (e).

The (let-expanded [26]) grammar is shown in Fig. 1. The main deviations from standard λ-calculus are the inclusion of location-related constructs, and the sharing constructs (share, focus and defocus).

We use a flat type grammar (Fig. 2) where both capabilities (i.e. typing artifacts without values, which includes our rely-guarantee protocols) and standard types (used to type values) coexist. Our design does not need to make a syntactic distinction between the two kinds since the type system ensures the proper separation in their use. We now overview the basic types, leaving the rely and guarantee types to be presented in the following Section together with the discussion on sharing. Pure types $!A$ enable a linear type to be used multiple times. $A \multimap A'$ describes a linear function of argument A and result A'. The stacking operation $A :: A'$ stacks A' (a capability, or abstracted capability) on top of A. This stacking is not commutative since it stacks a single type on the right of ::. Therefore, $*$ enables multiple types to be grouped together that, when later stacked, allow that type to list a commutative group of capabilities[4]. Both $\forall$ and $\exists$ offer the standard quantification, over location and type kinds, together with the respective location/type variables. $[\overline{\mathtt{f} : A}]$ are used to described labeled records of arbitrary length. A **ref** p type is a reference for location p noting that the contents of such a reference are tracked by the capability to that location and not immediately stored in the reference type. **rec**ursive types, that are automatically folded/unfolded through subtyping rules (see Fig. 4 and (T:Subsumption) on Fig. 3), are also supported. Sum types use the form $\mathtt{tag}\#A$ to tag type A with $\mathtt{tag}$. Alternatives ($\oplus$) model imprecision in the knowledge of the type by listing different possible states it may be in. **none** is the empty capability, while **rw** p A is the read-write capability to location p (a memory cell currently con-

[4] Note that while $A_0 :: (A_1 :: A_2)$ and $A_0 :: (A_2 :: A_1)$ are not (necessarily) subtypes, capability commutation is always possible with $*$ such that $A_0 :: (A_1 * A_2) <:> A_0 :: (A_2 * A_1)$.

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

Note: $\sum_i \mathtt{l}_i \# A_i$ denotes a single tagged type or a sequence of tagged types separated by +, such as "$\mathtt{t}\#A + \mathtt{u}\#B + \mathtt{v}\#C$". Separation, sum, alternative and intersection types are assumed commutative, i.e. without respective subtyping rules.

Fig. 2. Types and capabilities.

taining a value of type A). Finally, an $A \& A'$ type means that the client can choose to use either type A or type A' but not both simultaneously.

Our typing rules use typing judgments of the form: $\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1$ stating that with lexical environment Γ and linear resources Δ_0 we assign the expression e a type A and produce effects that result in Δ_1. The typing environments are as follows:

$\Gamma ::= \cdot$		(empty)	$\Delta ::= \cdot$		(empty)
|	$\Gamma,\ x : A$	(variable binding)	|	$\Delta,\ x : A$	(linear binding)
|	$\Gamma,\ p : \mathbf{loc}$	(location variable assertion)	|	$\Delta,\ A$	(capability/protocol)
|	$\Gamma,\ X : \mathbf{type}$	(type assertion)	|	$\Delta^G,\ A_0; A_1 \triangleright \Delta$	(defocus-guarantee)

where Δ^G syntactically restricts Δ to not include a defocus-guarantee (a sharing feature, see Section 4.3). Suffices to note that this restriction ensures that defocus-guarantees are nested on the right of $\triangleright$ and that, at each level, there exists only one pending defocus-guarantee. Δ^G is also used to forbid capture of defocus-guarantees by functions and other constructs that can keep part of the linear typing environment for themselves.

The main typing rules are shown in Fig. 3, but the last four typing rules are only discussed in Section 4. All values (which includes functions, tagged values, etc.) have no resulting effect ($\cdot$) since, operationally, they have no pending computations. Allocating a new cell results in a type, $\exists t.(\ \mathbf{ref}\ t :: \mathbf{rw}\ t\ A\ )$, that abstracts the fresh location that was created (t), and includes both a reference to that location and the capability to that location. To associate a value (such as $\mathbf{ref}\ t$) with some capability (such as the capability to access location t), we use a *stacking* operator ::. Naturally, to be able to use the existential location, we must first open that abstraction by giving it a *location variable* to refer the abstracted location, besides the usual variable to refer the contents of the existential type. Reading the content of a cell can be either destructive or not, depending on whether its content is pure (!). If it is linear, then to preserve linearity we must leave the unit type ([]) behind to avoid duplication. By banging the type of a variable binding, we can move it to the linear context which enables the function's typing rule to initially consider all arguments as linear even if they are pure. Functions can only capture a Δ^G linear environment to ensure that they will not hide a pending defocus-guarantee

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

(T:Ref) $\dfrac{}{\Gamma, \rho : \mathbf{loc} \mid \cdot \vdash \rho : \mathbf{ref}\ \rho \dashv \cdot}$ (T:Unit) $\dfrac{}{\Gamma \mid \cdot \vdash v : [] \dashv \cdot}$ (T:Pure-Read) $\dfrac{}{\Gamma, x : A \mid \cdot \vdash x : !A \dashv \cdot}$ (T:Linear-Read) $\dfrac{}{\Gamma \mid x : A \vdash x : A \dashv \cdot}$

(T:Pure) $\dfrac{\Gamma \mid \cdot \vdash v : A \dashv \cdot}{\Gamma \mid \cdot \vdash v : !A \dashv \cdot}$ (T:Pure-Elim) $\dfrac{\Gamma, x : A_0 \mid \Delta_0 \vdash e : A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : !A_0 \vdash e : A_1 \dashv \Delta_1}$ (T:Tag) $\dfrac{\Gamma \mid \Delta \vdash v : A \dashv \cdot}{\Gamma \mid \Delta \vdash \mathbb{1}\#v : \mathbb{1}\#A \dashv \cdot}$ (T:Loc-Pack) $\dfrac{\Gamma \mid \Delta \vdash v : A\{p/t\} \dashv \cdot}{\Gamma \mid \Delta \vdash \langle p, v \rangle : \exists t.A \dashv \cdot}$

(T:New) $\dfrac{\Gamma \mid \Delta_0 \vdash v : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{new}\ v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}$ (T:Delete) $\dfrac{\Gamma \mid \Delta_0 \vdash v : \exists t.(\mathbf{ref}\ t :: \mathbf{rw}\ t\ A) \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash \mathsf{delete}\ v : \exists t.A \dashv \Delta_1}$

(T:Function) $\dfrac{\Gamma \mid \Delta^G, x : A_0 \vdash e : A_1 \dashv \cdot}{\Gamma \mid \Delta^G \vdash \mathsf{fun}(x : A_0).e : A_0 \multimap A_1 \dashv \cdot}$ (T:Application) $\dfrac{\Gamma \mid \Delta_0 \vdash v_0 : A_0 \multimap A_1 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_1 : A_0 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash v_0\ v_1 : A_1 \dashv \Delta_2}$

