Fractional Permission Session Types

SELENE WU, New York University, USA
ANKUSH DAS, Boston University, USA

Session types provide a formal system for modeling and verifying message-passing concurrent systems. When constructed based on linear logic, they provide powerful guarantees of deadlock freedom and lack of data races, along with type preservation preventing communication errors. Unfortunately, these strong guarantees also require a major restriction that channels be treated strictly linearly, which precludes any form of sharing and significantly restricts the expressiveness of the resulting language. This work aims to relax this restriction by exploring a notion of immutability in message-passing. Intuitively, we introduce type-level constructors that demarcate an immutable operation, allowing for temporary state changes but requiring that state is restored at the end of the operation. To accommodate this notion of immutability, we introduce FracST, a novel type system based on a system of borrowing and fractional permissions that tracks whether a process holds a mutable or immutable reference to a channel. Our type system resembles Rust's 'Aliasing XOR Mutability' property: a channel can either have a single mutable reference or several immutable references. This enables safe sharing of channels between multiple clients without introducing the possibility of deadlocks, data races, or communication errors, proven via standard type safety theorems of progress and preservation. We implement FracST to demonstrate the efficacy of the language on real world examples, including shared data structures like queues, binary trees, and AVL trees, and protocols like an auction and MapReduce.

1 Introduction

Session types [12, 24–26] model communication protocols as types, specifying the sequence and structure of messages exchanged over channels and statically enforcing protocol adherence. In particular, session types rooted in linear logic provide strong guarantees: *type preservation* (which implies that processes follow the prescribed protocol), *deadlock freedom*, and *absence of data races* in well-typed programs. These guarantees arise from the Curry-Howard correspondence [11, 12, 50] between session types and intuitionistic linear logic [18, 19]: logical connectives are interpreted as communication primitives in a message-passing system.

Despite these advantages, traditional session type systems impose a fundamental limitation: *channels must be used linearly*. This means each channel endpoint is used by exactly one process, ensuring exactly one *client* and one *provider* per channel. This restriction is essential to linear logic-based session typing: it prevents *(i)* aliasing, guaranteeing preservation, and *(ii)* cyclic communication topologies, guaranteeing deadlock freedom. Yet the same restriction limits expressivity. In practical concurrent systems, it is often desirable for multiple clients to share access to a service—for example, a shared queue or database accessed by many clients simultaneously. Strict linearity precludes such sharing, significantly restricting the class of protocols expressible via traditional session types.

Owing to this limitation, prior work [2, 3, 44, 45, 47] has aimed at relaxing linearity by introducing shared mutable state with *locking*. SILL_S [2] and its variants [3, 47] rely on an *acquire-release* discipline: shared channels are first acquired, used for communication, and finally released. πSS [44] provides a Propositions-as-Types interpretation of shared state via reference cells that can be shared, read, or written, with locking supported in its extension πSSL. While these systems restore expressivity, they inherit the potential for deadlock associated with lock-based concurrency. This is mitigated either via a sophisticated priority-based type system [3], or taking reference cells outside processes and permitting only a tree-like topology [45].

Authors' Contact Information: Selene Wu, selenewu@nyu.edu, New York University, New York, New York, USA; Ankush Das, ankushd@bu.edu, Boston University, Boston, Massachusetts, USA.

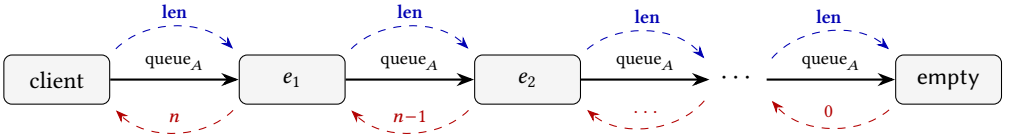


Fig. 1. A **len** operation on an n-element queue propagates down the chain of elem processes. Blue arrows show the request propagating right; red arrows show the aggregated length returning left.

In this work, we take a fundamentally different approach to sharing by distinguishing between *mutable* and *immutable* interactions with a process, without resorting to locks. Rather than uniformly requiring exclusive access for all channel interactions, we observe that only *state-modifying* operations need exclusivity, while *state-observing* operations can safely be performed by multiple clients concurrently. For example, in a queue, insertion and deletion modify the queue's contents and require exclusive access, while getting the length only reads information without permanently altering state. The key insight is that safe concurrent sharing does not require locking as long as we can identify which operations are genuinely immutable.

However, defining immutability in a session-typed setting is not straightforward. As a running example, consider the session type of a queue holding elements of type A:

$$\begin{aligned}
\text{queue}_A = \&\{&\textbf{ins} : A \multimap \text{queue}_A && \text{// receive element } a : A\text{, insert and recurse}\\
&\textbf{del} : \oplus\{\textbf{some} : A \otimes \text{queue}_A,\ \textbf{none} : \mathbf{1}\} && \text{// return front element, or signal empty}\\
&\textbf{len} : \text{nat} \otimes \text{queue}_A\} && \text{// return length and recurse}
\end{aligned}$$

The queue offers an external choice (&) of three operations. On **ins**, it receives an element of type A (via $\multimap$) and recurses. On **del**, it makes an internal choice ($\oplus$): either returning the **some** label followed by an element (via $\otimes$) and recursing, or signalling empty with **none** label and terminating (via $\mathbf{1}$). On **len**, it sends back a natural number indicating the queue length and recurses. Intuitively, **ins** and **del** are mutable—they change the queue's contents—while **len** is immutable: the queue before and after a **len** operation is the same.

Formalizing this intuition, however, turns out to be quite challenging for two reasons. First, even a read-only operation like **len** *temporarily* changes the state of the queue: after receiving **len**, the type transitions from queue_A to $\text{nat} \otimes \text{queue}_A$ while the length is being sent, before returning to queue_A. A strict definition of immutability that requires that the process state *never changes* would therefore rule out even this benign operation. Instead, we need a more flexible notion: an operation may temporarily alter state but must *restore* it before the interaction concludes. Second, immutability must *cascade* compositionally. A queue is implemented as a chain of processes, each holding one element (Figure 1). When the queue receives a **len** operation, it propagates the request down the entire chain and collects the result on the way back. Not only must the head process restore its state, but every element process in the chain must also restore its state. More generally, when a process concludes its interaction, the entire subtree of processes rooted at it must restore their state.

To address these challenges, we introduce Fractional Permission Session Types (FracST), a novel typing discipline integrating session types with *permission-based reasoning* [8, 9]. The first key idea is to annotate session types with two new modal operators, $\uparrow^I_M$ and $\downarrow^I_M$, that demarcate the boundaries of an immutable session. $\uparrow^I_M$ marks the *start* of a potential immutable interaction, recording the provider's current state. $\downarrow^I_M$ marks its *end*; it is only reachable if the provider's state matches its state at $\uparrow^I_M$, enforced statically by the type system with no runtime overhead. Annotating the queue type with these operators gives:

$$\begin{aligned}\text{queue}_A = \uparrow^I_M \,\&\{&\textbf{ins} : A \multimap \text{queue}_A && \text{// record state here ...}\\ &\textbf{del} : \oplus\{\textbf{some} : A \otimes \text{queue}_A,\ \textbf{none} : \mathbf{1}\}\\ &\textbf{len} : \text{nat} \otimes \downarrow^I_M \text{queue}_A\} && \text{// ... state must be restored here}\end{aligned}$$

The $\uparrow^I_M$ precedes the & denoting that an immutable operation (namely **len**) may potentially start, and $\downarrow^I_M$ follows nat$\otimes$ in the **len** branch. Intuitively, this captures the requirement that for *each process* in the chain (Figure 1), the process state *before* **len** is received must be identical to the process state *after* the nat result has been sent back.

This naive type, however, reveals a problem. Since $\uparrow^I_M$ appears before the &, the provider enters an immutable session before knowing which branch is requested. But **ins** and **del** are mutable operations—they should not be permitted in an immutable session. We may consider placing $\uparrow^I_M$ *after* receiving the **len** label. But alas, that is too late! The queue's type has already changed from queue_A to $\text{nat} \otimes \text{queue}_A$, meaning its state has already been modified.

FracST resolves this by introducing a third operator $\Downarrow^I_M$, an *escape hatch* that aborts an immutable session and transitions back to a regular mutable session. Adding $\Downarrow^I_M$ to the mutable branches yields the correct queue type:

$$\begin{aligned}\text{queue}_A = \uparrow^I_M \,\&\{&\textbf{ins} : \Downarrow^I_M A \multimap \text{queue}_A && \text{// mutable: abort, modify state}\\ &\textbf{del} : \Downarrow^I_M \ \oplus\{\textbf{some} : A \otimes \text{queue}_A,\ \textbf{none} : \mathbf{1}\} && \text{// mutable: abort, modify state}\\ &\textbf{len} : \text{nat} \otimes \downarrow^I_M \text{queue}_A\} && \text{// immutable: restore state}\end{aligned}$$

In this corrected type, every branch has a matching down arrow: **ins** and **del** exit via $\Downarrow^I_M$ (aborting the immutable session to perform mutable work), while **len** exits via $\downarrow^I_M$ (restoring the recorded state). The up arrow $\uparrow^I_M$ thus always pairs with a down-arrow in every branch, statically guaranteeing that no immutable session is left open.

FracST's final ingredient is *permissions*, which determine whether a *client* may invoke the $\Downarrow^I_M$ escape hatch. Each channel is associated with a permission that is granted to its client: full ownership (permission $*$) grants access to $\Downarrow^I_M$ branches, while a fractional permission restricts the client to immutable interactions only. In addition, a provider either permits a *single* client with full ownership, or *multiple* clients with fractional ownership. This resembles the *Aliasing XOR Mutability* principle in a message-passing setting: either one mutable reference or multiple immutable references to a channel, but never both simultaneously. To ensure the cascade property holds compositionally, each process is assigned a *mode* (mutable or immutable) and the type system ensures that when a process enters immutable mode, all processes it delegates to do the same.

Fractional permissions enable safe channel sharing without compromising type preservation or deadlock freedom. Because sharing is achieved without explicit locks or acquire-release patterns, proving deadlock freedom (progress) is relatively straightforward. Proving type preservation is significantly more challenging, and crucially relies on process constructs that explicitly demarcate transitions *into* and *out of* immutable mode. Entry into immutable mode records the current protocol state; exit is only permitted when the current state matches the recorded one. This static discipline ensures that any temporary state changes within an immutable session are fully restored, preventing interference among concurrently sharing clients.

FracST yields a practical programming paradigm for concurrent services. When a process spawns a channel, it holds exclusive ownership. It can *split* that ownership into fractional permissions and distribute them to multiple clients, enabling concurrent immutable access. Those fractions can later be *merged* to recover full ownership and resume mutable operations. More complex patterns are also supported: ownership can be passed between clients through structured protocols that enforce proper resource management. Through these mechanisms, FracST enables flexible sharing while preserving type preservation and deadlock freedom.

We have implemented FracST and evaluated it on several case studies: copyable types, shared data structures like stacks and queues, and binary and AVL trees, and protocols like auction and MapReduce. The data structures demonstrate that mutable operations (insertion, deletion) and immutable operations (length) can coexist within a single protocol. The auction features alternating phases of serial mutable access and concurrent immutable access shared among an arbitrary number of bidders. To minimize programmer burden, FracST equips a *reconstruction* algorithm that automatically inserts most of the process constructs related to the up/down modalities.

In summary, this paper makes the following contributions:

- A novel session type system integrating fractional permission-based reasoning, with modal operators that statically enforce immutability and state restoration (Section 3)
- An operational semantics that implements the modal discipline, including mode transitions and the cascading restoration invariant (Section 4)
- Metatheoretical results establishing type preservation and deadlock freedom in the presence of sharing for well-typed programs (Section 5)
- A prototype implementation evaluated on practical case studies that automatically reconstructs process constructs associated with type modalities (Section 6)

2 Overview

Session types correspond to intuitionistic linear logic [18, 19] via the Curry-Howard isomorphism [11, 12], giving each logical connective a message-passing interpretation. Each channel has a unique provider and client performing dual actions: every send by one is a receive by the other, preventing communication mismatches. Internal choice $\oplus\{\ell : A_\ell\}_{\ell \in L}$ (generalizing $A \oplus B$) is interpreted as the provider sending a label $\ell \in L$ and continuing at type A_ℓ. Dually, external choice $\&\{\ell : A_\ell\}_{\ell \in L}$ (generalizing $A \,\&\, B$) reverses the roles of provider and client, i.e., the client sends a label that the provider receives. Session *delegation*, i.e., communicating channels over channels is realized via the $\otimes$ and $\multimap$ constructors. Simultaneous conjunction $A \otimes B$ is the provider sending a channel of type A to the client and continuing at B; dually, $A \multimap B$ has the client sending the channel. Finally, $\mathbf{1}$ is interpreted as the provider closing the channel and terminating.

As our running example, recall the session type queue_A of a queue holding elements of type A, introduced in Section 1, which offers the three operations **ins**, **del**, and **len**. As described in Figure 1, a queue is implemented as a chain of elem processes terminated by an empty process (representing an empty queue). Each elem process holds one element of the queue and is defined as follows:

```
proc elem (a: A) (t: queue) |- (q: queue) =
  case q {
  | ins => a' <- recv q;      % receive a' : A on channel q
           t.ins;             % send ins label on channel t
           send t a';         % send a' : A on channel t
           q <- elem a t      % recurse back to elem process
  | del => ...                % elided for brevity
  | len => t.len;             % send len label on channel t
           n <- recv t;       % receive n : nat on channel t
           n1 <- succ n;      % increment n by 1 by calling 'succ' process
           send q n1;         % send n1 : nat on channel q
           q <- elem a t }    % recurse back to elem process
```

Each elem process holds an element $a : A$, offers a channel $q : \text{queue}_A$ to its successor process, and uses a channel $t : \text{queue}_A$ connecting to its predecessor. The first line of the code above declares the

process: the used channels and their types appear to the left of the turnstile, the provided channel and type to the right.

The implementation case-analyzes on the provided channel q and branches on the received label. In the **ins** branch, it receives $a' : A$ on q and forwards it down the chain: t.**ins** sends the **ins** label on t, followed by a', before recursing. In the **len** branch, it forwards the **len** label on t and receives $n : \text{nat}$, the length of the tail. It computes $n_1 = n + 1$ via the succ process, sends n_1 back on q, and recurses. We omit the **del** branch for brevity.

Due to the linearity restriction of session types, this queue must have a *single* client. This is sensible for **ins** and **del**: we do not want multiple clients concurrently modifying the queue, which would cause race conditions and could violate its first-in-first-out property.

