Ferrite: A Judgmental Embedding of Session Types in Rust

RUOFEI CHEN, Independent Researcher, Germany
STEPHANIE BALZER, Carnegie Mellon University, USA

This article introduces Ferrite, a shallow embedding of session types in Rust. In contrast to existing session type libraries and embeddings for mainstram languages Ferrite not only supports linear session types but also shared session types. Shared session types allow sharing (aliasing) of channels while preserving session fidelity (preservation) using type modalities for acquiring and releasing sessions. Ferrite adopts a propositions as types approach and encodes typing derivations as Rust functions, with the proof of successful type-checking manifesting as a Rust program. The encoding resides entirely within the safe fragment of Rust and makes use of type-level features to support an arbitrary-length linear resource context and recursive session types.

CCS Concepts: • **Theory of computation → Linear logic**; **Type theory**; • **Software and its engineering → Domain specific languages**; **Concurrent programming languages**.

Additional Key Words and Phrases: Session Types, Rust, DSL

ACM Reference Format:
Ruofei Chen and Stephanie Balzer. 2020. Ferrite: A Judgmental Embedding of Session Types in Rust. 1, 1 (May 2020), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Message-passing concurrency is a dominant concurrency paradigm, adopted by mainstream languages such as Erlang, Scala, Go, and Rust, putting the slogan *"to share memory by communicating rather than communicating by sharing memory"*[Gerrand 2010; Klabnik and Nichols 2018] into practice. In this setting, messages are exchanged over channels, which can be shared among several senders and recipients. Figure 1 provides an example in Rust. It sketches the main communication paths in Servo's canvas component [Servo 2020], with some code simplified. Servo is a browser engine under development that uses message-passing to parallelize tasks, such as DOM traversal and layout painting, which are executed sequentially in existing web browsers. The canvas component provides 2D graphic rendering services, allowing its clients to create new canvases and perform operations on a canvas such as moving the cursor, drawing lines and rectangles.

The component is implemented by the `CanvasPaintThread`, whose function `start` contains the main communication loop running in a separate thread (lines 9–17). This loop processes client requests received along `canvas_msg_receiver` and `create_receiver`, the receiving endpoints of the channels created prior to spawning the loop (lines 7–8). The channels are typed with the enumerations `ConstellationCanvasMsg` and `CanvasMsg`, defining messages for creating and terminating the canvas component and for executing operations on an individual canvas, respectively. Canvases are identified by an id, which is generated upon canvas creation (line 16) and stored in the thread's hash map `canvases` (line 4). Should a client request an invalid id, the failed assertion `expect("Bogus canvas id")` (line 20) will result in a `panic!`, causing the canvas component to crash and clients to deadlock when waiting for a response from the component. Such a reaction can be the result of a client terminating a canvas (line 13) while other clients are still trying to communicate with it.

Authors' addresses: Ruofei Chen, Independent Researcher, Germany, soares.chen@maybevoid.com; Stephanie Balzer, Carnegie Mellon University, USA.

```
enum CanvasMsg { Canvas2d(Canvas2dMsg, CanvasId), Close(CanvasId) }
  enum Canvas2dMsg { MoveTo(Point2D), LineTo(Point2D), ... }
enum ConstellationCanvasMsg { Create { id_sender: Sender<CanvasId>, size: Size2D } }
struct CanvasPaintThread { canvases: HashMap < CanvasId, CanvasData >, ... }
impl CanvasPaintThread { ...
  fn start() -> ( Sender < ConstellationCanvasMsg >, Sender < CanvasMsg > ) {
    let ( msg_sender, msg_receiver ) = channel();
    let ( create_sender, create_receiver ) = channel();
    thread::spawn( move || { loop { select! {
      recv ( canvas_msg_receiver ) -> { ...
        CanvasMsg::Canvas2d ( message, canvas_id ) => { ...
          Canvas2dMsg::MoveTo ( ref point ) => self.canvas(canvas_id).move_to(point), ... }
        CanvasMsg::Close ( canvas_id ) => canvas_paint_thread.canvases.remove(&canvas_id) } }
      recv ( create_receiver ) -> { ...
        ConstellationCanvasMsg::Create { id_sender, size } => {
          let canvas_id = ...; self.canvases.insert( canvas_id, CanvasData::new(size, ...) );
          id_sender.send(canvas_id).unwrap(); } } } });
    ( create_sender, msg_sender ) }
  fn canvas ( &mut self, canvas_id: CanvasId ) -> &mut CanvasData {
    self.canvases.get_mut(&canvas_id).expect("Bogus canvas id") } }
```

Fig. 1. Message-passing concurrency in Servo's canvas component (simplified for illustration purposes).