(T:Dereference-Pure) $\dfrac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ !A}{\Gamma \mid \Delta_0 \vdash !v : !A \dashv \Delta_1, \mathbf{rw}\ p\ !A}$ (T:Dereference-Linear) $\dfrac{\Gamma \mid \Delta_0 \vdash v : \mathbf{ref}\ p \dashv \Delta_1, \mathbf{rw}\ p\ A}{\Gamma \mid \Delta_0 \vdash !v : A \dashv \Delta_1, \mathbf{rw}\ p\ []}$ (T:Assign) $\dfrac{\Gamma \mid \Delta_0 \vdash v_1 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash v_0 : \mathbf{ref}\ p \dashv \Delta_2, \mathbf{rw}\ p\ A_1}{\Gamma \mid \Delta_0 \vdash v_0 := v_1 : A_1 \dashv \Delta_2, \mathbf{rw}\ p\ A_0}$

(T:Alternative-Left) $\dfrac{\Gamma \mid \Delta_0, A_0 \vdash e : A_2 \dashv \Delta_1 \qquad \Gamma \mid \Delta_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, A_0 \oplus A_1 \vdash e : A_2 \dashv \Delta_1}$ (T:Intersection-Right) $\dfrac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \qquad \Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_2}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1 \& A_2}$ (T:Case) $\dfrac{\Gamma \mid \Delta_0 \vdash v : \sum_i \mathbb{1}_i \# A_i \dashv \Delta_1 \qquad \overline{\Gamma \mid \Delta_1, x_i : A_i \vdash e_i : A \dashv \Delta_2} \qquad i \le j}{\Gamma \mid \Delta_0 \vdash \mathsf{case}\ v\ \mathsf{of}\ \overline{\mathbb{1}_j \# x_j \to e_j}\ \mathsf{end} : A \dashv \Delta_2}$

(T:Loc-App) $\dfrac{p : \mathbf{loc} \in \Gamma \qquad \Gamma \mid \Delta_0 \vdash v : \forall t.A \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash v[p] : A\{p/t\} \dashv \Delta_1}$ (T:Forall-Loc) $\dfrac{\Gamma, t : \mathbf{loc} \mid \Delta^G \vdash e : A \dashv \cdot}{\Gamma \mid \Delta^G \vdash \langle t \rangle\, e : \forall t.A \dashv \cdot}$ (T:Loc-Open) $\dfrac{\Gamma \mid \Delta_0 \vdash v : \exists t.A_0 \dashv \Delta_1 \qquad \Gamma, t : \mathbf{loc} \mid \Delta_1, x : A_0 \vdash e : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{open}\ \langle t, x \rangle = v\ \mathsf{in}\ e\ \mathsf{end} : A_1 \dashv \Delta_2}$

(T:Let) $\dfrac{\Gamma \mid \Delta_0 \vdash e_0 : A_0 \dashv \Delta_1 \qquad \Gamma \mid \Delta_1, x : A_0 \vdash e_1 : A_1 \dashv \Delta_2}{\Gamma \mid \Delta_0 \vdash \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1\ \mathsf{end} : A_1 \dashv \Delta_2}$ (T:Subsumption) $\dfrac{\Delta_0 <: \Delta_1 \qquad \Gamma \mid \Delta_1 \vdash e : A_0 \dashv \Delta_2 \qquad A_0 <: A_1 \qquad \Delta_2 <: \Delta_3}{\Gamma \mid \Delta_0 \vdash e : A_1 \dashv \Delta_3}$

(T:Cap-Elim) $\dfrac{\Gamma \mid \Delta_0, x : A_0, A_1 \vdash e : A_2 \dashv \Delta_1}{\Gamma \mid \Delta_0, x : A_0 :: A_1 \vdash e : A_2 \dashv \Delta_1}$ (T:Cap-Stack) $\dfrac{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}$ (T:Cap-Unstack) $\dfrac{\Gamma \mid \Delta_0 \vdash e : A_0 :: A_1 \dashv \Delta_1}{\Gamma \mid \Delta_0 \vdash e : A_0 \dashv \Delta_1, A_1}$

(T:Focus-Rely) $\dfrac{A_0 \in \overline{A}}{\Gamma \mid A_0 \Rightarrow A_1 \vdash \mathsf{focus}\ \overline{A} : [] \dashv A_0, A_1 \triangleright \cdot}$ (T:Defocus-Guarantee) $\dfrac{}{\Gamma \mid \Delta_0, A_0, A_0; A_1 \triangleright \Delta_1 \vdash \mathsf{defocus} : [] \dashv \Delta_0, A_1, \Delta_1}$

(T:Frame) $\dfrac{\Gamma \mid \Delta_0 \vdash e : A \dashv \Delta_1}{\Gamma \mid \Delta_0 \circledast \Delta_2 \vdash e : A \dashv \Delta_1 \circledast \Delta_2}$ (T:Share) $\dfrac{A_0 \Rrightarrow A_1 \parallel A_2}{\Gamma \mid \Delta, A_0 \vdash \mathsf{share}\ A_0\ \mathsf{as}\ A_1 \parallel A_2 : [] \dashv \Delta, A_1, A_2}$

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

Fig. 3. Static semantics (selected typing rules, see [21] for the rest).

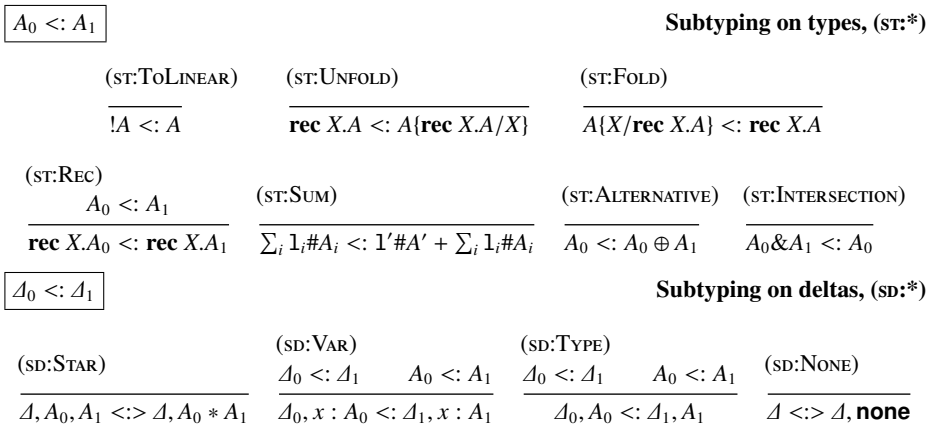

Fig. 4. Subtyping rules (selected, see [21] for the rest).

(and similarly on ∀ abstractions), since our types do not express such pending operation. *Stacking*, done through (T:Cap-Elim), (T:Cap-Stack) and (T:Cap-Unstack) enables the type system to manage capabilities in a non-syntax directed way, since they have no value nor associated identifier. The (T:Case) rule allows the set of tags of the value that is to be case analyzed (v) to be *smaller* than those listed in the branches of the case ($i \leq j$). This conditions is safe because it amounts to ignoring the effects of those branches, instead of being overly conservative and having to consider them all. These branches are not necessarily useless since, for instance, they may still be relevant on alternative program states (⊕). (T:Alternative-Left) expresses that if an expression types with both assumptions, A_0 and A_1, then it works with both alternatives. (T:Intersection-Right) is similar but on the resulting effect of that expression.