For **len**, however, the restriction is overly conservative, since a length operation does not *mutate* the queue. The state of *any* process in the chain before the **len** operation and after the length has been sent back are identical. Ideally, we would allow multiple clients to issue **len** queries simultaneously, which requires *sharing* the channel the queue provides.

Immutability in Types and Processes. Before sharing channels, we need a precise notion of what it means for a client to operate on a channel *immutably*. The most obvious definition would require the provider's state to remain unchanged throughout an immutable interaction. But this is far too restrictive as even receiving a message causes a process to mutate state. To relax this notion, we allow an immutable operation to be a *sequence* of interactions that may *temporarily* change the provider's state, as long as the original state is restored before the operation concludes. As introduced in Section 1, FracST enforces this with three modal type operators: $\uparrow^I_M$ marks the start of an immutable session, the state that will need to be restored; $\downarrow^I_M$ marks its end, reachable only when the current state matches the starting state; and $\Downarrow^I_M$ is an *escape hatch* that aborts the session and returns to a regular mutable session. Revisiting the queue type:

$$\begin{aligned}\text{queue}_A = \uparrow^I_M \,\&\{&\textbf{ins} : \Downarrow^I_M A \multimap \text{queue}_A,\\ &\textbf{del} : \Downarrow^I_M \oplus \{\textbf{some} : A \otimes \text{queue}_A, \textbf{none} : \mathbf{1}\}\\ &\textbf{len} : \text{nat} \otimes \downarrow^I_M \text{queue}_A\}\end{aligned}$$

Since $\uparrow^I_M$ precedes the external choice, the immutable session begins before the requested branch is known, so each branch must reach a matching down arrow: **ins** and **del** abort via $\Downarrow^I_M$ to perform their mutations, while **len** restores the recorded state and concludes via $\downarrow^I_M$.

Each type operator has a corresponding term construct in the implementation. The provider wraps its body in an `immut t { }` block (corresponding to $\uparrow^I_M$), which starts an immutable session on t. The matching `continue t` (corresponding to $\downarrow^I_M$) restores t's state and ends it. Mutable branches escape via a `mut { }` block (dual to $\Downarrow^I_M$):

```
proc elem (a: A) (t: queue) |- (q: queue) =
immut t {
  case q {
  | ins => mut { ... }
  | del => mut { ... }
  | len => ...; continue t } }
```

The code mirrors the type: `immut t` opens the session before the case analysis, the **ins** and **del** branches abort it with `mut { }`, and the **len** branch closes it with `continue t`.

Permissions. To exploit immutability, we need to share a channel among multiple clients, which we achieve with a system of *fractional permissions* [8, 9]. Each channel carries a permission drawn from $(0, 1] \cup \{*\}$ together with a *unique identifier*, so its type becomes a triple $\langle A, p, a\rangle$ of session

type A, permission p, and identifier a. Identifiers track which provider a channel comes from, to ensure soundness when we *split* and *merge* permissions. Accordingly, the delegation types take the form $\langle A, p, a\rangle \otimes B$ and $\langle A, p, a\rangle \multimap B$: the transmitted channel is given its full triple, while the continuation B remains a plain session type. This asymmetry in the $\otimes$ and $\multimap$ operators is by design: since the permission and identifier of a channel are already specified in its full type in a typing context, they do not need to be specified in a continuation type.

The permission $*$, read as *owned*, is what grants a client the right to perform mutable operations, i.e., to abort an immutable session via the $\Downarrow^I_M A$ escape hatch. This is distinct from permission 1: as we discuss later, a client may hold the only reference to a provider yet still be unable to mutate it.

Adding permissions and identifiers to channels also requires *parametricity* over them, so that processes can exchange a permission or identifier alongside a channel. We provide this through the quantified types $\exists v.\ A$ and $\forall v.\ A$, which bind the sent permission or identifier to the variable v. We further annotate the $\uparrow^I_M$ operator with a parameter k, by which the client communicates the provider's current permission when starting an immutable operation (Section 3.3). With these additions, the queue type is complete:

$$\begin{aligned}\text{queue}_A = \uparrow^I_M k.\ \&\{&\textbf{ins} : \Downarrow^I_M \forall a_i.\ \langle A, *, a_i\rangle \multimap \text{queue}_A,\\ &\textbf{del} : \Downarrow^I_M \oplus \{\textbf{some} : \exists a_d.\ \langle A, *, a_d\rangle \otimes \text{queue}_A, \textbf{none} : \mathbf{1}\}\\ &\textbf{len} : \exists n.\ \langle \text{nat}, *, n\rangle \otimes \downarrow^I_M \text{queue}_A\}\end{aligned}$$

The difference from the previous version lies to the left of the $\otimes$ and $\multimap$ operators, where each exchanged channel now carries a full triple. In the **ins** branch, $\forall a_i.\ \langle A, *, a_i\rangle$ requires the client to send an identifier a_i and then a channel of type A with permission $*$ and identifier a_i. Dually, in the **del** and **len** branches, $\exists a_d.\ \langle A, *, a_d\rangle$ and $\exists n.\ \langle \text{nat}, *, n\rangle$ reverse the direction of communication. They require the queue to send an identifier a_d (resp. n) and a channel of type A (resp. nat) with that identifier to the client. In every case the permission is $*$, so the exchanged channel is transferred with full ownership.

A subtle consequence of this type is that the queue stores *mutable* elements! Because the channels exchanged in the **ins** and **del** branches carry permission $*$, a client must own mutable access to an element when inserting it, and the queue likewise hands back mutable access on deletion. The same holds for the length returned by **len**.

Clients manipulate permissions through four constructs. The first, '$x_1\ x_2 \leftarrow$ split x', takes an *immutable* reference x and splits it into two immutable references x_1 and x_2; if x has permission p, each of x_1 and x_2 receives $p/2$, and both retain x's identifier. Dually, '$x \leftarrow$ merge $x_1\ x_2$' recombines references with permissions p_1 and p_2 into a single x of permission $p_1 + p_2$. Merging succeeds only when x_1 and x_2 share the *same identifier*, not merely the same session type, which is the primary reason for introducing unique identifiers. Merge is essential: once a mutable channel is shared and its permission split, merge is the only way to recombine the permissions so the channel can become mutable again. Both operations preserve the total permission.

Split and merge apply only to channels with a numeric permission, but a mutable channel has permission $*$, which cannot be split. The remaining two constructs bridge this gap: 'share x' transitions x's permission from $*$ to 1, and 'own x' transitions it back from 1 to $*$. Together with split and merge, they let a mutable channel be shared immutably and later reclaim full ownership to become mutable again.

Permission manipulation constructs may only be applied to channels with an immutable $\uparrow^I_M$ session type, ensuring that any operation performed while a channel is split belongs to an immutable session. Moreover, once $\uparrow^I_M$ has been reached, the only way to leave immutable mode is through a $\Downarrow^I_M$ type, which demands mutable access (permission $*$) and therefore that the channel is not

split. This guarantees that a channel carries no temporary state when its fractions are merged and enforces a clean separation between mutable and immutable operations.

Finally, we describe the complete and updated definition of elem. The process is parameterized by the identifiers `ai` and `ti` of a and t, and holds mutable access (permission $*$) to both channels. Remarkably, the provided channel q does not need identifier or permission annotations! FracST is designed such that a provider of a channel does not need to be aware of whether the channel is being used mutably or immutably. It can always presume that it holds the channel exclusively with the permission discipline enforced entirely on the client side.

```
proc elem[ai,ti] : (a : <A,*,ai>), (t : <queue,*,ti>) |- (q : queue) =
  share t;                          % share t so that immut is possible
  immut t { p =>                    % start an immut block; receive permission p
    case q (                        % t's permission is 1, so allowed
    | ins => mut {                  % abort immut session; start mut block
        own t;                      % own channel back; t's permission is now *
        {ai'} <- recv q;            % receive identifier of a'
        a' <- recv q;               % receive a' : A on q
        start t{*};                 % start an immutable session on t
        t.ins;                      % send ins label on t
        mutate t;                   % abort immutable session on t;
        send t {ai'};               % send identifier of a'
        send t a';                  % send a' on t
        q <- elem[ai,ti] a t }      % recurse
    | del => mut { ... }
    | len => start t{p}; t.len; {ni} <- recv t; n <- recv t; finish t;
             {n1i}, n1 <- succ[ni] n; send q {n1i}; send q n1; continue t  ) }
```

The process begins with `share t`, which lowers t's permission from $*$ to 1. This is required because an `immut` block may only use channels from an external scope if they have a numeric permission. The `immut t { p => }` block then opens an immutable session that operates on t, binding p to the permission supplied by q's client (the argument k of $\uparrow^I_M$). Inside, the process case-analyzes on the provided channel q. The **ins** branch is mutable, so it immediately escapes the immutable session with a `mut { }` block and reclaims full ownership of t via `own t`, restoring its permission to $*$. It then receives the identifier `ai′` and the channel `a′` of the element to be inserted. Because every queue interaction must begin in immutable mode, elem cannot send **ins** to its tail `t` directly: it first opens an immutable session on t with `start t{*}` (the client-side dual of `immut`), sends the **ins** label, and then escapes back to a mutable session with `mutate t` (the client-side dual of `mut`). With the tail now in mutable mode, it forwards a''s identifier and channel with `send t`, and finally recurses. The **del** branch follows the same discipline and is elided.

The **len** branch is much simpler: we start an immutable session on `t` with permission p, and the remaining communication is exactly like before. We send the **len** message on `t`, followed by receiving the nat identifier and channel. We then terminate the immutable session with a `finish` t, compute the successor, and then continue on `t`.

Identifiers in the Type. Because identifiers are first-class in FracST, i.e., they can be sent and received through $\exists v.\ A$ and $\forall v.\ A$ and bound locally to a variable, it may seem that a process could forge an identifier or attach the same identifier to two different providers, introducing aliasing. This is not possible precisely because *both* permission and identifier are part of the channel's *type*.

A process can only use a channel at the identifier recorded in its type, and the only ways to obtain a channel of type $\langle A, p, a\rangle$ are to receive one, or to derive one by split or merge, which preserve the identifier. Consider the **ins** branch of the queue type, $\forall a_i.\ \langle A, *, a_i\rangle \multimap \text{queue}_A$. The client chooses the identifier it sends, but the channel that follows must have type $\langle A, *, a_i\rangle$ for that very a_i. A client holding $x : \langle A, *, a\rangle$ and $y : \langle A, *, b\rangle$ therefore cannot send a followed by y, as the type of y does not match, and it cannot relabel y as a either. The same holds for merge. Suppose a process holds $x : \langle A, 1, a\rangle$ and $y : \langle A, 1, b\rangle$, two channels of the same session type A (an immutable $\uparrow^I_M$ type, as split and merge require) but with different identifiers $a \neq b$. It splits both:

$$\begin{aligned} x_1, x_2 &\leftarrow \mathsf{split}\ x & x_1 &: \langle A, 1/2, a\rangle\,,\ x_2 : \langle A, 1/2, a\rangle \\ y_1, y_2 &\leftarrow \mathsf{split}\ y & y_1 &: \langle A, 1/2, b\rangle\,,\ y_2 : \langle A, 1/2, b\rangle \end{aligned}$$

Each split retains the identifier of the channel being split, so x_1 and x_2 carry a while y_1 and y_2 carry b. And merging across channels is not permitted: '$z \leftarrow \mathsf{merge}\ x_i\ y_j$' is ill-typed for any $i, j \in \{1, 2\}$, because x_i and y_j have the same session type and the same permission $1/2$ but different identifiers. Without identifiers in the types, the two would look identical and the merge would be accepted, fabricating a channel with permission 1 out of two halves of unrelated providers. Merging is only permitted when the channel identifiers can be proven to be equal *statically*. Thus, identifiers need no runtime uniqueness check: the type system guarantees that two channels carry the same identifier only if they are fractions of the same provider.

Is it possible for multiple clients to mutate the queue? Yes, because permissions can be exchanged freely. A client holding mutable access can modify the queue and then pass ownership to another client, which mutates it and passes ownership on in turn, potentially forming complex patterns of exclusive mutators that emulates a lock. This pattern can even be enforced in the type system with a type of the form $\forall a.\ \langle A, *, a\rangle \multimap \langle A, *, a\rangle \otimes \ldots$ where reusing the identifier a guarantees that clients hand back the very same channel. A client in the schedule may also distribute immutable access to several readers, collect the fractions back to regain mutable access, and repeat. In general, FracST supports such complex access patterns while preserving deadlock freedom, which is impossible to express in a lock-based system like SILL_S [2]

To serve multiple concurrent requests, each possibly carrying temporary state, shared processes follow a *replicated server* semantics [25, 42, 49]: the provider spawns a fresh copy of itself for each incoming operation, which closes once that operation completes. This is safe precisely because shared operations are immutable and leave the provider's state unchanged. An owned process, by contrast, has a single client at a time and so serves operations directly, which both supports mutation and avoids needless copying.

Program Reconstruction. Dealing with permissions in FracST introduces a significant syntactic overhead, in particular *(i)* starting and finishing immutable sessions, *(ii)* owning and mutating channels, and *(iii)* exchanging identifiers right before exchanging the corresponding channel. To minimize this burden, FracST comes equipped with a novel *reconstruction* algorithm that automatically inserts all these constructs. This results in much more compact and user-intuitive code that improves the overall readability of FracST.