Although he code in Figure 1 uses a clever combination of enumerations to type channels and ownership to rule out races on the data sent along channels, the Rust type system is not expressive enough to enforce adherence to the intended *protocol* of message exchange and existence of a communication partner. The latter is a consequence of Rust's type system being *affine*, which permits *weakening*, i.e., "dropping of a resource". The dropping or premature closure of a channel, however, can result in a proliferation of `panic!` and thus cause an entire application to crash.

Session types [Honda 1993; Honda et al. 1998, 2008] were introduced to express the protocols of message exchange and their adherence at compile-time. The discovery of a Curry-Howard correspondence between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Caires et al. 2016; Toninho 2015; Toninho et al. 2013; Wadler 2012] gave session types a strong logical foundation, resulting in a linear treatment of channels and thus assurance of a communication partner. More recently, linear session types have been extended with shared session types to accommodate safe sharing (i.e., aliasing) of channels [Balzer and Pfenning 2017; Balzer et al. 2018, 2019], addressing the limitations of an exclusively linear treatment of channels and increasing the scope of applicability of session types to a multi-client scenario, such as the one in Figure 1.

For example, using linear and shared session types we can capture the protocols implicit in Figure 1 as follows:

$$\begin{aligned} &\mathsf{Canvas} && = (\mathtt{Canvas2dMsg} \triangleright \mathsf{Canvas}) \ \& \ \epsilon \\ &\mathsf{ConstellationCanvas} && = \uparrow^{\mathsf{S}}_{\mathsf{L}} \mathtt{Size2D} \triangleright \mathtt{CanvasId} \triangleleft \mathsf{Canvas} \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{ConstellationCanvas} \end{aligned}$$

The linear session type Canvas prescribes the protocol for performing operations on an individual canvas, the shared session type ConstellationCanvas the protocol for creating a new canvas. This setup allows multiple clients to concurrently create new canvases, but ensures that at any point in time there exists only one client per canvas. We use the language $\mathsf{SILL}_{\mathsf{R}}$, a formal session type language based on $\mathsf{SILL}_{\mathsf{S}}$ [Balzer and Pfenning 2017] that we extend with Rust constructs. Table 1 provides an overview of $\mathsf{SILL}_{\mathsf{R}}$'s connectives. These are the usual linear connectives for receiving and sending values and channels ($\triangleright$, $\triangleleft$, $\multimap$, $\otimes$) as well as external and internal choice ($\&$, $\oplus$). An external choice leaves the choice between its left and right option to the client, an internal choice leaves it to the provider. For example, Canvas provides the client the choice between sending a value of enumeration type `Canvas2dMsg`, after which the canvas recurs, or closing the canvas. Readers familiar with classical linear logic session types [Wadler 2012] may notice the absence

Table 1. Overview and semantics of session types in SILL_R and Ferrite.

SILL_R	**Ferrite**	**Description**
ϵ	`End`	Terminate session.
$\tau \triangleright A$	`ReceiveValue<T, A>`	Receive value of type τ, then continue as session type A.
$\tau \triangleleft A$	`SendValue<T, A>`	Send value of type τ, then continue as session type A.
$A \multimap B$	`ReceiveChannel<A, B>`	Receive channel of session type A, then continue as session type B.
$A \otimes B$	`SendChannel<A, B>`	Send channel of session type A, then continue as session type B.
$A \mathbin{\&} B$	`ExternalChoice<A, B>`	Receive label inl or inr, then continue as session type A or B, resp.
$A \oplus B$	`InternalChoice<A, B>`	Send label inl or inr, then continue as session type A or B, resp.
$\uparrow^S_L A$	`LinearToShared<A>`	Accept an acquire, then continue as a linear session type A.
$\downarrow^S_L S$	`SharedToLinear<S>`	Initiate a release, then continue as a shared session type S.

of linear negation. SILL_R adopts an intuitionistic, sequent calculus-based formulation [Caires and Pfenning 2010], which avoids explicit dualization of a connective by providing left and right rules.