Finally, (T:Subsumption) enables expressions to rely on weaker assumptions while ensuring a stronger result than needed. This rule is supported by subtyping rules (a selection is shown in Fig. 4) that follow the form $A_0 <: A_1$ stating that A_0 is a subtype of A_1, meaning that A_0 can be used wherever A_1 is expected. Similar meaning is used for subtyping on linear typing environments, $\Delta_0 <: \Delta_1$. Among other operations, these rules enable automatic fold/unfold of recursive types, as well as grouping ($*$) of resources.

4 Sharing Mutable State

The goal is to enable reads and writes to a cell through multiple aliases, without requiring the type system to precisely track the link between aliased variables. In other words, the type system is aware that a variable is aliased, but does not know exactly which other variables alias that same state. In this scenario, it is no longer possible to implicitly move capabilities between aliases. Instead, we split the original capability into multiple *protocol* capabilities to that same location, and ensure that these multiple protocols cannot interact in ways that destructively interfere with each other. Such *rely-guarantee* protocol accounts for the effects of other protocols (the *rely*), and limits the

actions of this protocol to *guarantee* that they do not contradict the assumptions relied on by other aliases. This allows independent, but constrained, actions on the different protocols to the same shared state without destructive interference. However, it also requires us to leverage additional type mechanisms to ensure safety, namely:

(a) **Hide intermediate states.** A rely-guarantee protocol restricts how aliases can use the shared state. However, we allow such specification to be temporarily broken provided that all unexpected changes are private, invisible to other aliases. Therefore, the type system ensures a kind of static mutual exclusion, a mechanism that provides a "critical section" with the desired level of isolation from other aliases to that same state. Consequently, other shared state that may overlap with the one being inspected simply becomes unavailable while that cell is undergoing private changes. Although this solution is necessarily conservative, we avoid any run-time overhead while preserving many relevant usages. To achieve this, we build on the concept of focus [11] (in a non-lexically scoped style, so that there is also a defocus) clearly delimiting the boundary in the code of where shared state is being inspected. Thus, on focus, all other types that may directly or indirectly see inconsistencies must be temporarily concealed only to reappear when those inconsistencies have been fixed, on defocus.

(b) **Ensure that each individual step of the protocol is obeyed.** In our system, sharing properties are encoded in a protocol composed of several rely-guarantee *steps*. As discussed in the previous paragraph, each step must be guarded by focus since private states should not be visible to other aliases. Consequently, the focus construct serves not only to safeguard from interference by other aliases, but also to move the protocol forward through each of its individual steps. At each such step, the code can assume on entry (focus) that the shared state will be in a given well-defined *rely* state, and must ensure on exit (defocus) that the shared state satisfies a given well-defined *guarantee* state. By characterizing the sequence of actions of each alias with an appropriate protocol, one can make strong local assumptions about how the shared state is used without any explicit dependence on how accesses to other aliases of that shared state are interleaved. This feature is crucial since we cannot know precisely if that same shared state was used between two focus-defocus operations.

4.1 Specifying Rely-Guarantee Protocols

We now detail our rely and guarantee types that are the building blocks of our protocols. To clarify the type structure of our protocols, we define the following sub-grammar of our types syntax (Fig. 2) with the types that may appear in a protocol, P.

$$P \ ::= \ \mathbf{rec}\ X.P \mid X \mid P \oplus P \mid P \,\&\, P \mid A \Rightarrow P \mid A; P \mid \mathbf{none}$$

A rely-guarantee protocol is a type of capability (i.e. has no value) consisting of potentially many steps, each of the form $A_C \Rightarrow A_P$. Each such step states that it is safe for the current client to assume that the shared state satisfies A_C and is required to obey the guarantee A_P, usually of the form $A'_C; A'_P$ which in turn requires the client to establish (guarantee) that the shared state satisfies A'_C before allowing the protocol to continue to be used as A'_P. Note that our design constrains the syntactical structure of these protocols through *protocol conformance* (Section 4.2), not in the grammar.

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

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

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

This protocol expresses that the client code can safely assume (on focus) a capability stating that location p initially holds type `Empty#[]`. It then requires the code that uses such state to leave it (on defocus) in one of two possible alternatives ($\oplus$) depending on whether the producer chooses to close the pipe or insert a new element to the buffer. To signal that the node is the last element of the pipe, the producer can just assign it a value of type `Closed#[]`. Insertions are slightly more complicated because that action implies that the tail element of the list will be changed. Therefore, after creating the new node, the producer component will keep an alias of the new tail for itself while leaving the old tail with a type that is to be used by the consumer. In this case, the node is assigned a value of type `Node#R`, where R denotes the type $[\ \mathsf{int}\ ,\ \exists p.(\ \textbf{ref}\ p\ ::\ \mathsf{H}[p]\ )\ ]$ (a pair of an integer and a reference to the next shared node of the buffer, as seen from the head pointer). Regardless of its action, the producer then forfeits any ownership of that state which is modeled by the empty capability (**none**)[5] to signal protocol termination.

We now present the abbreviations H and T, the rely-guarantee protocols that govern the use of the shared state of the pipe as seen by the `head` and `tail` aliases, respectively. Note that since we intend to apply the same protocol over different locations, we use "$Q \triangleq \forall p.A$" as a type definition (Q) where we can apply a location without requiring $\forall$ to be a value, such as location q in $Q[q]$. The T and H types are defined as follows:

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

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

Each protocol describes a partial view of the complete use of the shared state. Consequently, ensuring their safety cannot be done alone. In our system, protocols are introduced explicitly through the share construct that declares that a type (in practice limited

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

$\boxed{\langle A, P\rangle \to \langle A', P'\rangle}$ **Step, (STEP:*)**

$$\frac{}{\langle A, \mathbf{none}\rangle \to \langle A, \mathbf{none}\rangle}\ (\text{STEP:NONE}) \qquad \frac{}{\langle A_0, A_0 \Rightarrow A_1; P\rangle \to \langle A_1, P\rangle}\ (\text{STEP:STEP})$$

$$\frac{\langle A_0, P_0\rangle \to \langle A_1, P_2\rangle}{\langle A_0, P_0 \oplus P_1\rangle \to \langle A_1, P_2\rangle}\ (\text{STEP:ALTERNATIVE-P}) \qquad \frac{\langle A_0, P_0\rangle \to \langle A_2, P_1\rangle \qquad \langle A_1, P_0\rangle \to \langle A_2, P_1\rangle}{\langle A_0 \oplus A_1, P_0\rangle \to \langle A_2, P_1\rangle}\ (\text{STEP:ALTERNATIVE-S})$$

$$\frac{A_0 <: A_1 \qquad P_0 <: P_1 \qquad \langle A_1, P_1\rangle \to \langle A_2, P_2\rangle \qquad A_2 <: A_3 \qquad P_2 <: P_3}{\langle A_0, P_0\rangle \to \langle A_3, P_3\rangle}\ (\text{STEP:SUBSUMPTION})$$

Fig. 5. Protocol stepping rules.

to capabilities, including protocols) is to be split in two new rely-guarantee protocols. Safety is checked by simulating their actions in order to ensure that they preserve the overall consistency in the use of the shared state, no matter how their actions may be interleaved. Since a rely-guarantee protocol can subsequently continue to be split, this technique does not limit the number of aliases provided that the protocols conform.