We demonstrate this with a final example: a complete client program of the queue data structure that exercises these constructs to perform both mutable and immutable operations.

$$
\begin{aligned}
\text{Session Types} \quad A, B &::= \oplus\{\ell : A_\ell\}_{\ell \in L} \mid \&\{\ell : A_\ell\}_{\ell \in L} \mid T \otimes B \mid T \multimap B \mid \mathbf{1} \\
&\mid \uparrow^I_M k.\, A \mid \downarrow^I_M A \mid \Downarrow^I_M A \mid \exists v.\, A \mid \forall v.\, A \\
\text{Channel Types} \quad T &::= \langle A, p, a \rangle \\
\text{Permissions} \quad p &::= * \mid n \mid v \mid p \cdot p \mid p + p \\
\text{Permission/Identifier Expressions} \quad e &::= p \mid a
\end{aligned}
$$

Fig. 2. Type Grammar for FracST

Implicit Syntax:

```
1  q <- empty;
2  e1 <- newA;
3  q.ins;
4  send q e1;
5  q1, q2 <- split q;
6
7  q1.len; q2.len;
8  n1 <- recv q1;
9  n2 <- recv q2;
10
11 q <- merge q1, q2;
12 e2 <- newA;
13 q.ins;
14 send q e2;
```

Explicit Syntax:

```
1  {qi}, q <- empty;
2  {e1i}, e1 <- newA;
3  start q{*}; q.ins; mutate q;
4  send q {e1i}; send q e1;
5  share q; q1, q2 <- split q;
6  start q1{1/2}; start q2{1/2};
7  q1.len; q2.len;
8  {n1i} <- recv q1; n1 <- recv q1;
9  {n2i} <- recv q2; n2 <- recv q2;
10 finish q1; finish q2;
11 q <- merge q1, q2; own q;
12 {e2i}, e2 <- newA;
13 start q{*}; q.ins; mutate q;
14 send q {e2i}; send q e2;
```

It is easier to first read the implicit syntax (on the left) that the programmer is expected to write. The client first creates an empty queue that produces a mutable channel `q` and inserts a fresh element `e1` (lines 1–4). Then, it splits the channel `q` into two immutable channels `q1` and `q2` (line 5) and issues two concurrent **len** queries on them (line 7), followed by receiving the lengths (lines 8–9). Finally, the client reclaims ownership by merging `q1` and `q2` back into `q` (line 11) and inserts a second element `e2` (lines 12–14).

The explicit syntax (on the right) is significantly more verbose. First, there are no mentions of identifiers in the implicit syntax, whether in communication (lines 3, 8–9, 14) or spawning processes (lines 1, 2, 12). Second, the '`start`', '`mutate`', '`share`', '`finish`', and '`own`' constructs are also usually not needed in implicit syntax. All the aforementioned constructs can automatically be deduced from channel types in *linear-time*. Only the `split` and `merge` operations remain explicit, since they express the programmer's intent to share permissions and later reclaim it.

3 Formal Syntax and Type System

Formally, FracST is a conservative extension of a traditional binary session typing system [42]. Figure 2 describes the grammar of types in FracST. The choice types $\oplus\{\ell : A_\ell\}_{\ell \in L}$ and $\&\{\ell : A_\ell\}_{\ell \in L}$ are standard. The former represents *internal* choice requiring the provider to send a label $k \in L$ to the client who must branch on the label received. Dually, the latter represents *external* choice, which reverses the roles and has the client send a label to the provider. Session delegation (types $T \otimes B$ and $T \multimap B$) is also standard except the channel being exchanged is represented using the triple T of the form $\langle A, p, a \rangle$ where A denotes the channel type, p denotes its permission, and a denotes its unique identifier.

$$
\begin{array}{lll}
\text{Processes} & P, Q ::= x.\mathsf{k}\,;\, P \mid \mathsf{case}\ x\ (\ell \Rightarrow P_\ell)_{\ell \in L} \mid \mathsf{send}\ x\ y\,;\, P \mid y \leftarrow \mathsf{recv}\ x\,;\, P \\
& \quad \mid \mathsf{close}\ x \mid \mathsf{wait}\ x\,;\, P \mid x \leftrightarrow y \mid \{a\}\ x \leftarrow f[\overline{e}]\ \overline{y}\,;\, P \\
& \quad \mid \mathsf{mut}\ \{P\} \mid \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\} \mid \mathsf{continue}\ \overline{y} \\
& \quad \mid \mathsf{start}\ x\{p\}\,;\, P \mid \mathsf{finish}\ x\,;\, P \mid \mathsf{mutate}\ x\,;\, P \\
& \quad \mid x_1\ x_2 \leftarrow \mathsf{split}\ x\,;\, P \mid x \leftarrow \mathsf{merge}\ x_1\ x_2\,;\, P \mid \mathsf{share}\ x\,;\, P \mid \mathsf{own}\ x\,;\, P \\
& \quad \mid \mathsf{send}\ x\ \{e\}\,;\, P \mid \{v\} \leftarrow \mathsf{recv}\ x\,;\, P \\
\text{Declaration} & D ::= \mathsf{proc}\ f[\overline{v}] : \overline{y : T} \vdash (x : A) = P
\end{array}
$$

Fig. 3. Process Grammar for FracST

The three modal operators $\uparrow^I_M k.\ A$, $\downarrow^I_M A$, and $\Downarrow^I_M A$ are unique to FracST and demarcate immutable sessions: $\uparrow^I_M k.\ A$ marks the *start* of an immutable session, binding the provider's permission to k; $\downarrow^I_M A$ marks its *end*. And $\Downarrow^I_M A$ is the *escape hatch* that aborts the immutable session to perform mutable work. The quantifiers $\exists v.\ A$ and $\forall v.\ A$ provide *parametricity over permissions and identifiers*, letting a provider send (former) or receive (latter) a permission or identifier, bound to variable v, before continuing at A, and requiring the client to perform the dual action.

Permissions are denoted by p, which can be $*$ (full mutable permission) or a rational number in the interval $(0, 1]$. To support parametricity, we allow permission variables x and constants n along with addition and multiplication operators. Lastly, we use e to denote either a permission expression or an identifier.

Type validity is expressed using the judgement $\mathcal{V} \vdash A$ valid, where $\mathcal{V}$ is a set of permission and identifier variables and A is a session type. This judgment simply checks that all freely occuring permission and identifier variables in A are a subset of $\mathcal{V}$. The full rules for this judgement are provided in the appendix.

FracST process expressions P are given by the grammar in Figure 3. The first group of constructs is standard: 'x.k' sends a label k on channel x, while case $x\ (\ell \Rightarrow P_\ell)_{\ell \in L}$ receives a label on x and branches accordingly; '$\mathsf{send}\ x\ y$' and '$y \leftarrow \mathsf{recv}\ x$' send and receive a channel y over x; '$\mathsf{close}\ x$' terminates channel x and the process, while '$\mathsf{wait}\ x$' waits for x to terminate. Finally, '$x \leftrightarrow y$' corresponds to *identity* and forwards one channel to another, and '$\{a\}\ x \leftarrow f[\overline{e}]\ \overline{y}$' spawns a new process running f that offers channel x with identifier a. We use the notation $\overline{e}$ to denote a sequence of expressions of the form e (resp. $\overline{y}$ denotes a sequence of channels y).

The remaining constructs are specific to FracST. A provider opens an immutable session that will use channels $\overline{y}$ with '$\mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}$', binding the channel's permission to p, to be freely used in P. This session can either end with '$\mathsf{continue}\ \overline{y}$', or abort with '$\mathsf{mut}\ \{P\}$'. Dually, a client drives the corresponding transitions on a used channel via '$\mathsf{start}\ x\{p\}$', '$\mathsf{finish}\ x$', and '$\mathsf{mutate}\ x$' respectively. To exchange permissions and identifiers, we allow '$\mathsf{send}\ x\ \{e\}$' and '$\{v\} \leftarrow \mathsf{recv}\ x$': the former sends expression e, while the latter receives the expression and binds it to v. To split and recombine permissions, we use '$x_1\ x_2 \leftarrow \mathsf{split}\ x$' that equally divides x's permission into x_1 and x_2 and '$x \leftarrow \mathsf{merge}\ x_1\ x_2$' and sums up x_1 and x_2's permissions into x. Finally, '$\mathsf{share}\ x$' transitions a channel from full $(*)$ to unit (1) permission so that it can be split up, and '$\mathsf{own}\ x$' does the reverse transition after permissions have been merged.

A FracST program is a collection of mutually recursive process *declarations*, gathered into a global signature Σ. A declaration $\mathsf{proc}\ f[\overline{v}] : \overline{y : T} \vdash (x : A) = P$ introduces a process named f that is parametric in the permission and identifier variables $\overline{v}$. The entries to the left of the turnstile, $\overline{y : T}$, are the channels f *uses*, each annotated with its full triple T, while the entry to its right, $x : A$, is the channel f *provides*. P is a process expression that represents the function body.

Typing Judgment. Following the Curry-Howard correspondence between linear logic and session types [11, 50], the typing judgements for FracST is modeled on intuitionistic linear logic. As a result, the logical sequent $A_1, A_2, \ldots, A_n \vdash A$ can be generalized to $(x_1 : A_1), (x_2 : A_2), \ldots, (x_n : A_n) \vdash P :: (x : A)$. This denotes a session A being provided on channel x by a process P which uses sessions $A_1, A_2, \ldots, A_n$ being provided on channels $x_1, x_2, \ldots, x_n$ respectively. In order to account for our augmented channel types with permissions and identifiers, we extend each $(x_i : A_i)$ to $(x_i : \langle A_i, p_i, a_i \rangle)$. As is standard, we gather these into a linear context Δ. To fully account for our tracking of permissions and immutability, we extend the typing judgement to

$$\mathcal{V} \,;\, \mathcal{O} \,;\, \Delta_L \,;\, \Delta \vdash_{\mathcal{N}} P :: (x : A).$$

The context $\mathcal{V}$ collects the free permission and identifier variables in scope. The context $\mathcal{O}$ records the identifiers of channels that P is permitted to own (move from permission 1 to permission $*$) and is treated linearly. This ownership tracking lets us forbid mutation of a channel that is being used in an immutable session. The context Δ_L holds the channels that are frozen for the duration of an immutable session: P may not communicate on them. This enables *cascading* of immutability by guaranteeing that an immutable session leaves their state untouched. Finally, the subscript $\mathcal{N}$ is the *restoration snapshot*. It is **None** when P is an owned, mutable process, and **Some**$(A_c, \Delta_c, \mathcal{O}_c, \overline{y}_c, p_c)$ when P is inside an immutable session, in which case it records the type A_c, context Δ_c, and owned channels $\mathcal{O}_c$ that P must restore via continue at the end of the session, together with the $\overline{y}_c$ and p_c of the enclosing immut block. The restoration snapshot is what lets the type system verify, statically, that the recorded state is restored before the session concludes. We also implicitly assume that $\mathcal{V} \vdash A$ valid and that $\mathcal{V} \vdash B$ valid for all $x : \langle B, p, b \rangle$ in Δ and Δ_L.

Note that the provided channel $x : A$ carries no permission or identifier annotation, unlike the channels in Δ. This is because the client, not the provider, controls a channel's permission: it may freely split, merge, share, or own it, so the provider need not track the exact permission of what it offers. Outside an immutable session, a provider may simply assume its provided channel is at permission $*$. On entering an immutable session, the client sends over its current permission (the k of $\uparrow^I_M k$. A and the p of '$\mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}$'). Since this permission cannot change for the duration of the session, the provider can still reason about it precisely.

The typing rules for FracST can be broken down into four categories: basic rules, permission/identifier parametricity rules, immutability rules, and sharing rules. We omit some dual rules in the descriptions below and present the full set of rules in the appendix.

3.1 Basic Rules

Figure 4 describes the basic rules of FracST that are inherited almost unchanged from standard intuitionistic linear session types [11], governing label and channel communication. The first two of these rules govern the sending and receiving of labels for the internal choice ($\oplus$) type constructor. In the $\oplus$R rule, the construct $x.k \,;\, P$ sends label k on the provided channel x and then continues with P, updating the type of x to the type in k-th branch, i.e., A_k. In the $\oplus$L rule, the case construct receives a label $\ell \in L$ on a used channel x and continues with corresponding branch Q_ℓ and x's type A_ℓ. These rules do not otherwise care about any of our additional contexts, as is the case for most of the other basic rules.

The only exceptions to this are the rules for terminating and spawning, which interact with the additional contexts in minor ways. The **1**R and id rules both require the restoration snapshot to be **None**, reflecting that termination is a mutable operation and so cannot occur inside an immutable session. The id rule additionally requires the forwarded channel to be at permission $*$: a process with a restoration snapshot of **None** is itself providing a channel at permission $*$, and forwarding identifies the two.

$$\frac{(k \in L) \qquad \mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta \vdash_{\mathcal{N}} P :: (x : A_k)}{\mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta \vdash_{\mathcal{N}} (x.k\,;\,P) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus\mathrm{R}$$

$$\frac{(\forall \ell \in L) \qquad \mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta, x : \langle A_\ell, p, a\rangle \vdash_{\mathcal{N}} Q_\ell :: (z : C)}{\mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta, x : \langle \oplus\{\ell : A_\ell\}_{\ell \in L}, p, a\rangle \vdash_{\mathcal{N}} (\mathsf{case}\ x\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) :: (z : C)}\ \oplus\mathrm{L}$$

$$\frac{}{\mathcal{V}\,;\,\cdot\,;\,\cdot\,;\,\cdot \vdash_{\mathbf{None}} (\mathsf{close}\ x) :: (x : \mathbf{1})}\ \mathbf{1}\mathrm{R} \qquad \frac{}{\mathcal{V}\,;\,\cdot\,;\,\cdot\,;\,x : \langle A, *, a\rangle \vdash_{\mathbf{None}} (y \leftrightarrow x) :: (y : A)}\ \mathrm{id}$$

$$\frac{\begin{array}{c}\mathsf{proc}\ f[\overline{v}] : \overline{y' : T'} \vdash (x : A) = P \in \Sigma \qquad \overline{T'}[\overline{e}/\overline{v}] = \overline{T} \qquad (a\ \text{fresh}) \\ \mathcal{V}, a\,;\,\Theta\,;\,\Delta_L\,;\,\Delta, x : \langle A[\overline{e}/\overline{v}], *, a\rangle \vdash_{\mathcal{N}} Q :: (z : C) \qquad \mathcal{V} \vDash \overline{e}\end{array}}{\mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta, \overline{y : T} \vdash_{\mathcal{N}} (\{a\}\ x \leftarrow f[\overline{e}]\ \overline{y}\,;\,Q) :: (z : C)}\ \mathrm{spawn}$$

$$\frac{\overline{v}\,;\,\cdot\,;\,\cdot\,;\,\overline{y : T} \vdash_{\mathbf{None}} P :: (x : A)}{(\mathsf{proc}\ f[\overline{v}] : \overline{y : T} \vdash (x : A) = P)\ \mathsf{well\text{-}typed}}\ \mathrm{proc}$$

Fig. 4. Selected Basic Typing Rules

$$\frac{\mathcal{V} \vDash e \qquad e \neq * \qquad \mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta \vdash_{\mathcal{N}} P :: (x : A[e/k])}{\mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta \vdash_{\mathcal{N}} (\mathsf{send}\ x\ \{e\}\,;\,P) :: (x : \exists k.\ A)}\ \exists\mathrm{R}$$

$$\frac{\mathcal{V}, v\,;\,\Theta\,;\,\Delta_L\,;\,\Delta, x : \langle A[v/k], p, a\rangle \vdash_{\mathcal{N}} Q :: (z : C)}{\mathcal{V}\,;\,\Theta\,;\,\Delta_L\,;\,\Delta, x : \langle \exists k.\ A, p, a\rangle \vdash_{\mathcal{N}} (\{v\} \leftarrow \mathsf{recv}\ x\,;\,Q) :: (z : C)}\ \exists\mathrm{L}$$

Fig. 5. Selected Permission/Identifier Parametricity Typing Rules

The spawn rule types process spawning and instantiates a process f drawn from the global signature Σ (parametric in variables $\overline{v}$). Recall that Σ holds all process declarations present in the program, allowing for arbitrary mutual recursion. The spawn supplies arguments $\overline{e}$ for them, so the consumed channels $\overline{y}$ must match the instantiated types $\overline{T'}[\overline{e}/\overline{v}]$, including the permission and identifiers. The premise $\mathcal{V} \vDash \overline{e}$ checks that these arguments are in scope. The spawn allocates a fresh channel x which f provides, with permission $*$ and a fresh identifier a, after which the continuation Q proceeds using x at type $A[\overline{e}/\overline{v}]$. The permission $*$ expresses that a spawned process always begins fully owned. For a signature Σ to be valid, we require that each definition be well-typed, ensuring that all spawned processes are well-typed. Formally, $\forall(\mathsf{proc}\ f[\overline{v}] : \overline{y : T} \vdash (x : A) = P) \in \Sigma$, we require $(\mathsf{proc}\ f[\overline{v}] : \overline{y : T} \vdash (x : A) = P)$ well-typed. The proc rule shows how to check this for a given definition.