Additionally, SILL_R comprises connectives to safely share channels ($\uparrow^S_L A$, $\downarrow^S_L S$). Since shared channels have a *sharing semantics* as opposed to a *copying semantics*, as is the case for the linear exponential, it is crucial for safety to ensure that the multiple clients interact with the shared component in *mutual exclusion* of each other. To this end, an *acquire-release* semantics is adopted for shared components such that a shared component must first be acquired prior to any interaction. When a client acquires a shared component by sending an acquire request along the component's shared channel, it obtains a linear channel to the component, becoming its unique owner. Once the client is done interacting with the component, it releases the linear channel, relinquishing the ownership and being left only with a shared channel to the component. Key to type safety is to establish acquire-release not as a mere programming idiom but to *manifest* it in the type structure [Balzer and Pfenning 2017] such that session types prescribe when to acquire and when to release. This is achieved with the modalities $\uparrow^S_L A$ and $\downarrow^S_L S$, denoting the begin and end of a critical section, respectively. For example, the session type ConstellationCanvas prescribes that a client must first acquire the canvas component before it can ask for a canvas to be created by sending values for the canvas' size (`Size2D`) and id (`CanvasId`). The component then returns to the client a linear channel to the new canvas (Canvas), initiates a release ($\downarrow^S_L$), and recurs to be available to the next client.

The benefits of session types for software development have led to the introduction of session type libraries and embeddings for languages such as Java [Hu et al. 2010, 2008], Scala [Scalas and Yoshida 2016], Haskell [Imai et al. 2010; Lindley and Morris 2016; Pucella and Tov 2008; Sackman and Eisenbach 2008], OCaml [Imai et al. 2019; Padovani 2017], and Rust [Jespersen et al. 2015; Kokke 2019]. This article introduces *Ferrite*, a shallow embedding of session types in Rust. Ferrite allows programmers to specify *linear* and *shared* session types and write message-passing programs that adhere to the specified protocol. Protocol adherence is guaranteed statically by the Rust compiler. Figure 2 shows the corresponding session type declarations for the above session types Canvas and ConstellationCanvas in Ferrite. Table 1 provides a mapping between SILL_R and Ferrite constructs. As we discuss in detail in Sections 2 and 5, Ferrite introduces a `Fix` type operator to support recursive session types; the type argument `Z` denotes the base case of the type application and thus the recursion point. In case of a shared recursive session types such as `ConstellationCanvas`, `Z` combines recursion with a release (`SharedToLinear<S>`).

A key contribution of Ferrite is the support of shared session types. Existing solutions focus on enforcing a linear treatment of sessions with varying guarantees, ranging from partial to dynamic

```
type Canvas = Fix <
  ExternalChoice <
    ReceiveValue < Canvas2dMsg, Z >
    End > >;
```

```
type ConstellationCanvas = LinearToShared <
  ReceiveValue < Size2D,
    SendValue < CanvasId,
      SendChannel < Canvas, Z > > > >;
```

Fig. 2. Canvas protocol specification in Ferrite, defining session types Canvas and ConstellationCanvas.

or static. Enforcing linearity statically in an affine host language posed a considerable challenge for the development of Ferrite. We adopt the idea of *lenses* [Foster et al. 2007; Pickering et al. 2017] from prior work [Imai et al. 2019, 2010] to support an arbitrary-length linear typing context with random access, but avoid explicit dualization of session types for higher-order channels thanks to our intuitionistic formalization. Another distinguishing characteristics of Ferrite is its *propositions as types* approach. Building on the Curry-Howard correspondence between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Wadler 2012], Ferrite encodes SILL_R typing judgments and derivations as Rust functions. A successfully type-checked Ferrite program thus manifests in a Rust program that is the actual SILL_S typing derivation and thus the proof of protocol adherence.

In summary, Ferrite is an embedded domain-specific language (EDSL) for writing session-typed programs in Rust, which supports

- shared and linear session types using adjoint modalities for acquiring and releasing sessions,
- arbitrary recursive session types using type-level recursion,
- arbitrary-length linear context using lenses from profunctor optics to support random access,
- input and output of channels (a.k.a.higher-order channels) in addition to values, and
- managed concurrency, shielding programmers from channel creation and thread allocation.

Remarkably, the Ferrite code base remains entirely within the safe fragment of Rust.