4.2 Checking Protocol Splitting

The key principle of ensuring a correct protocol split is to verify that both protocols consider all visible states that are reachable by stepping, ensuring a form of progress. Protocols are not required to always terminate and may be used indefinitely, for instance when modeling invariant-based sharing. However, regardless of interleaving or of how many times a shared alias is (consecutively) used, no unexpected state can ever appear in well-formed protocols. Thus, the type information contained in a protocol is valid regardless of all interference that may occur, i.e. it is *stable* [17, 32].

Technically, the correctness of protocol splitting is ensured by two key components: 1) a *stepping* relation, that simulates a single use of the shared state through one focus-defocus block; and 2) a *protocol conformance* definition, that ensures full coverage of all reachable states by considering all possible interleaved uses of those steps. Thus, even as the rely and guarantee conditions evolve through the protocol's lifetime, protocol conformance ensures each protocol will never get "stuck" because the protocol must be aware of all possible "alias interleaving" that may occur for that state.

The stepping relation (Fig. 5) uses steps of the form $\langle A, P\rangle \to \langle A', P'\rangle$ expressing that, assuming shared state A, the protocol P can take a step to shared state A' with residual protocol P'. Due to the use of $\oplus$ and & types in the protocols, there may be *multiple* different steps that may be valid at a given point in that protocol. Therefore, protocol conformance must account for *all* those different transitions that may be picked.

We define protocol conformance as splitting an existing protocol (or capability) in two, although it can also be interpreted as merging two protocols. Regardless of the direction, the actions of the original protocol(s) must be fully contained in the resulting protocol(s). This leads to the three stepping conditions of the definition below.

Definition 1 (Protocol Conformance). Given an initial state A_0 and a protocol γ_0, such protocol can be split in two new protocols α_0 and β_0 if their combined actions conform with those of the original protocol γ_0, noted $\langle A_0\ ,\ \gamma_0 \iff \alpha_0 \parallel \beta_0\rangle$. This means that there is a set $\mathcal{S}$ of *configurations* $\langle A\ ,\ \gamma \iff \alpha \parallel \beta\rangle$ closed under the conditions:

1. The initial configuration is in $\mathcal{S}$: $\langle A_0\ ,\ \gamma_0 \iff \alpha_0 \parallel \beta_0\rangle \in \mathcal{S}$
2. All configurations take a step, and the result is also in $\mathcal{S}$.
   Therefore, if $\langle A\ ,\ \gamma \iff \alpha \parallel \beta\rangle \in \mathcal{S}$ then:
   (a) exists A', α' such that $\langle A, \alpha\rangle \to \langle A', \alpha'\rangle$, and for all A', α', $\langle A, \alpha\rangle \to \langle A', \alpha'\rangle$ implies $\langle A, \gamma\rangle \to \langle A', \gamma'\rangle$ and $\langle A'\ ,\ \gamma' \iff \alpha' \parallel \beta\rangle \in \mathcal{S}$.
   (b) exists A', β' such that $\langle A, \beta\rangle \to \langle A', \beta'\rangle$, and for all A', β', $\langle A, \beta\rangle \to \langle A', \beta'\rangle$ implies $\langle A, \gamma\rangle \to \langle A', \gamma'\rangle$ and $\langle A'\ ,\ \gamma' \iff \alpha \parallel \beta'\rangle \in \mathcal{S}$.
   (c) exists A', γ' such that $\langle A, \gamma\rangle \to \langle A', \gamma'\rangle$, and for all A', γ', $\langle A, \gamma\rangle \to \langle A', \gamma'\rangle$ implies either:
   - $\langle A, \alpha\rangle \to \langle A', \alpha'\rangle$ and $\langle A'\ ,\ \gamma' \iff \alpha' \parallel \beta\rangle \in \mathcal{S}$, or;
   - $\langle A, \beta\rangle \to \langle A', \beta'\rangle$ and $\langle A'\ ,\ \gamma' \iff \alpha \parallel \beta'\rangle \in \mathcal{S}$.

The definition yields that all configurations must step (i.e. never get stuck) and that a step in one of the protocols (α or β) must also step the original protocol (γ) such that the result itself still conforms. Conformance ensures that all interleavings are coherent. This also means that each protocol "view" of the shared state can work independently in a safe way — even when the other aliases to that shared state are never used. Ownership recovery does not require any special treatment since it just expresses that the focused capability is not returned back to the protocol, enabling it to remain in the local context.

We now apply protocol conformance to our running example, as follows:

$A : E$
$\gamma : \mathbf{rec}\ X.(\ E \Rightarrow E; X\ \&\ (\ E \Rightarrow N \oplus C\ ;\ (\ N \Rightarrow \mathbf{none}\ \oplus\ C \Rightarrow \mathbf{none}\ )\ )\ )$
$\alpha : E \Rightarrow N \oplus C$ (Tail protocol)
$\beta : \mathbf{rec}\ X.(\ E \Rightarrow E; X\ \oplus\ N \Rightarrow \mathbf{none}\ \oplus\ C \Rightarrow \mathbf{none}\ )$ (Head protocol)

Therefore, applying the definition yields the following set of configurations, $\mathcal{S}$:

$\langle E\ ,\ \mathbf{rec}\ X.(\ E \Rightarrow E; X\ \&\ (\ E \Rightarrow N \oplus C; (\ N \Rightarrow \mathbf{none}\ \oplus\ C \Rightarrow \mathbf{none}\ )\ )\ )\ \iff$
$E \Rightarrow C \oplus N\ \parallel\ \mathbf{rec}\ X.(\ E \Rightarrow E; X\ \oplus\ N \Rightarrow \mathbf{none}\ \oplus\ C \Rightarrow \mathbf{none}\ )\rangle$ (1)

The initial configuration.

by step on γ (subtyping for &) with $E \Rightarrow E; X$ and same with β, using (STEP:ALTERNATIVE-P).

$\langle N \oplus C\ ,\ N \Rightarrow \mathbf{none}\ \oplus\ C \Rightarrow \mathbf{none}\ \iff$
$\mathbf{none}\ \parallel\ \mathbf{rec}\ X.(\ E \Rightarrow E; X \oplus N \Rightarrow \mathbf{none} \oplus C \Rightarrow \mathbf{none}\ )\rangle$ (2)

by step on (1) with γ (subtyping for &) with $E \Rightarrow N \oplus C$; ... and similarly using α.