Finally, none of the basic rules may touch the frozen context Δ_L: a channel in Δ_L can neither be communicated on nor sent over a $\otimes$ or $\multimap$. This is what enforces that an immutable session leaves its frozen channels untouched.

3.2 Permission/Identifier Parametricity Rules

These rules, realizing the quantifiers $\exists$ and $\forall$, govern how processes exchange permissions and identifiers. Such exchange is needed in two situations. First, before exchanging a channel, the sender often needs to first communicate its identifier. Second, a process that needs only immutable access to a channel can abstract over its exact fractional permission, whose precise value is irrelevant.

Figure 5 provides selected rules for the $\exists$ quantifier; the $\forall$ quantifier is dual and simply reverses the communication direction. In the $\exists$R rule, the construct '$\mathsf{send}\ x\ \{e\}$' sends the permission or

$$\frac{\Delta' = \{y : \langle \uparrow^I_M k.\, A', p', a' \rangle \in \Delta \mid p' \neq * \wedge a' \in \mathcal{O}\} \qquad \mathrm{vars}(\Delta') = \overline{y} \quad \mathcal{V}, p\,;\,\cdot\,;\,\Delta \setminus \Delta'\,;\,\Delta' \cdot p \vdash_{\mathbf{Some}(\Delta' \cdot p, \uparrow^I_M k.\, A, \mathcal{O}, \overline{y}, p)} P :: (x : A[p/k])}{\mathcal{V}\,;\,\mathcal{O}\,;\,\cdot\,;\,\Delta \vdash_{\mathbf{None}} (\mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}) :: (x : \uparrow^I_M k.\, A)}\ \uparrow^I_M \mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \langle A[\mathrm{unstar}(p)/k], p, a \rangle \vdash_{\mathcal{N}} Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \langle \uparrow^I_M k.\, A, p, a \rangle \vdash_{\mathcal{N}} (\mathsf{start}\ x\{p\}\,;\,Q) :: (z : C)}\ \uparrow^I_M \mathrm{L}$$

$$\frac{\mathrm{vars}(\Delta) = \overline{y} \qquad \Delta_c = \Delta[\overline{y}_c/\overline{y}]}{\mathcal{V}\,;\,\cdot\,;\,\Delta_L\,;\,\Delta \vdash_{\mathbf{Some}(\Delta_c, \uparrow^I_M k.\, A_c, \mathcal{O}_c, \overline{y}_c, p_c)} (\mathsf{continue}\ \overline{y}) :: (x : \downarrow^I_M \uparrow^I_M k.\, A_c)}\ \downarrow^I_M \mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \langle A, p, a \rangle \vdash_{\mathcal{N}} Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \langle \downarrow^I_M A, p, a \rangle \vdash_{\mathcal{N}} (\mathsf{finish}\ x\,;\,Q) :: (z : C)}\ \downarrow^I_M \mathrm{L}$$

$$\frac{\mathcal{V} \setminus \{p_c\}\,;\,\mathcal{O} \cup \mathcal{O}_c\,;\,\cdot\,;\,\Delta[1/p_c], \Delta_L \vdash_{\mathsf{None}} P :: (x : A[1/p_c])}{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta \vdash_{\mathbf{Some}(\Delta_c, A_c, \mathcal{O}_c, \overline{y}_c, p_c)} (\mathsf{mut}\ \{P\}) :: (x : \Downarrow^I_M A)}\ \Downarrow^I_M \mathrm{R}$$

$$\frac{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \langle A, *, a \rangle \vdash_{\mathcal{N}} P :: (z : C)}{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \langle \Downarrow^I_M A, *, a \rangle \vdash_{\mathcal{N}} (\mathsf{mutate}\ x\,;\,P) :: (z : C)}\ \Downarrow^I_M \mathrm{L}$$

Fig. 6. Typing Rules for Immutable Operations

identifier e on provided channel x, whose type updates to $A[e/k]$. We require e is well-formed under $\mathcal{V}$ and that it is not equal to $*$. This allows us to assume that all received permissions must be fractional, and therefore any channel with a parametric permission can safely be split and merged. The '$\{b\} \leftarrow \mathsf{recv}\ x$' construct in the $\exists$L rule dually receives a permission or identifier on used channel x, which is then bound to the name v. v is then substituted for k in x's type A.

3.3 Immutability Rules

The immutability rules govern the transitions into and out of an immutable session, and so concern the three modal operators $\uparrow^I_M k.\, A$, $\downarrow^I_M A$, and $\Downarrow^I_M A$. The rules for all three operators are collected in Figure 6, and we describe each in turn.

Start Immutable Session. The $\uparrow^I_M$R rule types the provider-side construct $\mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}$, which begins an immutable session. Its conclusion requires the process to be outside any session already: the restoration snapshot is **None** and the frozen context is empty. The first premise computes Δ', the sub-context of Δ holding the channels that will form the session's immutable *state*. A channel qualifies only if it meets three conditions: it is at a $\uparrow^I_M$ type; it has a fractional (non-$*$) permission, and its identifier is owned (in $\mathcal{O}$). The first condition ensures the channel sits at a session boundary, so its provider holds *no temporary state*. The second and third together ensure it cannot be taken to permission $*$ and mutated during the session, since its identifier stays in $\mathcal{O}$, which is itself frozen for the duration (third premise). Every channel of Δ outside Δ' is then *frozen*, so the session can operate only on its recorded state Δ', and not carry any mutable operations on channels in $\Delta \setminus \Delta'$. The second premise $\mathrm{vars}(\Delta') = \overline{y}$ requires the programmer to list the channels of Δ' in an explicit order, which a later $\mathsf{continue}\ \overline{y}$ matches against when the session finishes.

The third premise types the continuation P, the body of the immutable session. It adds the permission variable p to $\mathcal{V}$, so P may refer to p and the provided type becomes $A[p/k]$. The owned

context is emptied, since no channel may be taken to full ownership while a session is in progress. The frozen context becomes $\Delta \setminus \Delta'$, so P may operate only on its recorded state Δ'. The restoration snapshot (on the turnstile) *records* this state: the context Δ, the type of $\uparrow^I_M k.\ A$ of x, the owned channels O, the channels $\overline{y}$ that form the state, and permission p. This restoration snapshot is later utilized at the end of a session (i.e., continue) or escape hatch (i.e., mut).

Why do both the active context and the restoration snapshot scale Δ' by p, written $\Delta' \cdot p$? Since FracST is able to distinguish between mutable and immutable operations, it is able to serve immutable operations in *parallel* using a *replicated-server semantics*. On receiving a start request, a shared provider at permission p splits off a p-fraction of each state channel in its children (i.e., channels in Δ') for the duration of the session, leaving the remainder for other concurrent requests. Allocating each channel a share proportional to p keeps the total permission of every channel summing to 1, as visualized in Figure 7. Formally, this notation means to multiply the permission of each channel in Δ' by p.

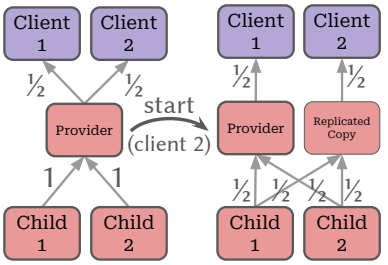


Fig. 7. Copies of replicated servers are allocated permission proportional to their provided permission

The client-side rule $\uparrow^I_M \mathrm{L}$ is the dual by which a client opens an immutable session on a channel x it uses, currently at type $\uparrow^I_M k.\ A$ with permission p. Issuing start $x\{p\}$ communicates the client's permission p to the provider, which the provider binds in $\uparrow^I_M \mathrm{R}$. In the continuation Q, the channel x advances past the $\uparrow^I_M$ boundary to its session body A, with k replaced by $\mathrm{unstar}(p)$: the operator $\mathrm{unstar}(\cdot)$ maps $*$ to 1 and leaves fractional permissions unchanged, keeping the substituted permission away from $*$.

End Immutable Session. The $\downarrow^I_M \mathrm{R}$ rule types the end of an immutable session from the provider's perspective by returning to its recorded start state. It applies only when the restoration snapshot holds a recorded state $\mathbf{Some}(\Delta_c, \uparrow^I_M k.\ A_c, O_c, \overline{y}_c, p_c)$ and the provider currently offers $\downarrow^I_M \uparrow^I_M k.\ A_c$: the down arrow $\downarrow^I_M$ applied to the very type $\uparrow^I_M k.\ A_c$ recorded when the session began. The premises then check that the current state matches the recorded one. Again, $\mathrm{vars}(\Delta) = \overline{y}$ ensures the programmer names the current state channels in order, and $\Delta_c = \Delta[\overline{y}_c/\overline{y}]$ requires that, after renaming them to the recorded names $\overline{y}_c$, they coincide with the recorded context Δ_c. In effect, 'continue $\overline{y}$' behaves like a tail-call to the matching immut block, with $\overline{y}$ supplying the arguments and the recorded $\overline{y}_c$ fixing their order.

The client-side rule $\downarrow^I_M \mathrm{L}$ is the dual: on 'finish x', used channel x at type $\downarrow^I_M A$ advances to A in the continuation, stepping past the $\downarrow^I_M$ boundary while leaving all contexts unchanged.

Escape Hatch. The $\Downarrow^I_M \mathrm{R}$ rule types the provider aborting an immutable session to return to a regular mutable session. Aborting discards the recorded state and reverts the restoration snapshot to **None**; the frozen channels Δ_L are merged back into the active context, and the owned set O_c stashed at the start of the session is restored to O. Because a channel can be mutated only when it is owned (permission $*$), the provider also learns that the client must therefore have sent 1 at $\uparrow^I_M$. As a result, we replace the recorded permission variable p_c by 1 throughout and drop it from $\mathcal{V}$, performing the substitution $[1/p_c]$ in the context and provided type.

This restriction, that only an owned channel may be mutated, is enforced on the client side by rule $\Downarrow^I_M \mathrm{L}$. It requires the used channel x to be at permission $*$ while typing 'mutate x' which advances the type of x from $\Downarrow^I_M A$ to A, leaving everything else unchanged.

$$\frac{p \neq * \qquad \mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x_1 : \left\langle \uparrow^I_M k.\ A, \frac{p}{2}, a \right\rangle, x_2 : \left\langle \uparrow^I_M k.\ A, \frac{p}{2}, a \right\rangle \vdash_{\mathcal{N}} Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \left\langle \uparrow^I_M k.\ A, p, a \right\rangle \vdash_{\mathcal{N}} (x_1\ x_2 \leftarrow \mathsf{split}\ x\,;\,Q) :: (z : C)}\ \text{split}$$

$$\frac{p \neq * \qquad q \neq * \qquad \mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \left\langle \uparrow^I_M k.\ A, p + q, a \right\rangle \vdash_{\mathcal{N}} Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x_1 : \left\langle \uparrow^I_M k.\ A, p, a \right\rangle, x_2 : \left\langle \uparrow^I_M k.\ A, q, a \right\rangle \vdash_{\mathcal{N}} (x \leftarrow \mathsf{merge}\ x_1\ x_2\,;\,Q) :: (z : C)}\ \text{merge}$$

$$\frac{\mathcal{V}\,;\,\mathcal{O}, a\,;\,\Delta_L\,;\,\Delta, x : \left\langle \uparrow^I_M k.\ A, 1, a \right\rangle \vdash_{\mathcal{N}} Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \left\langle \uparrow^I_M k.\ A, *, a \right\rangle \vdash_{\mathcal{N}} (\mathsf{share}\ x\,;\,Q) :: (z : C)}\ \text{share}$$

$$\frac{\mathcal{V}\,;\,\mathcal{O}\,;\,\Delta_L\,;\,\Delta, x : \left\langle \uparrow^I_M k.\ A, *, a \right\rangle \vdash_{\mathcal{N}} Q :: (z : C)}{\mathcal{V}\,;\,\mathcal{O}, a\,;\,\Delta_L\,;\,\Delta, x : \left\langle \uparrow^I_M k.\ A, 1, a \right\rangle \vdash_{\mathcal{N}} (\mathsf{own}\ x\,;\,Q) :: (z : C)}\ \text{own}$$

Fig. 8. Typing Rules for Permission Management

3.4 Sharing Rules

Finally, our last set of rules in Figure 8 govern management of the permission of channels. These include splitting and recombining permissions, along with transitioning permissions between $*$ and 1. Because permission handling is done entirely on the client side, these are client-side rules that manipulate channels being used. Additionally, the rules only operate on channels with a $\uparrow^I_M$ type, ensuring that a channel does not change permissions midway through an immutable operation.

The split rule consumes a used channel x with permission p (that must not be $*$) and splits it into two channels x_1 and x_2, each with permission $p/2$. Dually, the merge rule combines the permissions p of x_1 and q of x_2, consuming both the channels and producing channel x with permission $p + q$. Noteworthy here is that for x_1 and x_2 to merge, they must have the *same* identifier a, hence the need to track them in the type system. We additionally require that p and q must not be $*$.