Outline: Section 2 provides a summary of the key ideas underlying Ferrite, with subsequent sections refining those. Section 3 introduces the Ferrite type system, focusing on its judgmental embedding and enforcement of linearity. Section 4 elaborates on Ferrite's dynamics, detailing the use of Rust channels to implement Ferrite channels. Section 5 explains how Ferrite addresses Rust's limited support of recursive data types to allow the definition of arbitrary recursive and shared session types. Section 6 provides a discussion of Ferrite's guarantees, design choices, and directions for future work, and Section 7 reviews related work. The Ferrite code base with examples is provided as supplementary materials. For convenient look-up of definitions we also include an appendix. We plan to submit Ferrite as an artifact.

2 KEY IDEAS

This section highlights the key ideas underlying Ferrite. Subsequent sections provide further details.

2.1 Judgmental Embedding

In SILL_R, the formal session type language that we use in this article to study linear and shared session types, a program type-checks if there exists a derivation using the SILL_R typing rules. Central to a typing rule is the notion of a *typing judgment*. For SILL_R, we use the judgment

$$\Gamma; \Delta \vdash \mathit{expr} :: A$$

to denote that expression *expr* has session type A, given the typing of free shared and linear channel variables in contexts Γ and Δ, respectively. Γ is a *structural* context, which permits exchange, weakening, and contraction. Δ is a *linear* context, which only permits exchange but neither weakening

Table 2. Judgmental embedding of SILL_R in Ferrite.

SILL_R	**Ferrite**	**Description**
$\Gamma\,;\cdot \vdash A$	`Session < C, A >`	Typing judgment for top-level session (i.e., closed program).
$\Gamma\,;\Delta \vdash A$	`PartialSession < C, A >`	Typing judgment for partial session.
Δ	`C : Context`	Linear context; explicitly encoded.
Γ	-	Shared context; delegated to Rust, but with clone semantics.
A	`A : Procotol`	Session type.

nor contraction. The significance of the absence of weakening and contraction is that it becomes possible to "drop a resource" (premature closure) and "duplicate a resource" (aliasing), respectively.

For example, to type-check session termination, SILL_R defines the following typing rules:

$$\frac{\Gamma\,;\,\Delta \vdash \mathit{cont} :: A}{\Gamma\,;\,\Delta,\, a : \epsilon \vdash \mathsf{wait}\, a;\, \mathit{cont} :: A}\ (\text{T-}\epsilon_\text{L}) \qquad \frac{}{\Gamma\,;\cdot \vdash \mathsf{terminate} :: \epsilon}\ (\text{T-}\epsilon_\text{R})$$

As mentioned in the previous section, we get a *left* and *right* rule for each connective because SILL_R adopts an intuitionistic, sequent calculus-based formulation [Caires and Pfenning 2010]. Whereas the left rule describes the session from the point of view of the *client*, the right rule describes the session from the point of view of the *provider*. We read the rules bottom-up, with the meaning that the premise denotes the continuation of the session. The left rule ($\text{T-}\epsilon_\text{L}$) indicates that the client is waiting for the linear session offered along channel a to terminate, and continues with its continuation *cont* once a is closed. The linear channel a is no longer available to the continuation due to the absence of contraction. The right rule ($\text{T-}\epsilon_\text{R}$) indicates termination of the provider and thus has no continuation. The absence of weakening forces the linear context Δ to be empty, making sure that all resources are consumed.

Given the notions of a typing judgment, typing rule, and typing derivation, we can get the Rust compiler to type-check SILL_R programs by encoding these notions as Rust programs. The basic idea underlying this encoding can be schematically described as follows:

$$\frac{\Gamma\,;\Delta_2 \vdash \mathit{cont} :: A_2}{\Gamma\,;\Delta_1 \vdash \mathit{expr}; \mathit{cont} :: A_1}$$

```
fn expr < C1: Context, C2: Context, A1: Protocol, A2: Protocol >
  ( cont : PartialSession < C2, A2 > )
  -> PartialSession < C1, A1 >
```

On the left we show a SILL_R typing rule, on the right its encoding in Ferrite. Ferrite encodes a SILL_R typing judgment as the Rust type `PartialSession<C, A>`, with `C` representing the linear context Δ and `A` representing the session type A. A SILL_R typing rule for an expression `expr`, is encoded as a Rust function `expr: Fn(PartialSession<C2, A2>) -> PartialSession<C1, A1>`, with the function *return* being the *conclusion* of the rule and the *argument* being its *premise*. The encoding uses a *continuation passing*-style representation where the premise is passed as the argument to the conclusion. This representation naturally arises from the sequent calculus-based formulation of SILL_R.