$\langle \mathbf{none}\ ,\ \mathbf{none}\ \iff\ \mathbf{none} \parallel \mathbf{none}\rangle$ (3)

by step on (2) with γ and β using (STEP:ALTERNATIVE-S).

$\mathcal{S}$ is closed (up to subtyping, including unfolding of recursive types).

Regardless of how the use of the state is interleaved at run-time, the shared state cannot reach an unexpected (by the protocols) state. Thus, conformance ensures the stability of the type information contained in a protocol in the face of all possible "alias interleaving". There exists only a finite number of possible (relevant) states, meaning that it suffices for protocol conformance to consider the smallest set of configurations

that obeys the conditions above. Since there is also a finite number of possible interleavings resulting from mixing the steps of the two protocols, there are also a finite number of distinct (relevant) steps. Effectively, protocol conformance resembles a form of bisimulation or model checking (where each protocol is modeled using a graph) with a finite number of states, ensuring such process remains tractable.

In the following text we use a simplified notation, of the form $A \Rrightarrow A' \parallel A''$, as an idiom (defined in [21]) that applies protocol conformance uniformly regardless of whether A is a state (for an initial split) or a rely-guarantee protocol (to be re-split and perhaps extended). The missing type is inferred by this idiom.

Example We illustrate these concepts by going back to the pipe's protocols. We introduced the protocols for the head and tail aliases through the share construct:

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

which is checked by the (T:SHARE) typing rule, using protocol conformance, as follows:

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

With it we share a capability (A_0) by splitting it in two protocols (A_1 and A_2) whose individual roles in the interactions with that state conform ($\Rrightarrow$). Consequently, the conclusion states that, if the splitting is correct, then in some linear typing environment initially consisting of a type A_0 and Δ, the share construct produces effects that replace A_0 with A_1 and A_2 but leave Δ unmodified (i.e. it is just threaded through).

The next examples show conformance in a simplified way, with only the state and the two resulting protocols of a configuration. Remember that E is the abbreviation for **rw** q `Empty#[]` that, just like the abbreviations C and N, were defined above. Thus, the use of the share construct on line 3 yields the following set of configurations, $\mathcal{S}$:

$$\langle \mathsf{E} \Rrightarrow \mathbf{rec}\ X.(\ \mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\ ;\ X\ ) \parallel \mathsf{E} \Rightarrow (\ \mathsf{N} \oplus \mathsf{C}\ )\rangle \quad (1)$$
$$\langle \mathsf{N} \oplus \mathsf{C} \Rrightarrow \mathbf{rec}\ X.(\ \mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\ ;\ X\ ) \parallel \mathbf{none}\rangle \quad (2)$$
$$\langle \mathbf{none} \Rrightarrow \mathbf{none} \parallel \mathbf{none}\rangle \quad (3)$$

The definition is only respected if E is the state to be shared by the protocols. If instead we had shared, for instance, C we would get the next set of configurations:

$$\langle \mathsf{C} \Rrightarrow \mathbf{rec}\ X.(\ \mathsf{N} \Rightarrow \mathbf{none} \oplus \mathsf{C} \Rightarrow \mathbf{none} \oplus \mathsf{E} \Rightarrow \mathsf{E}\ ;\ X\ ) \parallel \mathsf{E} \Rightarrow (\ \mathsf{N} \oplus \mathsf{C}\ )\rangle \quad (1)$$
$$\langle \mathbf{none} \Rrightarrow \mathbf{none} \parallel \mathsf{E} \Rightarrow (\ \mathsf{N} \oplus \mathsf{C}\ )\rangle \quad (2)$$

The set above does not satisfy our conformance definition. Both the state in configuration (1) and **none** in (2) are not expected by the right protocol. Thus, those configurations are "stuck" and cannot take a step. Although splittings are checked from a high-level and abstracted perspective, their consequences link back to concrete invalid program states that could occur if such invalid splittings were allowed. For instance, in (2), it would imply that the alias that used the right protocol would assume E on focus long after the ownership of that state was recovered by some other alias of that cell. Consequently, such behavior could allow unexpected changes to be observed by that alias, potentially resulting in a program stuck on some unexpected value.

4.3 Using Shared State

Using shared state is centered on two constructs: focus (that exposes the shared state of a protocol) and defocus (that returns the exposed state to the protocol), combined with our version of the *frame rule* (Section 4.4). We now describe how focus is checked:

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

In general, focus may be applied over a disjunction ($\oplus$) of program states and expected to work on any of those alternatives. By using $\overline{A}$, the programmer can list the types that may become available after focus, nominating what they expect to gain by focus.

focus results in a typing environment where the step of the protocol that was focused on ($A_0 \Rightarrow A_1$) now has its rely type (A_0) available to use. However, it is not enough to just make that capability available, we must also *hide* all other linear resources that may use that same shared state (directly or indirectly) in order to avoid interference due to the inspection of private states. To express this form of hiding, the linear typing environments may include a *defocus-guarantee*. This element, written as $A \triangleright \Delta$, means that we are hiding the typing environment Δ until A is satisfied. Therefore, in our system, the only meaningful type for A is a guarantee type of the form $A'; A''$ that is satisfied when A' is offered and enables the protocol to continue to be use as A''. Although the typing rule shown above only includes a single element in the initial typing environment (and, consequently, the defocus-guarantee contains the empty typing environment, $\cdot$), this is not a limitation. In fact, the full potential of (t:Focus-Rely) is only realized when combined with (t:Frame). Together they allow for the non-lexically scoped framing of potentially shared state, where the addition of resources that may conflict with focused state will be automatically nested inside the defocus-guarantee ($\triangleright$). Operationally share, focus, and defocus are no-ops which results in those expressions having type unit ([]).

$$\frac{}{\Gamma \mid \Delta_0,\ A',\ A'; A'' \triangleright \Delta_1 \vdash \mathsf{defocus} : [] \dashv \Delta_0,\ A'',\ \Delta_1}\ (\textsc{t:Defocus-Guarantee})$$

The complementary operation, defocus, simply checks that the required guarantee type (A') is present. In that situation, the typing environment (Δ_1) that was hidden on the right of $\triangleright$ can now safely be made available once again. At the same time, the step of the protocol is concluded leaving the remainder protocol (A'') in the typing environment. Nesting of defocus-guarantees is possible, but is only allowed to occur on the right of $\triangleright$. Note that defocus-guarantees can never be captured (such as by functions, see Fig. 3 of Section 3) and, therefore, pending defocus operations cannot be forgotten or ignored.

Example We now look at the implementation of the `put` and `close` functions to exemplify the use of focus and defocus. Both functions are closures that capture an enclosing Γ where t is a known location such that `tail` has type **ref** t. T was defined above as: $\forall p.$(**rw** p `Empty#`[] $\Rightarrow$ **rw** p `Node#`R $\oplus$ **rw** p `Closed#`[]) where R is a pair of an integer and a protocol for the head, H (whose definition, given above, is not important here).