The share rule transitions permission $*$ of channel x to 1 and adds its identifier a to $\mathcal{O}$. Dually, the own rule transitions x's permission from 1 to $*$ and also removes its identifier a from $\mathcal{O}$. Together, these dual rules ensure that each identifier is either attached to a channel at permission $*$ or is owned by being present in $\mathcal{O}$ (but never both).

4 Formal Semantics

The operational semantics of FracST is a multiset-rewriting system [13] whose runtime state is a configuration C: a collection of concurrent processes related by a single transition $C \mapsto C'$. Each process is written $\mathrm{proc}(S, \mathcal{M}, P)$, where S is the channel (or set of channels) it provides, $\mathcal{M}$ is its *mode* (described below), and P is its process expression.

Modes. At any point, each process is in one of four *modes*: **OM**, **OI**(P), **SM**(a, c), or **SI**(a). A mode records two independent bits of status: whether the process is **O**wned, providing its channel at permission $*$, or **S**hared, providing it at a fractional permission. The second bit can either be **M**utable outside any immutable session, or **I**mmutable, inside one. The four modes are exactly the combinations of these two choices. Each mode also carries some extra data its process needs. An **OI**(P) process stores the process expression P it will resume on a 'continue', namely the body of its immut block. The identifiers a on **SI**(a) and **SM**(a, c) processes are the same identifiers assigned in the type system. Recall that immutability uses a *replicated-server* model: an immutable provider duplicates itself to serve concurrent requests, represented as one **SM** process together with one **SI** process per active request. The shared identifier a links each **SI** process to the **SM** process it

$$(\oplus)\quad \mathsf{proc}(c, \mathcal{M}, c.k\,;\, P), \mathsf{proc}(d, \mathcal{M}', \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \mapsto \mathsf{proc}(c, \mathcal{M}, P), \mathsf{proc}(d, \mathcal{M}', Q_k)$$

$$(\mathsf{spawn})\quad \mathsf{proc}(d, \mathcal{M}, \{a\}\ x \leftarrow f[\overline{e}]\ \overline{c}\,;\, Q) \mapsto \mathsf{proc}(b, \mathbf{OM}, P[b/x][\overline{c}/\overline{y}][\overline{e}/\overline{v}]), \mathsf{proc}(d, \mathcal{M}, Q[b/x][a'/a])$$
$$(\mathsf{proc}\ f[\overline{v}] : \overline{y : T} \vdash (x : A) = P \in \Sigma) \qquad (b, a'\ \text{fresh})$$

$$(\mathsf{id})\quad \mathsf{proc}(c, \mathcal{M}, P), \mathsf{proc}(d, \mathbf{OM}, d \leftrightarrow c) \mapsto \mathsf{proc}(d, \mathcal{M}, P[d/c])$$

$$(\exists)\quad \mathsf{proc}(c, \mathcal{M}, \mathsf{send}\ c\ \{e\}\,;\, P), \mathsf{proc}(d, \mathcal{M}', \{v\} \leftarrow \mathsf{recv}\ c\,;\, Q) \mapsto \mathsf{proc}(c, \mathcal{M}, P), \mathsf{proc}(d, \mathcal{M}', Q[e/v])$$

Fig. 9. Selected Basic and Permission/Identifier Semantic Rules

was spawned from. Finally, an **SM** process also records a channel name c because, unlike the other modes, it may offer more than one channel, and c names the one used in its process expression.

Two of these modes, **SM** and **OI**, may at first seem to be somewhat contradictory, as in general we expect that a shared channel should be immutable and an owned channel should be mutable. What allows these modes to be meaningful is that the formal distinction between **M** modes and **I** modes is whether the process is *in an immutable session*. For **SM**, this means that **SM** processes are highly limited, as shared channels outside an immutable session can only undergo permission modifications (split, merge, and own) or start an immutable session. For **OI**, this just represents that an owned channel has the flexibility to still execute immutable operations.

4.1 Basic and Permission/Identifier Exchange Rules

We present the semantic rules in three groups, mirroring the typing rules of Section 3: this section covers the basic rules together with permission and identifier exchange, Section 4.2 presents the sharing rules, and Section 4.3 covers the immutability rules. Like their typing counterparts, the basic rules are inherited from standard intuitionistic linear session types and are largely indifferent to the process modes. Figure 9 shows a representative selection, and as before, we omit dual rules and defer the full set to the appendix.

Communication in FracST is *synchronous*, i.e., the sender and receiver both take a step simultaneously. The $(\oplus)$ rule illustrates the basic shape of communication: the provider $c.k$ sends a label k on c which is received by the client's case. The provider continues with P while the client proceeds with the selected branch Q_k. Its dual (&) and the channel-passing rules $(\otimes)$ and $(\multimap)$ are symmetric and deferred to the appendix. Like the corresponding typing rules, these communication rules are oblivious to the modes $\mathcal{M}$ and $\mathcal{M}'$ of the participating processes.

Again, the (spawn) and (id) rules are exceptions, requiring that both a newly spawned process and a terminating one are at mode **OM**. The (spawn) rule looks up f in the signature Σ and spawns a fresh process running its body P on a fresh channel b at mode **OM**, substituting the identifier/permission arguments $\overline{v}$ with $\overline{e}$ and channels $\overline{y}$ with $\overline{c}$. The newly spawned process at mode **OM** offers a fresh channel b which is substituted for x both in P and Q. Finally, the fresh channel b has a fresh identifier a' which is substituted for a in Q, but not in P! The reason is that FracST semantics are designed such that a provider remains oblivious of its offered channel's identifier, which is managed entirely by the client, thereby greatly simplifying the semantics rules and type safety proofs. The (id) rule forwards: a process $d \leftrightarrow c$ at mode **OM** identifies c with d, so the partner providing c continues as $P[d/c]$.

Finally, $(\exists)$ exchanges a permission or identifier in place of a channel: send $c\ \{e\}$ sends the expression e on c, which the receiver substitutes for v, leaving modes unchanged.

(share) $\mathsf{proc}(c, \mathbf{OM}, \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathsf{proc}(d, \mathcal{M}', \mathsf{share}\ c\ ;\ Q) \mapsto$
$\mathsf{proc}(\{c\}, \mathbf{SM}(a, c), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathsf{proc}(d, \mathcal{M}', Q)$
(a fresh)

(own) $\mathsf{proc}(\{c\}, \mathbf{SM}(a, c'), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathsf{proc}(d, \mathcal{M}', \mathsf{own}\ c\ ;\ Q) \mapsto$
$\mathsf{proc}(c, \mathbf{OM}, \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P[c/c']\}), \mathsf{proc}(d, \mathcal{M}', Q)$

(split) $\mathsf{proc}(\{c\} \cup \overline{cs}, \mathbf{SM}(a, c'), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathsf{proc}(d, \mathcal{M}', x_1\ x_2 \leftarrow \mathsf{split}\ c\ ;\ Q) \mapsto$
$\mathsf{proc}(\{c, c_2\} \cup \overline{cs}, \mathbf{SM}(a, c'), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathsf{proc}(d, \mathcal{M}', Q[c/x_1][c_2/x_2])$
(c_2 fresh)

(merge) $\mathsf{proc}(\{c_1, c_2\} \cup \overline{cs}, \mathbf{SM}(a, c'), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathsf{proc}(d, \mathcal{M}', x \leftarrow \mathsf{merge}\ c_1\ c_2\ ;\ Q) \mapsto$
$\mathsf{proc}(\{c_1\} \cup \overline{cs}, \mathbf{SM}(a, c'), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathsf{proc}(d, \mathcal{M}', Q[c_1/x])$

Fig. 10. Sharing Semantic Rules

4.2 Sharing Rules

The four sharing constructs manipulate permissions entirely on the client side, and in the semantics they manifest as transitions of the *provider* between modes, without ever stepping its process expression (Figure 10). Figure 11a visualizes these transitions: a provider (drawn as a node annotated with its mode) moves between owned and shared modes and gains or loses provided channels as the client issues each operation. The (share) rule moves a provider from **OM** to $\mathbf{SM}(a, c)$: a client's 'share c' shares the owned channel c, and the provider, still poised at its immut block, is assigned a fresh identifier a and begins offering the singleton set $\{c\}$ in shared mode. The (own) rule is its inverse: a client's 'own c' collapses an $\mathbf{SM}(a, c')$ provider back to **OM** on channel c, renaming the internal channel name c' to c in its expression P.

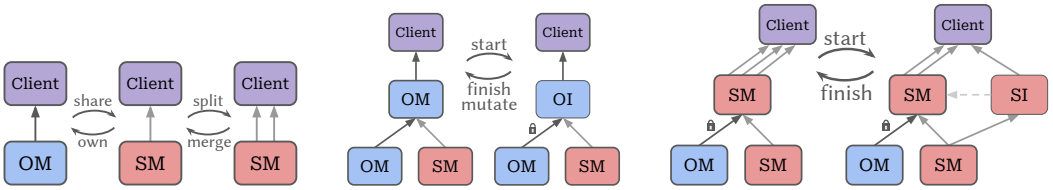
The (split) and (merge) rules instead manage the set of channels a shared (**SM**) provider offers. On '$x_1\ x_2 \leftarrow \mathsf{split}\ c$', the rule adds a fresh channel c_2 to the provider's set, and the client continues with x_1 bound to the original c and x_2 to c_2. Dually, on '$x \leftarrow \mathsf{merge}\ c_1\ c_2$', the rule drops c_2 from the set, and the client continues with x bound to the surviving c_1. Neither rule changes the provider's mode or expression.

Crucially, none of these rules advances the provider into its immut block; the provider stays poised at the block throughout. A client may therefore perform any number of these permission operations, in any order, before actually starting an immutable session, giving the programmer substantial flexibility. Finally, since the semantics does not track exact permission values, the channels offered by an **SM** process are interchangeable; this lets us arbitrarily keep the first named channel as the original in (split) and as the result in (merge).

4.3 Immutability Rules

The semantics of an immutable operation depends on whether its provider is *owned* or *shared*. Splitting the two cases avoids unnecessary copying, since an owned provider has a single client and need not serve concurrent requests, and it simplifies the treatment of the mut/mutate escape hatch. The semantic rules appear in Figure 12, with the two cases visualized in Figures 11b and 11c.

Owned. An owned provider has a single client and never serves concurrent requests, so its rules are simple; the provider's mode merely toggles between **OM** and **OI**, with no copying. Starting a session (rule $\uparrow^I_M$ **OM**) responds to a client's 'start $c\{*\}$' where the provider records its current immut block expression in its **OI** mode, so it can resume later, and steps into the body of this block. Because an owned provider is at permission $*$, it substitutes 1 for the permission variable p (as $*$ cannot be communicated) in expression P. Finishing (rule $\downarrow^I_M$**OI**) matches the provider's 'continue'



(a) Sharing (b) Owned process (c) Shared process

Fig. 11. Visualizations of sharing rules (a) and immutability rules for owned (b) and shared (c) processes.

$$(\uparrow^I_M \mathbf{OM}) \quad \mathrm{proc}(c, \mathbf{OM}, \mathrm{immut}\ \overline{y}\ \{p \Rightarrow P\}), \mathrm{proc}(d, \mathcal{M}', \mathrm{start}\ c\{*\}\,;\, Q) \mapsto \mathrm{proc}(c, \mathbf{OI}(\mathrm{immut}\ \overline{y}\ \{p \Rightarrow P\}), P[1/p]), \mathrm{proc}(d, \mathcal{M}', Q)$$

$$(\downarrow^I_M \mathbf{OI}) \quad \mathrm{proc}(c, \mathbf{OI}(\mathrm{immut}\ \overline{y}\ \{p \Rightarrow P_c\}), \mathrm{continue}\ \overline{b}), \mathrm{proc}(d, \mathcal{M}', \mathrm{finish}\ c\,;\, Q) \mapsto \mathrm{proc}(c, \mathbf{OM}, \mathrm{immut}\ \overline{y}\ \{p \Rightarrow P_c\}[\overline{b}/\overline{y}]), \mathrm{proc}(d, \mathcal{M}', Q)$$

$$(\Downarrow^I_M) \quad \mathrm{proc}(c, \mathbf{OI}(P_c), \mathrm{mut}\ \{P\}), \mathrm{proc}(d, \mathcal{M}', \mathrm{mutate}\ c\,;\, Q) \mapsto \mathrm{proc}(c, \mathbf{OM}, P), \mathrm{proc}(d, \mathcal{M}', Q)$$

$$(\uparrow^I_M \mathbf{SM}) \quad \mathrm{proc}(\{b_i\} \cup \overline{bs_i}, \mathbf{SM}(a_i, b_i''), \mathrm{immut}\ \overline{y_i}\ \{p' \Rightarrow P_i\}), \mathrm{proc}(\{c\} \cup \overline{cs}, \mathbf{SM}(a, c'), \mathrm{immut}\ \overline{b}\ \{p' \Rightarrow P\}), \mathrm{proc}(d, \mathcal{M}', \mathrm{start}\ c\{p\}\,;\, Q) \mapsto$$

(b_i' fresh)
$$\mathrm{proc}(\{b_i, b_i'\} \cup \overline{bs_i}, \mathbf{SM}(a_i, b_i''), \mathrm{immut}\ \overline{y_i}\ \{p' \Rightarrow P_i\}), \mathrm{proc}(\overline{cs}, \mathbf{SM}(a, c'), \mathrm{immut}\ \overline{b}\ \{p' \Rightarrow P\}), \mathrm{proc}(c, \mathbf{SI}(a), P[c/c'][\overline{b'}/\overline{b}][p/p']), \mathrm{proc}(d, \mathcal{M}', Q)$$

$$(\downarrow^I_M \mathbf{SI}) \quad \mathrm{proc}(\{b_i, b_i'\} \cup \overline{bs_i}, \mathbf{SM}(a_i, b_i''), \mathrm{immut}\ \overline{y_i}\ \{p' \Rightarrow P_i\}), \mathrm{proc}(\overline{cs}, \mathbf{SM}(a, c'), \mathrm{immut}\ \overline{b}\ \{p' \Rightarrow P\}), \mathrm{proc}(c, \mathbf{SI}(a), \mathrm{continue}\ \overline{b'}), \mathrm{proc}(d, \mathcal{M}', \mathrm{finish}\ c\,;\, Q) \mapsto \mathrm{proc}(\{b_i\} \cup \overline{bs_i}, \mathbf{SM}(a_i, b_i''), \mathrm{immut}\ \overline{y_i}\ \{p' \Rightarrow P_i\}), \mathrm{proc}(\{c\} \cup \overline{cs}, \mathbf{SM}(a, c'), \mathrm{immut}\ \overline{b}\ \{p' \Rightarrow P\}), \mathrm{proc}(d, \mathcal{M}', Q)$$

Fig. 12. Immutability Semantic Rules for Owned and Shared Processes

with the client's 'finish c': the provider returns to **OM** and resumes the recorded immut block, substituting the continue arguments $\overline{b}$ for the block's parameters $\overline{y}$. Aborting (rule $\Downarrow^I_M$) responds to 'mutate c' from the client while itself starting a mut block. The provider again returns to **OM** and proceeds with the body P of its mut block, discarding the recorded expression.

Figure 11b embodies these transitions: a single owned process toggles in place between its mutable (**OM**) and immutable (**OI**) modes, while keeping the modes of its children unchanged.

Shared. A shared provider must serve multiple requests in parallel, so its rules are more involved and follow the replicated-server model: the **SM** process acts as a server that *spawns* a fresh **SI** process for *each* immutable request.

Rule $\uparrow^I_M$ **SM** starts an immutable session when the provider receives a 'start $c\{p\}$' request from a client by spawning a new **SI** process. To give this **SI** process its own access to the provider's state, the rule splits off a fresh channel from each of the **SM** process's state channels (the children it operates on immutably). The $\uparrow^I_M$R typing rule and the configuration typing rules guarantee that every state channel of an **SM** process is itself provided by an **SM** process, so this splitting is always possible. The spawned process runs the immut block body with its provided channel renamed to c, its state channels rebound to the freshly split copies, and the permission variable instantiated to the constant p supplied by the client. Rule $\downarrow^I_M$**SI** describes finishing a session and is an exact reverse. It

$$\frac{(x \mapsto \langle *, a, O\rangle) \in \Psi \qquad (a \mapsto \mathbf{None}) \in \Psi \qquad \cdot\,;\, O\,;\, \cdot\,;\, \Delta \vdash_{\mathbf{None}} P :: (x : A)}{\Psi\,;\,\Gamma\,;\,\Delta', \Delta \Vdash (\mathrm{proc}(x, \mathbf{OM}, P)) :: (\Gamma, a\,;\,\Delta', x : \langle A, *, a\rangle)}\ \textsc{proc-OM}$$

$$\frac{\begin{array}{c}(x \mapsto \langle *, a, O\rangle) \in \Psi \qquad \cdot\,;\, O\,;\, \Delta_L\,;\, \Delta \vdash_{\mathbf{Some}(\Delta_c, A_c, O_c, \overline{y}_c, p_c)} P :: (x : A) \\ (a \mapsto \mathbf{Some}(\Delta_c, A_c, O_c, \overline{y}_c, p_c)) \in \Psi \qquad \cdot\,;\, O_c\,;\, \cdot\,;\, \Delta_L, \Delta_c \vdash_{\mathbf{None}} (\mathrm{immut}\ \overline{y}_c\ \{p_c \Rightarrow P_c\}) :: (x : A_c)\end{array}}{\Psi\,;\,\Gamma\,;\,\Delta', \Delta_L, \Delta \Vdash (\mathrm{proc}(x, \mathbf{OI}(\mathrm{immut}\ \overline{y}_c\ \{p_c \Rightarrow P_c\}), P)) :: (\Gamma, a\,;\,\Delta', x : \langle A, *, a\rangle)}\ \textsc{proc-OI}$$

$$\frac{\begin{array}{c}\forall i.\ (x_i \mapsto \langle p_i, a, \cdot\rangle) \in \Psi \qquad (a \mapsto \mathbf{Some}(\Delta, \uparrow^I_M k.\ A, O, \overline{y}, p')) \in \Psi \qquad \mathrm{vars}(\Delta) = \overline{y} \\ \cdot\,;\, O\,;\, \cdot\,;\, \Delta_L, \Delta \vdash_{\mathbf{None}} (\mathrm{immut}\ \overline{y}\ \{p' \Rightarrow P\}) :: (x : \uparrow^I_M k.\ A)\end{array}}{\Psi\,;\,\Gamma\,;\,\Delta', \Delta_L, \Delta \cdot \textstyle\sum_i p_i \Vdash (\mathrm{proc}(\{\overline{x_i}\}, \mathbf{SM}(a, x), \mathrm{immut}\ \overline{y}\ \{p' \Rightarrow P\})) :: (\Gamma, a\,;\,\Delta', x_i : \left\langle \uparrow^I_M k.\ A, p_i, a\right\rangle)}\ \textsc{proc-SM}$$

$$\frac{\begin{array}{c}(x \mapsto \langle p, a, O\rangle) \in \Psi \qquad p \neq 0 \qquad (a \mapsto \mathbf{Some}(\Delta_c, A_c, O_c, \overline{y}_c, p_c)) \in \Psi \\ \cdot\,;\, O\,;\, \Delta_L\,;\, \Delta \vdash_{\mathbf{Some}(\Delta_c \cdot p, A_c, O_c, \overline{y}_c, p_c)} P :: (x : A)\end{array}}{\Psi\,;\,\Gamma, a\,;\,\Delta', \Delta \Vdash (\mathrm{proc}(x, \mathbf{SI}(a), P)) :: (\Gamma, a\,;\,\Delta', x : \langle A, p, a\rangle)}\ \textsc{proc-SI}$$

Fig. 13. Selected Configuration Typing Rules

closes the **SI** process that is executing a 'continue $\overline{b'}$', merges the split-off channels back into the corresponding state channels, and returns the channel c to the **SM** process. Figure 11c depicts how the process graph changes across these two operations; the dotted edge from a spawned **SI** process marks the shared identifier it carries with the originating **SM** process.

For clarity, both rules are stated as if the state channels of the **SM** and **SI** processes come from *distinct* **SM** providers. In general, several may share a provider, so ($\uparrow^I_M$ **SM**) splits each channel separately and ($\downarrow^I_M$**SI**) merges each pair separately.

5 Type Safety

We prove the soundness of FracST using the standard theorems of progress (deadlock-freedom) and preservation. To state these theorems, we first lift typing to whole configurations. A runtime configuration is typed by the judgement $\Psi\,;\,\Gamma_1\,;\,\Delta_1 \Vdash C :: (\Gamma_2\,;\,\Delta_2)$, meaning that C uses the channels Δ_1 and identifiers Γ_1 and provides the channels Δ_2 and identifiers Γ_2. The map Ψ assigns each channel its permission, identifier, and O context, and assigns each identifier the restoration snapshot (on the turnstile of process typing) that every process offering it must use. The rules for this judgment are given in Figure 13. Each rule relies on the process typing judgment to construct a *singleton* configuration, i.e., a configuration containing a single process.

The rules are distinguished based on the mode the process is in. Remarkably, modes only appear in typing of configurations, and process typing stays mode-agnostic. The proc-**OM** rule types owned processes in mutable mode, where Ψ enforces that x is at permission $*$ and its identifier a has a **None** restoration snapshot associated with it. The third premise establishes P is well-typed and requires an empty frozen context since no channels are frozen in a mutable session. The proc-**OI** rule additionally checks that the immut-block stored in the mode (to be resumed on continue) itself typechecks against the recorded restoration context Δ_c, owned context O_c, and offered type A_c.

The proc-**SM** rule accounts for an **SM** process offering several channels. It is the only rule where the process is typed with a **None** restoration snapshot, but this is not what is recorded in Ψ. Instead, the restoration snapshot in Ψ is used to check the immut block body P, matching channels $\overline{y}$ and owned context O. The state channels Δ it consumes are scaled by the total permission of all offered channels, because an active **SI** process may have split off some permission to the children (Figure 7). The **SM** process is checked as if it held full permission to its children, which need not hold while immutable operations are in flight. Finally, the proc-**SI** rule types a shared provider in immutable mode, where the restoration snapshot is looked up in Ψ and scaled by the offered permission p to typecheck process expression P, mirroring the multiplication in $\uparrow^I_M$R.

Across all PROC-$\mathcal{M}$ rules, the $\mathcal{V}$ context is empty during typechecking. This reflects that runtime configurations contain no free permission or identifier variables, and all such variables must be substituted for constants.

*Why is there a premise $p \neq 0$ in PROC-**SI** rule?* An **SM** process may also offer no channels at all, or offer a channel at permission 0. This is the only situation in FracST where permission 0 arises because process typing never assigns a channel permission 0. Hence a permission-0 channel can be consumed only by an **SM** process that itself offers only permission-0 channels (or none). To account for this mismatch on permission-0 channels in configuration and process typing, we require $p \neq 0$ when typing an **SI** process. Figure 14 shows a common such case, where every channel of an **SM** process is currently being served by an **SI** process. The consistency judgement below reinforces this requirement on permission-0 channels.

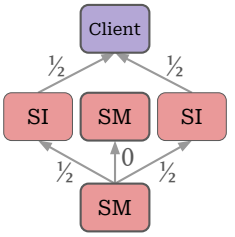


Fig. 14. Configuration with permission 0 channels

A configuration is also required to satisfy an additional judgement Ψ consistent, which holds when each identifier in Ψ falls into exactly one of the following three cases:

- offered by exactly one process at permission $*$ and not present in any $\mathcal{O}$ contexts, or
- offered by one or more processes at fractional permissions totalling 1 (*no* 0s) and in the restoration context for some identifier, or
- offered by one or more process at fractional permissions totalling 1, potentially *including* 0s, and in the $\mathcal{O}$ context for some other channel.

Together, these requirements ensure that permissions are never created or destroyed, and channel ownership is treated linearly. In addition, configuration typing permits two rules (that we skip here) for an empty set of processes, and composing two sets to form a bigger set. The full rules are given in the appendix.

We then establish a standard preservation theorem by induction on the transition judgement $C \mapsto C'$. It states that typing and consistency are preserved by every step of the semantics, i.e. a well-typed configuration remains well-typed after the step.

THEOREM 5.1 (PRESERVATION). *If* $\Psi\,;\,\Gamma_1\,;\,\Delta_1 \Vdash C :: (\Gamma_2, \Delta_2)$, $C \mapsto C'$, *and* Ψ consistent, *then there exist* Ψ' *and* Γ_2' *such that* $\Psi'\,;\,\Gamma_1\,;\,\Delta_1 \Vdash C' :: (\Gamma_2', \Delta_2)$ *and* Ψ' consistent.

The theorem allows for Γ_2 and Ψ to change to Γ_2' and Ψ', respectively. Since Γ_2 is the set of identifiers offered by any process, it grows when a process spawns and shrinks when one closes. Ψ changes whenever a process changes its permission or restoration snapshot, so rather than fixing it we only require that it remain consistent after each step.

To state progress, we need one more judgement: C poised [12, 42], which holds when every process in C is ready to communicate on the channel(s) it provides (full rules in the appendix). We prove progress by induction on the configuration typing judgement.

THEOREM 5.2 (PROGRESS). *If* $\Psi\,;\,\Gamma_1\,;\,\Delta_2 \Vdash C :: (\Gamma_2, \Delta_2)$ *and* Ψ consistent, *then either* C poised *or there exists* C' *such that* $C \mapsto C'$.

In particular, progress implies that FracST is *deadlock-free*: a well-typed configuration can always either take a step or is poised, waiting only on interaction with its external clients.

5.1 Immutability

Beyond type safety, the defining guarantee of FracST is *immutability*: a client that holds only a fractional reference to a channel can never change the provider behind it. This is built into the sematics: a shared provider sits poised at its immut block, splitting off a replica to serve each request, and then terminating it and merging back its permissions. Any temporary state of a session

therefore lives entirely in the ephemeral replica, with the original shared provider unchanged. In order to state this formally, we need the following auxiliary definitions, which we give precise rules for in the appendix:

Definition 5.3 (Uses). Given a typing derivation $\mathcal{D}$ of $\Psi\,;\,\Gamma_1\,;\,\Delta_1 \Vdash C :: (\Gamma_2, \Delta_2)$ and a process $\mathcal{P} = \mathsf{proc}(c, \mathcal{M}, P) \in C$, $\mathcal{D}$ must contain an application of the PROC-$\mathcal{M}$ rule (that is, the PROC-**OM** rule, the PROC-**OI** rule, the PROC-**SM** rule, or the PROC-**SI** rule as appropriate) for $\mathcal{P}$. Then we define $\mathsf{uses}(\mathcal{D}, C, \mathcal{P})$ as the Δ used in this application of the PROC-$\mathcal{M}$ rule. We will generally omit $\mathcal{D}$ and just write $\mathsf{uses}(C, \mathcal{P})$ when it is known that C is well-typed.

Definition 5.4 (Accessible State). The *accessible state* $\mathrm{acc}(C, \mathcal{P})$ of a process $\mathcal{P}$ in a well-typed configuration C is the subconfiguration containing the process itself, unioned with the accessible states of the processes offering all channels it uses.

Definition 5.5 (Matching). Two configurations C_1 and C_2 *match* if they are equal except that matching **SM** processes may offer different sets of channels to processes outside C_1 and C_2.

Intuitively, the accessible state of a process $\mathcal{P}$ in a configuration C can be thought of as the subtree of C rooted at $\mathcal{P}$. Then we can state the immutability theorem as follows:

THEOREM 5.6 (IMMUTABILITY). *If*

- $\Psi\,;\,\Gamma_1\,;\,\Delta_1 \Vdash C :: (\Gamma_2, \Delta_2)$,
- Ψ consistent,
- $C \mapsto C'$,
- $\mathcal{P} = \mathsf{proc}(\overline{cs}, \mathbf{SM}(a, x), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}) \in C$, *and*
- $\mathcal{P}' = \mathsf{proc}(\overline{cs'}, \mathbf{SM}(a, x), \mathsf{immut}\ \overline{y}\ \{p \Rightarrow P\}) \in C'$,

then $\mathrm{acc}(C, \mathcal{P})$ *and* $\mathrm{acc}(C', \mathcal{P}')$ *match.*

Note that the accessible state deliberately excludes processes reachable only through locked channels, i.e., those in the frozen context Δ_L. A well-typed process cannot communicate along a locked channel, so these processes are inaccessible to it. They may still take mutating steps, but such mutation cannot be observed until the channel is unlocked. And unlocking, by design, requires the state to be restored, so it does not affect immutability.

Matching must allow **SM** processes to offer different sets of channels to an outside client. An **SM** process may be shared with other clients, and these clients can perform split, merge, start, or finish operations on its channels, which change which channels it offers. The theorem shows that these operations cause no other change to the internal structure of the accessible state of $\mathcal{P}$.