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

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

4.4 Framing State

On its own, (T:Focus-Rely) is very restrictive since it requires a single rely-guarantee protocol to be the exclusive member of the linear typing environment. This happens because more complex applications of focus are meant to be combined with our version of the frame rule. Together they enable a kind of mutual exclusion that also ensures that the addition of any potentially interfering resources will forcefully be on the right of ▷ (thus making them inaccessible until defocus). The typing rule is as follows:

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

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

not enable destructive interference. Therefore, types that may refer (directly or indirectly) values that access shared cells that are currently inconsistent due to pending defocus cannot be accessible and must be placed "inside" (on the right of $\triangleright$) the defocus-guarantee. However, statically, we can only make such distinction conservatively by only allowing types that are **non-shared** (and therefore that are known to never conflict with other shared state) to not be placed inside the defocus-guarantee. The formal definition of **non-shared** is in [21], but for this presentation it is sufficient to consider it as pure types, or capabilities (**rw** $p\ A$) that are not rely-guarantee protocols and that whose contents are also non-shared. This means that all other linear types (even abstracted capabilities and linear functions) must be assumed to be potential sources of conflicting interference. For instance, these types could be abstracting or capturing a rely-guarantee protocol that could then result in a re-entrant inspection of the shared state.

To build the extended typing environment, we define an *environment extension* ($\circledast$–) operation that takes into account frame defocus-guarantees up to a certain depth. This means that one can always consider extensions of the current footprint as long as any added shared state is hidden from all focused state. By conservatively hiding it behind a defocus-guarantee, we ensure that such state cannot be touched. This enables locality on focus: if a protocol is available, then it can safely be focused on.

Definition 2 (Environment Extension). Given environments Δ and Δ' we define environment extension, noted $\Delta \circledast \!\!- \Delta'$, as follows. Let $\Delta = \Delta_n, \Delta_s$ where n-indexed environments only contains ***n*on-shared** elements and s-indexed environments contain the remaining elements (i.e. all those that may, potentially, include *s*haring). Identically, assume $\Delta' = \Delta'_n, \Delta'_s$. Extending Δ with Δ' corresponds to $\Delta \circledast \!\!- \Delta' = \Delta_n, \Delta'_n, \Delta''_s$ where:

(a) $\Delta''_s = \Delta_{s_0}, A \triangleright (\ \Delta_{s_1} \circledast \!\!- \Delta'_s\ )$ if $\Delta_s = \Delta_{s_0}, A \triangleright \Delta_{s_1}$
(b) $\Delta''_s = \Delta_s, \Delta'_s$ otherwise.

that either (a) further nests the shared part of Δ' deeper in Δ_{s_1}; or (b) simply composes Δ' if the left typing environment (Δ) does not carry a defocus-guarantee.

Although the definition appears complex, it works just like regular environment composition when Δ' does not contain a defocus-guarantee, i.e. the (b) case. The complexity of the definition arises from the need to nest these structures when they do exist, which results in the inductive definition above. In that situation, we must ensure that any potentially interfering shared state is placed deep inside all previously existing defocus-guarantees, so as to remain inaccessible. This definition is compatible with the basic notion of disjoint separation, but (from a framing perspective) allows us to frame-away defocus-guarantees beyond a certain depth. Such state can be safely hidden if the underlying expression will not reach it (by defocusing).

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

Example We now look back at the focus of line 12. To better illustrate framing, we consider an extra linear type (that is *not* **non-shared**), S, to show how it will become hidden (on the right of $\triangleright$) after focus. We also abbreviate the two non-shared capabilities ("**rw** t []" and "**rw** l Empty#[]")[6] as A_0 and A_1, and abbreviate the protocol so that it does not show the type application of location o. With this, we get the following derivation:

$$\dfrac{\dfrac{\dfrac{\mathsf{E} \in \mathsf{E}}{\Gamma \mid \mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C}) \vdash \mathsf{focus}\ \mathsf{E} : [] \dashv \mathsf{E}, (\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright \cdot}\ (3)}{\Gamma \mid (\mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C})) \circledast\!\!- S, A_0, A_1 \vdash \mathsf{focus}\ \mathsf{E} : [] \dashv (\mathsf{E}, (\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright \cdot) \circledast\!\!- S, A_0, A_1}\ (2)}{\Gamma \mid \mathsf{E} \Rightarrow (\mathsf{N} \oplus \mathsf{C}), S, A_0, A_1 \vdash \mathsf{focus}\ \mathsf{E} : [] \dashv \mathsf{E}, ((\mathsf{N} \oplus \mathsf{C}); \mathbf{none} \triangleright S), A_0, A_1}\ (1)$$

where (1) - (Environment Extension), (2) - (T:Frame), and (3) - (T:Focus-Rely).

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

4.5 Consumer code

We now show the last function of the pipe example, `tryTake`:

```
26 tryTake = fun( _ [] :: rw h exists p.(ref p :: H[p]) ).                    Δ = rw h ∃p.(ref p :: H[p])
27   open <f,first> = !head in          Δ = rw h [] , (N[f] ⇒ none) ⊕ (C[f] ⇒ none) ⊕ (E[f] ⇒ E[f]) ; ...)
         [a] Δ = rw h [], N[f] ⇒ none  [b] Δ = rw h [], C[f] ⇒ none  [c] Δ = rw h [], E[f] ⇒ E[f] ; ...
28     focus C[f], E[f], N[f]; // same abbreviations that were defined above
           [a] Δ = ..., N[f], none;none ▷ ·  [b] Δ = ..., C[f], none;none ▷ ·  [c] Δ = ..., E[f], E[f] ; ... ▷ ·
29     case !first of
30       Empty#_ →                                                  [c] Δ = rw h [] , rw f [] , rw f Empty#[]; ... ▷ ·
31         first := Empty#{}; // restore linear type          [c] Δ = rw h [] , rw f Empty#[] , rw f Empty#[]; ... ▷ ·
32         defocus; // the next assignment must occur after defocus and just on this branch  [c] Δ = rw h [] , H[f]
33         head := <f,first::H[f]>;                                    [c] Δ = rw h ∃p.(ref p :: H[p])
34         NoResult#{} : NoResult#([] :: rw h ∃p.(ref p :: H[p]))  //assume auto stacked      [c] Δ = ·
35     | Closed#_ →                                                [b] Δ = rw h [] , rw f [] , none;none ▷ ·
36         delete first;                                              [b] Δ = rw h [] , none;none ▷ ·
37         delete head;                                                  [b] Δ = none;none ▷ ·
38         defocus;                                                                [b] Δ = ·
39         Depleted#{} : Depleted#[]                                               [b] Δ = ·
40     | Node#[element,n] → //opens pair  [a] Δ = rw h [] , rw f [] , n : ∃p.(ref p :: H[p]) , none;none ▷ ·
41         delete first;                        [a] Δ = rw h [] , n : ∃p.(ref p :: H[p]) , none;none ▷ ·
42         head := n;                           [a] Δ = rw h ∃p.(ref p :: H[p]) , none;none ▷ ·
43         defocus;                                          [a] Δ = rw h ∃p.(ref p :: H[p])
44         Result#element : Result#(int :: rw h ∃p.(ref p :: H[p])) // assume auto stacked   [a] Δ = ·
45     end
46   end
```

The code should be straightforward up to the use of alternative program states ($\oplus$). This imprecise state means that we have one of several different alternative capabilities

[6] Note that the content of each capability can be made **non-shared** by subtyping rules.

and, consequently, the expression must consider all of those cases separately. On line 28, to use each individual alternative of the protocol, we check the expression separately on each alternative (marked as **[a]**, **[b]**, and **[c]** in the typing environments), cf. (T:ALTERNATIVE-LEFT) in Fig. 3. Our `case` gains precision by ignoring branches that are statically known to not be used. On line 29, when the type checker is case analyzing the contents of `first` on alternative **[b]** it obtains type `Closed#`[]. Therefore, for that alternative, type checking only examines the `Closed` tag and the respective case branch. This feature enables the `case` to obey different alternative program states simultaneously, although the effects/guarantee that each branch fulfills are incompatible.