To prove immutability, we first show that the accessible state of an **SM** process consists only of **SM** processes. We then show that the only change a step can make to such a subconfiguration is to the set of channels it offers externally, which matching allows. The full proof is in the appendix.

6 Implementation and Evaluation

We have implemented a prototype for FracST in OCaml, comprising a lexer, parser, and type checker. The implementation totals 1997 lines of OCaml together with 119 lines of Menhir parser specification. We highlight some details of type checking and reconstruction.

Type equality is at the heart of type checking, and is required for spawning, forwarding, and session delegation. Checking type equality requires deciding equality of the permission and identifier expressions that may occur in types, both of which can contain variables. Identifier variables are modeled as unique strings and compared syntactically. A permission is either $*$ or a linear term over permission variables, and two such terms are compared by normalizing, sorting on variable

names and matching the resulting coefficient lists. Session type equality itself is decided by the standard coinductive algorithm for structural type equality [14, 17].

Reconstruction. This is the most interesting aspect of our implementation and is key to minimizing the programmer burden imposed by FracST. Recall from Section 2 that FracST offers both an explicit and an implicit syntax. The reconstruction algorithm bridges them with two goals: *(i)* eliminate explicit uses of the 'start', 'finish', 'mutate', 'share', and 'own' constructs, and *(ii)* eliminate explicit mention of identifiers.

To address the first goal, we sort most of the reconstructible constructs into two groups, *eager* and *lazy* [15]. Eager constructs ('mutate', 'finish', and 'own') are inserted as early as possible, i.e., as soon as the channel has a modal type. Dually, lazy constructs ('start' and 'share') are inserted as late as possible, i.e., just before performing an action on that channel. The 'own' construct is further restricted to appear only immediately after a 'merge' or on entering an 'immut' block, which prevents spuriously inserting it after every 'share'. This reconstruction is sound and complete with respect to the explicit syntax, with the mild restriction that a channel of $\downarrow^I_M$- or $\Downarrow^I_M$-type is never sent over another channel.

To address the second goal, we exploit a few common usage patterns for identifiers. The first is the type $\forall a.\ \langle A, p, a\rangle \multimap B$ and its dual $\exists a.\ \langle A, p, a\rangle \otimes B$, which require a process to send (dually, receive) an identifier and then a channel carrying that same identifier. The identifier to send is always inferable, because in the implicit syntax, whenever a process is about to send a channel where an identifier is expected, we identify that channel's identifier and insert a send of that channel's identifier. Dually, we insert a receive of an identifier before receiving a channel, binding the received identifier to a fresh name that cannot be referenced directly but is reconstructed automatically. If a programmer wishes to explicitly use an identifier in implicit syntax, they must receive the identifier manually and bind it to a name of their choice.

The remaining common patterns concern spawning. A process that takes channel arguments is generally parametric over their identifiers, which must then be supplied at each spawn, and the spawned process is itself given a fresh identifier that must be bound to a name. As above, these fresh identifiers are bound to names inaccessible to the programmer. We reconstruct the argument identifiers under the assumption that a process has as many identifier arguments as channel arguments and that the two correspond one-to-one in order. When this assumption does not hold, the programmer can always specify the identifiers manually instead.

6.1 Examples

To evaluate FracST, we used our implementation to build a range of data structures and protocols, whose full definitions are included in the supplementary material. We highlight three representative examples here: copyable base types, the MapReduce protocol, and an auction protocol.

Copyable Types. Larger data structures and protocols routinely need simple component types such as natural numbers. The usual session type for a unary natural number is

$$\text{nat} = \oplus\{\textbf{zero} : \mathbf{1}, \textbf{succ} : \text{nat}\}$$

This is an internal choice requiring the provider to either send **zero** and close, or send **succ** and continue as`nat` (a client therefore receives this sequence of **succ** labels followed by **zero**).

The trouble is that this type is inherently mutable: unless we own the `nat`, we cannot even read its value. We would instead like to read a `nat` while holding only a fractional permission to it,

which requires the ability to copy it. The type above does not permit this, so we refine it to

$$\begin{aligned}\text{nat} = \uparrow^I_M k.\ \&\{\textbf{val} : \Downarrow^I_M \text{mut_nat}, \qquad & \text{mut_nat} = \oplus\{\textbf{zero} : \mathbf{1},\\ \textbf{dup} : \exists a.\ \langle \text{nat}, *, a\rangle \otimes \downarrow^I_M \text{nat}\} \qquad & \textbf{succ} : \text{mut_nat}\}\end{aligned}$$

Now, `nat` is an immutable session that begins with $\uparrow^I_M$ and then offers an external choice. The **val** branch takes the escape hatch $\Downarrow^I_M$ down to `mut_nat`, the original mutable encoding, while the **dup** branch sends a fresh channel of type `nat` at full permission $*$ before continuing as `nat`.

Because `mut_nat` is exactly the old type, all existing code that operated on a strictly mutable `nat` still works unchanged through the **val** branch. The **dup** branch adds what we wanted: even holding only fractional permission, a client can obtain a fully owned copy of the number. A process providing this `nat` behaves as before in the **val** case, and in the **dup** case it builds a copy by recursively duplicating its children and then calling itself.

The same transformation applies to any data structure which is inherently copyable, i.e., represented by labels only (e.g. binary numbers). This recovers the effect of *unrestricted* channels [4, 49], while retaining fine-grained control over exactly when a process is mutable.

MapReduce. We demonstrate how immutability and mutability interact with channels in a distributed protocol with MapReduce where a worker is represented using the following type:

$$\begin{aligned}\text{worker} = \uparrow^I_M k.\ \&\{\textbf{submit} : \Downarrow^I_M \forall \text{in}.\ \langle \text{work}, *, \text{in}\rangle \multimap \exists \text{out}.\ \langle \text{result}, *, \text{out}\rangle \otimes \text{worker},\\ \textbf{count} : \exists a.\ \langle \text{nat}, *, a\rangle \otimes \downarrow^I_M \text{worker}\}\end{aligned}$$

On the **submit** label, the worker accepts a work item and returns a result. This is a mutable operation, since the worker maintains a running count of the items it has processed, which it reports on the **count** label. The **count** operation is *immutable*, and returns the value of the running count. We implement two kinds of worker that share this type and are therefore interchangeable. The first is a basic worker that processes individual work items directly, while the second is a coordinator that partitions incoming work among several child workers, and on an (immutable) **count** operation recurses into its children and returns the sum of their counts. Because a coordinator's children can themselves be coordinators, this scales to an arbitrarily deep tree of workers. This example demonstrates the *immutability cascading*: while a coordinator serves a **count** operation, its children are immutable as well and can likewise only be counted, and so on down the tree.

Auction Protocol. The most complex example we demonstrate is an auction modeled as follows:

$$\begin{aligned}\text{auction} = \uparrow^I_M k.\ \oplus\{\textbf{running} : \exists \text{max_bid}.\ \langle \text{nat}, *, \text{max_bid}\rangle \otimes \text{bidding},\\ \textbf{done} : \exists \text{winner}.\ \langle \text{nat}, *, \text{winner}\rangle \otimes \downarrow^I_M \text{auction}\}\\ \text{bidding} = \&\{\textbf{bid} : \Downarrow^I_M \forall \text{id}.\ \langle \text{nat}, *, \text{id}\rangle \multimap \forall \text{amount}.\ \langle \text{nat}, *, \text{amount}\rangle \multimap \text{auction}, \textbf{ok} : \downarrow^I_M \text{auction}\}\\ \text{buyer} = \forall a.\ \&\{\textbf{take} : \langle \text{auction}, *, a\rangle \multimap \langle \text{auction}, *, a\rangle \otimes \text{buyer},\\ \textbf{view} : \forall p.\ \langle \text{auction}, p, a\rangle \multimap \langle \text{auction}, p, a\rangle \otimes \mathbf{1}\}\end{aligned}$$

Reading the protocol, an auction is an immutable session that is either **running** or **done**. While running, it broadcasts the current maximum bid and then offers the bidding choice: a buyer with mutable access may **bid**, sending an identifier and an amount, while a buyer holding only fractional permission can take the **ok** branch and thus only pass. Once the auction is **done**, it broadcasts the identifier of the winner. Dually, a buyer is either handed ownership of the auction, in which case it can bid, or a fractional permission, in which case it can only view the broadcast state before closing. This example also shows how identifier names in a type, such as **max_bid** or **winner**, can document the role of a sent channel when its type alone does not.

In our implementation, the auction uses a queue to manage buyers' access to the auction, served by two processes: one grants mutable access serially, the other grants immutable access concurrently, and both handle an *arbitrary* number of buyers. This example exercises both the mutable (to bid) and immutable regimes (to view) of the auction protocol simultaneously.

The process granting concurrent immutable access is shown below in implicit syntax.

```
proc all_view{aucp} : (auc : <auction,aucp,auci>), (q : <buyer_queue,*,qi>) |-
                          (res : <auction,aucp,auci> * 1) =
  q.del;
  case q ( none => wait q; send res auc; close res
  | some => buyer <- recv q;
            auc1, auc2 <- split auc; rest <- all_view{0.5*aucp} auc1 q;
            buyer.view; send buyer {0.5*aucp}; send buyer auc2;
            auc2 <- recv buyer; wait buyer; auc1 <- recv rest; wait rest;
            auc <- merge auc1, auc2; send res auc; close res )
```

The process is parametric in the permission aucp at which it holds the auction, and it is given that auction auc together with a queue q of waiting buyers held at full permission. Its result channel res eventually hands the auction back at the same permission aucp. Execution begins by dequeuing a buyer with q.del and casing on the reply. If the queue is empty (**none**), there is no one left to serve: the process waits for the queue to close, returns the auction unchanged on res, and closes.

The interesting case is **some**, where a buyer channel is received and served. To do so, all_view first splits auc's permission, producing two references each at half the incoming permission aucp. It immediately spawns a recursive instance of all_view using auc1 and one half of aucp permission for the remaining queue, obtaining a handle rest. Because this spawn runs concurrently, the entire tail of the queue begins viewing at the same time as the current buyer. The current buyer is then served on the other half: the process selects **view**, sends the permission $0.5 \cdot$ aucp followed by the channel auc2, and the buyer reads the auction immutably at that fractional permission.

Finally, the process reassembles what it split. It receives auc2 back from the buyer and waits for the buyer to close, then receives auc1 back from the recursive call and waits for it to close. With both halves in hand, it recombines them into a single reference which is sent on res before closing. The net effect is that all buyers are served in parallel without any deadlocks or blocking.

Table 1 reports our benchmark suite covering all the examples so far and a few more. Stack provides the same interface as Queue but implements a last-in-first-out data structure. Binary Tree (unbalanced) supports a mutable insert and an immutable size operation. Leader Election arranges the processes in a chain, where each process interacts only with its neighbors. The processes pass a mutable message up and down the chain to find the process with the greatest id, whos is elected as the leader. AVL Tree is our largest and most complicated benchmark: it implements a self-balancing binary search tree and supports a mutable insert and an immutable find operation. As Table 1 demonstrates, reconstruction consistently generates a substantial fraction of the source (roughly 30–60% more lines in explicit syntax), confirming that the implicit syntax substantiallly lowers programmer burden. Typechecking and reconstruction both complete in under a millisecond on all benchmarks, so the permission and identifier reasoning imposes minimal practical overhead.

7 Related Work

Session types were introduced by Honda et al. [24–27] as a type discipline for structuring communication in concurrent systems, ensuring protocol fidelity, deadlock-freedom, and absence of communication errors. Subsequent work [11, 50] established a firm logical foundation for session

Program	LOC (Impl)	LOC (Expl)	Defs	TC (ms)	Recon (ms)
Stack	67	93	9	0.0509	0.0476
Queue	68	97	9	0.0475	0.0279
Binary Tree	77	109	9	0.0570	0.0377
Copyable Nats	85	119	10	0.0436	0.0132
Copyable Bins	90	116	8	0.0460	0.0189
Leader Election	120	161	16	0.0726	0.0144
MapReduce	136	216	16	0.0976	0.0288
Auction	291	384	24	0.5228	0.1148
AVL Tree	364	500	26	0.3457	0.0245

Table 1. Evaluation of FracST: LOC (Expl) is the explicit lines of code; LOC (Impl) is the implicit lines of code; Defs contains the number of type and process definitions; TC (ms) and Recon (ms) are typechecking and reconstruction times (in milliseconds)

types by providing a Curry-Howard correspondence with linear logic [18, 19]. Early work on relaxing linearity relied on introducing a notion of *unrestricted* channels [4], based on the $!A$ modality that introduces controlled replication in linear logic. The key idea here is that a server provides an unrestricted session that is cloned every time a new client attempts to interact with the server. This way, the server does support multiple clients simultaneously but they all have independent sessions that are essentially linear and do not share any resources with each other. This is achieved either by sharing thread names and session delegation [25], or adding linear/unrestricted qualifiers to session types [49], or globally accessible persistent processes [42]. A different way to relax linearity is the recent work of Saffrich et al. [46], who introduce a notion of *borrowing* for functional session types, building on ordered linear typing to let the ownership of a channel be temporarily split off and later returned, though such borrowed access remains exclusive rather than shared. Differently from FracST, none of these works support any form of shared mutable state.

Shared Session Types. Although we are not aware of any prior work on permission-based reasoning with session types, most closely related to FracST are works on shared session types. Balzer and Pfenning [2] introduced sharing into session types with SILL_S, adding modal operators $\uparrow^\mathsf{S}_\mathsf{L}$ and $\downarrow^\mathsf{S}_\mathsf{L}$ that *acquire* and *release* a shared channel. A client acquires a shared channel to gain exclusive linear access, communicates (and possibly mutates), then releases it back to shared mode. This acquire-release discipline gives a controlled form of shared mutability in which only one client interacts with a shared provider at a time. Unlike FracST, since SILL_S does not distinguish mutable from immutable operations, a client must acquire the process even for read-only interaction. As a result, the queue from Section 2 is typed as described in Figure 15a in SILL_S: the acquire-release discipline is necessary for insertion and deletion as they mutate state, but is an overkill for length. Recognizing the immutable nature of the **len** operation, FracST implements a replicated-server semantics to process such requests *in parallel*, while SILL_S grants access only one client at a time. Finally, clients in SILL_S blocked on acquire requests in a cyclic topology can deadlock. The deadlock limitation was lifted by later work [3] using a separate type system of partial orders on events to rule out cyclic dependencies, though these orders must be supplied manually by the programmer. The discipline was further extended with phased communication [47], where a channel may be released at a subtype of the type at which it was acquired.