5 Technical Results

Our soundness results (details in [21]) use the next progress and preservation theorems:

Theorem 1 (Progress). *If e_0 is a closed expression (and where Γ and Δ are also closed) such that $\Gamma \mid \Delta_0 \vdash e_0 : A \dashv \Delta_1$ then either:*

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

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

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

$$\Gamma_0 \mid \Delta_0 \vdash e_0 : A \dashv \Delta \qquad \Gamma_0 \mid \Delta_0 \circledast \Delta_2 \vdash H_0 \qquad \langle H_0 \parallel e_0 \rangle \mapsto \langle H_1 \parallel e_1 \rangle$$

then, for some Δ_1 and Γ_1 we have: $\Gamma_0, \Gamma_1 \mid \Delta_1 \circledast \Delta_2 \vdash H_1 \qquad \Gamma_0, \Gamma_1 \mid \Delta_1 \vdash e_1 : A \dashv \Delta$

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

We define *store typing* (see [21]), noted $\Gamma \mid \Delta \vdash H$, in a linear way so that each heap location must be matched by some capability in Δ or potentially many rely-guarantee protocols. Thus, no instrumentation is necessary to show these theorems.

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

6 Additional Examples

We now exemplify some sharing idioms captured by our rely-guarantee protocols.

6.1 Sharing a Linear ADT

Our protocols are capable of modeling monotonic [12,24] uses of shared state. To illustrate this, we use the linear stack ADT from [22] where the stack object has two possible typestates: Empty and Non-Empty. The object, with an initial typestate E(mpty), is accessible through closures returned by the following "constructor" function:

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

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

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

Monotonicity means that the type becomes successively more precise, although each alias does not know when those changes occurred. Note that, due to focus, the object can undergo intermediate states that are not compatible with the required NE guarantee. However, on defocus, clients must provide NE such as by pushing some element to the stack. The protocol itself can be repeatedly shared in equal protocols. Since each copy will produce the same effects as the original protocol, their existence is not observable.

6.2 Capturing Local Knowledge

Although our types cannot express the same amount of detail on local knowledge as prior work [4, 18], they are expressive enough to capture the underlying principle that enables us to keep increased precision on the shared state between steps of a protocol.

For this example, we use a simple two-states counter. In it, N encodes a number that may be zero and P some positive number, with the following relation between states:

$$\mathsf{N}\triangleq\mathtt{Z\#}[]+\mathtt{NZ\#int}\qquad\mathsf{P}\triangleq\mathtt{NZ\#int}\qquad\text{(note that: P <: N, vital to show conformance)}$$

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

$$
\begin{aligned}
\mathtt{IncOnly}&\triangleq\textbf{rec}\;X.(\;\mathsf{rw}\;p\;\mathsf{N}\Rightarrow\mathsf{rw}\;p\;\mathsf{P}\;;\;X\;)\\
\mathtt{Any}&\triangleq\textbf{rec}\;Y.(\;\mathsf{rw}\;p\;\mathsf{N}\Rightarrow\mathsf{rw}\;p\;\mathsf{P}\;;\;\mathsf{rw}\;p\;\mathsf{P}\Rightarrow\mathsf{rw}\;p\;\mathsf{N}\;;\;Y\;)
\end{aligned}
$$

Thus, by constraining the actions of IncOnly we can rely on the assumption that Any remains positive on its second step, even when the state is manipulated in some other unknown program context. Therefore, on the second step of Any, the case analysis can be sure that the value of the shared state must have remained with the NZ tag between focuses. Note that the actions of that alias allow for it to change the state back to Z.

6.3 Iteratively Sharing State

Our technique is able to match an arbitrary number of aliases by splitting an existing protocol. Such split can also extend the original uses of the shared state by appending additional steps, if those uses do not destructively interfere with the old assumptions.

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

We also show how we can share multiple locations together, bundled using $*$, by each job being kept in a *c*ontainer cell while the *f*lag (used to communicate the information on the kind of content stored in the container) is in a separate cell. The raw value is typed with A and the processed value has type B. The types and protocols are:

$$\mathsf{F} \triangleq \mathbf{rw}\ f\ \texttt{Free\#}[]\ *\ \mathbf{rw}\ c\ [] \qquad \mathsf{R} \triangleq \mathbf{rw}\ f\ \texttt{Raw\#}[]\ *\ \mathbf{rw}\ c\ A \qquad \mathsf{D} \triangleq \mathbf{rw}\ f\ \texttt{Done\#}[]\ *\ \mathbf{rw}\ c\ B$$

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

The protocol for the Receiver is straightforward since it just processes a free cell by assigning it a raw value. Similarly, Compressor and Storer follow analogous ideas by using a kind of "waiting" steps until the cell is placed with the desired type for the actions that they are to take (note how Storer keeps a more precise context when the state is not F, even though it is not allowed to publicly modify the state). To obtain these protocols through binary splits, we need an *intermediate* protocol that will be split to create the Compressor and Storer protocols. The initial split (of F) is as follows:

$$\mathsf{F}\ \Rrightarrow\ \mathsf{Receiver}\ \|\ \mathbf{rec}\ X.(\ \mathsf{F} \Rightarrow \mathsf{F}; X\ \oplus\ \mathsf{R} \Rightarrow \mathbf{none}\ )$$

The protocol on the right is then further split, and its ownership recovery step further extended with additional steps, to match the two new desired protocols:

$$\mathbf{rec}\ X.(\ \mathsf{F} \Rightarrow \mathsf{F}; X\ \oplus\ \mathbf{rec}\ Y.(\ \mathsf{R} \Rightarrow \mathsf{R}; Y\ \&\ \mathsf{R} \Rightarrow \mathsf{D}; \mathsf{D} \Rightarrow \mathbf{none}\ )\ )\ \Rrightarrow\ \mathsf{Compressor}\ \|\ \mathsf{Storer}$$

The Receiver alias never needs to see how the other two aliases use the shared state. Although the second split is independent from the initial one, protocol conformance ensures that it cannot cause interference by breaking what Receiver initially relied on.

7 Related Work

We now discuss other works that offer flexible sharing mechanisms. Although there are other interesting works [1, 2, 4, 5, 7, 31] in the area, they limit sharing to an invariant.