In a different direction, Rocha and Caires [44] introduce πSS, a pure Propositions-as-Types interpretation of shared mutable state, obtained by extending classical linear logic with two dual modalities that type *reference cells*. A cell stores a *persistent* value, and a client manipulates it with three operations: *(i)* read atomically *copies* out the contents, *(ii)* wrt destructively replaces them, and *(iii)* free releases the usage and deallocates the cell if no usage remains. Importantly, cells

$$\begin{aligned}\text{queue}_A = \uparrow^{\mathsf{S}}_{\mathsf{L}} \& \{&\textbf{ins} : A \multimap \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{queue}_A,\\ &\textbf{del} : \oplus\{\textbf{some} : A \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{queue}_A,\\ &\qquad \textbf{none} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{queue}_A\},\\ &\textbf{len} : \text{nat} \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \text{queue}_A\}\end{aligned}$$

(a) The queue in SILL_{S}: type begins with an acquire ($\uparrow^{\mathsf{S}}_{\mathsf{L}}$), so every branch, even the immutable operation **len**, must release ($\downarrow^{\mathsf{S}}_{\mathsf{L}}$).

	SILL_{S}	πSSL	FracST
Deadlock Freedom	(✓)†	✓	✓
Race Freedom	✓	(✓)‡	✓
Parallel Reads	✗	✓	✓
Immutable Ops	✗	✗	✓
Lock-free Sharing	✗	✓	✓

(b) ✓/✗/(✓): property satisfied / absent / conditional. †via manual lock orders [3]; ‡unlocked access races.

Fig. 15. The SILL_{S} queue protocol and a comparison of shared session-typed systems.

can be aliased across threads (logically, co-contraction) where concurrent writes are reconciled by reducing to a *non-deterministic sum* of the resulting states [21].

For atomic critical sections, πSS is extended to πSSL [44], which adds first-class lock/unlock primitives ensuring that at most one thread interacts with a locked cell at a time. Remarkably, πSSL enjoys deadlock freedom despite permitting locks, a consequence of requiring two threads in a share to share at most one cell. Due to this, the sharing topology stays acyclic without any programmer-supplied lock orders (unlike SILL_{S}). Its successor CLASS [45] extends the store to higher-order and linear objects while preserving these guarantees. In contrast to FracST, none of these systems have a notion of an immutable operation: concurrent reads and writes to a shared cell interleave into a non-deterministic sum and may yield *racy* outcomes. The only remedy is to serialize them using locks, a requirement that is *not* enforced by the type system that permits using reference cells without locking. FracST instead types immutability directly and serves immutable requests through a replicated-server semantics, so many clients read concurrently, lock-free and race-free, while mutation requires full ownership. These comparisons are summarized in Table 15b.

Fractional Permissions. Fractional permissions, introduced by Boyland [8, 9], provide a principled way to manage shared access to mutable resources. In this model, permissions are represented as fractions: a full permission grants write access, while fractional permissions allow read-only access. Permissions can be split and recombined, enabling flexible sharing while preventing data races. These ideas have been widely adopted in verification frameworks, particularly those based on separation logic [38, 39, 43]. However, their application has primarily focused on shared memory rather than message-passing systems. Bornat et al. [7] develop *permission accounting* within separation logic, providing a semantic framework for splitting and recombining permissions to enable concurrent reads and exclusive writes, and establishing fractional ownership as a quantitative resource. Haack et al. [20] build on top of Concurrent Separation Logic [10] to associate static access permissions with heap locations to rule out data races in multi-threaded Java programs. Crucially, they allow transfer of permissions between threads and enable modular reasoning about interference, including reentrant locks and thread synchronization. The Chalice verification system [33] introduces *implicit dynamic frames*, combining separation logic with permission accounting to verify object-oriented programs. Permissions allows Chalice to deduce upper bounds on location sets that can be modified by a method, which is exploited to guarantee data race freedom. To relieve the programmer from the burden of explicitly annotating locations with mathematical values for permissions, Heule et al. [23] define abstract read permissions that abstract away numeric fractions and allow programmers to reaon at the level of read and write permissions. VeriFast [28–30] uses ghost fields and fractional permissions to prove fork-join patterns porting Owicki-Gries reasoning [41] to dynamic threads, locks, and objects. Permission logics [22] generalize separation logic by treating permissions as first-class resources, forming the basis of verification frameworks such as Viper [37] . These systems support advanced features such as quantified permissions and iterated

separating conjunctions [36], enabling reasoning about complex data structures. Although these systems provide practical verification tools based on permissions, FracST lifts this interpretation to communication channels. This requires reconciling quantitative ownership with linear communication structure, which is not addressed in classical permission accounting. In particular, FracST needs to deal with *cascading* permissions across a session to establish that an immutable operation does not cause mutation in any of the processes in a configuration.

Ownership, Borrowing, and Aliasing Control. The *aliasing XOR mutability* principle of FracST has been inspired from the Rust programming language [32, 35]. The ownership and borrowing model of Rust enforces this discipline, ensuring that at any point either a value has a single mutable reference or multiple immutable references. This principle has been formalized in RustBelt [31], providing a semantic foundation for Rust's safety guarantees. More recently, Wagner et al. [51] distill borrowing to its essence, starting from a linear type system and incrementally adding immutable borrows, lexical lifetimes, reborrowing, and mutable borrows. FracST realizes the same discipline over communicating session-typed processes, but with a more generalized notion of immutability where state is temporarily mutated but eventually restored.

Numerous other works have also encoded fractional ownership into the type system for safety verification in the presence of aliasing and mutability. ConSORT [48] combines refinement types with fractional ownership to provide precise aliasing information for reference variables. In particular, reference types are annotated with a rational number in the range $[0, 1]$ indicating its ownership. Only references with ownership 1 can be mutated and any references with a mutable alias must have ownership 0. These types are extended with context- and flow-sensitive information, further improving the precision of the type system. This provides FracST with a meaningful direction for future research to encode flow-sensitivity to the type system, to further improve its precision. Marshall and Orchard [34] borrows the ideas of mutable, immutable, and partial borrows from Rust to a functional setting by applying a form of grading based on fractional permissions. These ideas have been incorporated into the Granule programming language [40] that uses uniqueness types to represent ownership. Bierhoff and Aldrich [5, 6] combine typestate and aliasing information into a novel abstraction called access permissions to design a protocol-compliance checker. The checker tracks the degree of possible aliasing and permissions through method implementations. Plaid [1, 16] elevates these ideas to a permission-based programming language whose object model, type system, and runtime are all designed on the foundation of permissions. The key novelty is that objects can change structure over time, but these changes, along with aliasing information, is tracked in the type system. Although this type system bears resemblence to FracST, a unique advantage that FracST provides is temporary mutation and state restoration at session boundaries. Moreover, FracST allows splitting and recombining of permissions through special language constructs, a property not supported in Plaid.

8 Conclusion

This paper introduces fractional permission-based reasoning to a session-typed system. To the best of our knowledge, FracST is the first work to explore mutability in the context of message-passing, particularly session types. To that end, we introduced a unique notion of an immutable session, which is demarcated by modal type operators. State can temporarily be mutated within a session but must be restored at session boundaries, a condition that is fully tracked and enforced by the type system. Through mutable and immutable references to a channel, FracST ensures that a process can either offer a single mutable reference or multiple immutable references to a channel. Owing to this discipline (despite sharing), FracST guarantees type preservation and deadlock-freedom.

9 Data-Availability Statement

This paper is supplemented with an artifact that contains the implementation of the FracST typechecker, along with the FracST programs for each of the examples mentioned in Table 1. It is available at https://github.com/selenexwu/FracST.

References

[1] Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. 2011. Plaid: a permission-based programming language. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 183–184. doi:10.1145/2048147.2048197

[2] Stephanie Balzer and Frank Pfenning. 2017. Manifest sharing with session types. *Proc. ACM Program. Lang.* 1, ICFP, Article 37 (Aug. 2017), 29 pages. doi:10.1145/3110281

[3] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 611–639.

[4] P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL '94)*. Springer-Verlag, Berlin, Heidelberg, 121–135.

[5] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 301–320. doi:10.1145/1297027.1297050

[6] Kevin Bierhoff and Jonathan Aldrich. 2008. PLURAL: checking protocol compliance under aliasing. In *Companion Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE Companion '08)*. Association for Computing Machinery, New York, NY, USA, 971–972. doi:10.1145/1370175.1370213

[7] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 259–270. doi:10.1145/1040305.1040327

[8] John Boyland. 2003. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 55–72.

[9] John Boyland. 2013. Fractional Permissions. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 270–288. doi:10.1007/978-3-642-36946-9_10

[10] Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theor. Comput. Sci.* 375, 1–3 (April 2007), 227–270. doi:10.1016/j.tcs.2006.12.034

[11] Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–236.

[12] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 760 (11 2014). doi:10.1017/S0960129514000218

[13] Iliano Cervesato and Andre Scedrov. 2006. Relating State-Based and Process-Based Concurrency through Linear Logic. *Electronic Notes in Theoretical Computer Science* 165 (2006), 145–176. doi:10.1016/j.entcs.2006.05.043 Proceedings of the 13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006).

[14] Ankush Das and Frank Pfenning. 2020. Session Types with Arithmetic Refinements. In *31st International Conference on Concurrency Theory (CONCUR 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 13:1–13:18. doi:10.4230/LIPIcs.CONCUR.2020.13

[15] Ankush Das and Frank Pfenning. 2020. Verified Linear Session-Typed Concurrent Programming. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 15 pages. doi:10.1145/3414080.3414087

[16] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Trans. Program. Lang. Syst.* 36, 4, Article 12 (Oct. 2014), 44 pages. doi:10.1145/2629609

[17] Simon Gay and Malcolm Hole. 2005. Subtyping for session types in the pi calculus. *Acta Informatica* 42, 2 (01 Nov 2005), 191–225. doi:10.1007/s00236-005-0177-z

[18] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. doi:10.1016/0304-3975(87)90045-4

[19] Jean-Yves Girard. 1995. Linear logic: its syntax and semantics. In *Proceedings of the Workshop on Advances in Linear Logic*. Cambridge University Press, USA, 1–42.

[20] Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. 2015. Permission-Based Separation Logic for Multithreaded Java Programs. *Logical Methods in Computer Science* Volume 11, Issue 1 (Feb. 2015). doi:10.2168/lmcs-11(1:2)2015

[21] Matthew Hennessy and Robin Milner. 1985. Algebraic laws for nondeterminism and concurrency. *J. ACM* 32, 1 (Jan. 1985), 137–161. doi:10.1145/2455.2460

[22] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. 2013. Verification condition generation for permission logics with abstract predicates and abstraction functions. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 451–476. doi:10.1007/978-3-642-39038-8_19

[23] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–334.

[24] Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.

[25] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

[26] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. *SIGPLAN Not.* 43, 1 (Jan. 2008), 273–284. doi:10.1145/1328897.1328472

[27] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1, Article 9 (March 2016), 67 pages. doi:10.1145/2827695

[28] Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 271–282. doi:10.1145/1926385.1926417

[29] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: a powerful, sound, predictable, fast verifier for C and java. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 41–55.

[30] Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A quick tour of the VeriFast program verifier. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems* (Shanghai, China) *(APLAS'10)*. Springer-Verlag, Berlin, Heidelberg, 304–311.

[31] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. doi:10.1145/3158154

[32] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.

[33] K. Rustan Leino, Peter Müller, and Jan Smans. 2009. *Verification of Concurrent Programs with Chalice*. Springer-Verlag, Berlin, Heidelberg, 195–222. https://doi.org/10.1007/978-3-642-03829-7_7

[34] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 131 (April 2024), 31 pages. doi:10.1145/3649848

[35] Nicholas D. Matsakis and Felix S. Klock. 2014. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (Portland, Oregon, USA) *(HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. doi:10.1145/2663171.2663188

[36] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 405–425. doi:10.1007/978-3-319-41528-4_22

[37] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583* (St. Petersburg, FL, USA) *(VMCAI 2016)*. Springer-Verlag, Berlin, Heidelberg, 41–62. doi:10.1007/978-3-662-49122-5_2

[38] Peter W. O'Hearn. 2004. Resources, Concurrency and Local Reasoning. In *CONCUR 2004 - Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–67. doi:10.1007/978-3-540-28644-8_4

[39] Peter W. OHearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1–3 (April 2007), 271–307. doi:10.1016/j.tcs.2006.12.035

[40] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. doi:10.1145/3341714

[41] Susan Owicki and David Gries. 1976. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM* 19, 5 (May 1976), 279–285. doi:10.1145/360051.360224

[42] Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–22. doi:10.1007/978-3-662-46678-0_1

[43] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

[44] Pedro Rocha and Luís Caires. 2021. Propositions-as-types and shared state. *Proc. ACM Program. Lang.* 5, ICFP, Article 79 (Aug. 2021), 30 pages. doi:10.1145/3473584

[45] Pedro Rocha and Luís Caires. 2023. Safe Session-Based Concurrency with Shared Linear State. In *Programming Languages and Systems*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 421–450. doi:10.1007/978-3-031-30044-8_16

[46] Hannes Saffrich, Janek Spaderna, Peter Thiemann, and Vasco T. Vasconcelos. 2025. Borrowing from Session Types. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 395 (Oct. 2025), 28 pages. doi:10.1145/3763173

[47] Chuta Sano, Stephanie Balzer, and Frank Pfenning. 2021. Manifestly Phased Communication via Shared Session Types. In *Coordination Models and Languages: 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14–18, 2021, Proceedings* (Valletta, Malta). Springer-Verlag, Berlin, Heidelberg, 23–40. doi:10.1007/978-3-030-78142-2_2

[48] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 684–714. doi:10.1007/978-3-030-44914-8_25

[49] Vasco T. Vasconcelos. 2009. Fundamentals of Session Types. In *Formal Methods for Web Services: 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures*, Marco Bernardo, Luca Padovani, and Gianluigi Zavattaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–186. doi:10.1007/978-3-642-01918-0_4

[50] Philip Wadler. 2012. Propositions as sessions. *SIGPLAN Not.* 47, 9 (Sept. 2012), 273–286. doi:10.1145/2398856.2364568

[51] Andrew Wagner, Olek Gierczak, Brianna Marshall, John M. Li, and Amal Ahmed. 2025. From Linearity to Borrowing. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 415 (Oct. 2025), 27 pages. doi:10.1145/3764117