In *Chalice* [19], programmer-supplied permissions and predicates are used to show that a program is free of data races and deadlocks. A limited form of rely-guarantee

is used to reason about changes to the shared state that may occur between atomic sections. All changes from other threads must be expressed in auxiliary variables and be constrained to a two-state invariant that relates the *current* with the *previous* state, and where all rely and guarantee conditions are the same for all threads.

Several recent approaches that use advanced program logics [9, 10, 23, 30, 32] employ rely-guarantee reasoning to verify inter-thread interference. Although our approach is type-based rather than logic-based, there are several underlying similarities. *Concurrent abstract predicates* [9] extend the concept of *abstract predicates* [23] to express how state is manipulated, supporting internally aliased state through a *fiction of disjointness* (also present in [16, 18]) that is based on rely-guarantee principles and has similarities to our own abstractions. Their use of rely-guarantee also allows intermediate states within a critical section, which are immediately weakened (made stable) to account for possible interference when that critical section is left. Although our use of rely-guarantee is tied to state (be it references or abstracted state), not threads, our protocols capture an identical notion of stability through a simpler constraint that ensures all visible states are considered during protocol conformance. Another modeling distinction is that our interference specification lists the resulting states (from interference), not the actions that can (or cannot [10]) occur from external/unknown sources.

Monotonic [12, 24] based sharing enables unrestricted aliasing that cannot interfere since the changes converge to narrower, more precise, states. Our protocols are able to express monotonicity. However, since the rely and guarantee types of a step in the protocol must describe a finite number of states, we lack the type expressiveness of [24]. We believe this concern is orthogonal to our core sharing concepts, and is left as future work. We are also capable of expressing more than just monotonicity. For instance, due to ownership recovery, a cell can oscillate between shared and non-shared states during its lifetime, and with each sharing phase completely unrelated to previous uses.

Gordon *et al.* [15] propose a type system where references carry three additional type components: a predicate (for local knowledge), a guarantee relation, and a rely relation. They handle an unknown number of aliases by constraining the writes to a cell to fit within the alias' declared guarantee, similarly to how rely-guarantee is used in program logics to handle thread-based interference. Although they support a limited form of protocol (and their technique can generally be considered as a two-state protocol), their system effectively limits the actions allowed by each new alias to be strictly decreasing since their guarantee must fit within the original alias' guarantee. Since we support ownership recovery of shared state, a cell can be shared and return to non-shared without such restriction. Unlike ours, their work does not allow intermediate inconsistent states since all updates are publicly visible. In addition, their work requires proof obligations for, among other things, guarantee satisfaction while we use a more straightforward definition of protocol conformance that is not dependent on theorem-proving. However, their use of dependent refinement types adds expressiveness (e.g. their predicates capture an infinite state space, while our state space is finite) but increases the challenges in automation, as typechecking requires manual assistance in Coq.

Krishnaswami *et al.* [18] define a generic sharing rule based on the use of frame-preserving operations over a commutative monoid (later shown to be able to encode rely-guarantee [8]). The core principle is centered on splitting the internal resources of

an ADT such that all aliases obey an invariant that is shared, while also keeping some knowledge about the locally-owned shared state. By applying a frame condition over its specification, their shared resources ensure that any interference between clients is benign since it preserves the fiction of disjointness. Thus, local assumptions can interact with the shared state without being affected by the actions done through other aliases of that shared state. The richness of their specification language means that although it might not always be an obvious, simple or direct encoding, protocols are likely encodable through the use of auxiliary variables. However, our use of a protocol paradigm presents a significant conceptual distinction since we do not need sharing to be anchored to an ADT. Therefore, we can share individual references directly without requiring an intermediary module to indirectly offer access to the shared state, but we also allow such uses to exist. Similarly, although both models allow ownership recovery, our protocols are typing artifacts which means that we do not need an ADT layer to enable this recovery and the state of that protocol can be switched to participate in completely unrelated protocols, later on. Their abstractions are also shared symmetrically, while our protocols can restrict the available operations of each alias asymmetrically. Additionally, after the initial split, our shared state may continue to be split in new ways. Finally, we use focus to statically forbids re-entrant uses of shared state, while they use dynamic checks that diverge the execution when such operation is wrongly attempted.

8 Conclusions

We introduced a new flexible and lightweight interference control mechanism, *rely-guarantee protocols*. By constraining the actions of an alias and expressing the effects of the remaining aliases, our protocols ensure that only benign interference can occur when using shared state. We showed how these protocols capture many challenging and complex aliasing idioms, while still fitting within a relatively simple protocol abstraction. Our model departs from prior work by, instead of splitting shared resources encoded as monoids, offering an alternative paradigm of "temporal" splits that model the coordinated interactions between aliases. A prototype implementation, which uses a few additional annotations to ensure typechecking is decidable, is currently underway[7].

Acknowledgments. This work was partially supported 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, CITI PEst-OE / EEI / UI0527 / 2011, the U.S. National Science Foundation under grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages," and the U.S. Air Force Research Laboratory. We thank the Plaid (at CMU) and the PLASTIC (at UNL) research groups, and the anonymous reviewers for their helpful comments.

References

1. A. Ahmed, M. Fluet, and G. Morrisett. L3: A linear language with locations. *Fundam. Inf.*, 2007.

[7] Available at: https://code.google.com/p/deaf-parrot/

2. N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA*, 2008.
3. N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP*, 2011.
4. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.
5. L. Caires and J. a. C. Seco. The type discipline of behavioral separation. In *POPL*, 2013.
6. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. Logic in Computer Science*, 2007.
7. R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP*, 2004.
8. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, 2013.
9. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.
10. M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.
11. M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, 2002.
12. M. Fähndrich and K. R. M. Leino. Heap monotonic typestate. In *IWACO*, 2003.
13. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, 2010.
14. J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 1987.
15. C. S. Gordon, M. D. Ernst, and D. Grossman. Rely-guarantee references for refinement types over aliased mutable data. In *PLDI*, 2013.
16. J. B. Jensen and L. Birkedal. Fictional separation logic. In *ESOP*, 2012.
17. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 1983.
18. N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *ICFP*, 2012.
19. K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.
20. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *ICFP*, 2003.
21. F. Militão, J. Aldrich, and L. Caires. Rely-guarantee protocols (technical report). CMU-CS-14-107, 2014.
22. F. Militão, J. Aldrich, and L. Caires. Substructural typestates. In *PLPV*, 2014.
23. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, 2005.
24. A. Pilkiewicz and F. Pottier. The essence of monotonic state. In *TLDI*, 2011.
25. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. Logic in Computer Science*, 2002.
26. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proc. LISP and Functional Programming*, 1992.
27. F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, 2000.
28. R. E. Strom. Mechanisms for compile-time enforcement of security. In *POPL*, 1983.
29. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 1986.
30. K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, 2013.
31. J. A. Tov and R. Pucella. Practical affine types. In *POPL*, 2011.
32. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.
33. G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manipulating programs. *Electron. Notes Theor. Comput. Sci.*, 2005.