Table 2 provides an overview of the mapping between SILL_R notions and their Ferrite encodings; Section 3.1 elaborates on them further. Whereas Ferrite explicitly encodes the linear context Δ, it delegates the handling of the shared context Γ to Rust with the obligation that shared channels implement Rust's `Clone` trait to permit contraction. To type a closed program, Ferrite moreover defines the type `PartialSession<C, A>`, which stands for a SILL_R judgment with an empty linear context.

2.2 Linear Context

A immediate encoding of the linear context C would be a type-level list $C = (A_0, (A_1, (..., (A_{N-1}))))$ of session types A_i with an appropriate update operation. This encoding has the advantage that it

allows the context to be of arbitrary length, but the disadvantage that it imposes a fixed order on the context's elements, disallowing *exchange*. A further challenge for the encoding is the support of arbitrary channel names: programmers should be able to freely choose the names of channel variables and not be forced to comply with a predetermined naming scheme. Next, we sketch the high-level idea of how Ferrite supports an arbitrary-length linear context with random access to programmer-named channels while using an ordered, type-level list internally.

An important abstraction in pursuit of this goal is the notion of a *lens* [Foster et al. 2007; Pickering et al. 2017], present in earlier work [Imai et al. 2019, 2010], that generalizes access and modification of a data structure's component. We combine this abstraction with *de Bruijn levels* as nameless indexes into an ordered, type-level list representation of the linear context. Given an inductive trait definition of natural numbers as zero (`Z`) and successor (`S<N>`), we can now implement the trait `ContextLens<C, B1, B2>` for any natural number `N: Nat` such that the session type `B1` at the N-th position in the linear context `C` is replaced with the session type `B2` and the update linear context becomes the associated type `N::Target`. Schematically this encoding can be captured as follows:

$$\frac{\Gamma, i{:}N\,;\Delta_2, N{:}B_2, \Delta_2' \;\vdash\; \mathit{cont} :: A_2}{\Gamma, i{:}N\,;\Delta_1, N{:}B_1, \Delta_1' \;\vdash\; \mathit{expr}(i); \mathit{cont} :: A_1}$$

```
fn expr < N, C, A1: Protocol, B1: Protocol, ... >
  ( i: N, cont: PartialSession < C, A2 > )
  -> PartialSession < N::Target, A1 >
where N : ContextLens < C, B1, B2 >
```

We note that the index `N` amounts to the type of the variable `i` that the programmer chooses as a name for a channel in the linear context. Ferrite takes care of the mapping, thus supporting random access to programmer-named channels. The above encoding is simplified to illustrate the key idea, sections 3.2 and 3.3 provide further details, including the support of higher-order channels.

2.3 Recursive and Shared Session Types

Rust's support for recursive types is limited to recursive struct definitions of a known size. To circumvent this restriction and support arbitrary recursive session types, Ferrite introduces a type-level fixed-point combinator `Fix<F>` to obtain the fixed point of a type function `F`. Since Rust lacks higher-kinded types such as `Type -> Type`, we use *defunctionalization* [Reynolds 1972; Yallop and White 2014] by accepting any Rust type `F` implementing the trait `TypeApp` with a given associated type `F::Applied`. Schematically we can capture this encoding as follows; Section 5.1 provides further details:

```
trait TypeApp < X > { type Applied; }
```

```
struct Fix < F : TypeApp < Fix < F > > >
{ unfix : Box < F::Applied > }
```

Recursive types are also vital for encoding shared session types. In line with [Balzer et al. 2019], Ferrite restricts shared session types to be recursive, making sure that a shared component is continuously available. To guarantee preservation, recursive session types must be *strictly equi-synchronizing* [Balzer and Pfenning 2017; Balzer et al. 2019], requiring an acquired session to be released to the same type at which it was previously acquired. Ferrite easily enforces this invariant by defining a specialized trait `SharedTypeApp` which omits an implementation for `End`. Section 5.2 provides further details on the encoding.

3 TYPE SYSTEM

This section introduces the statics of Ferrite, detailing the encoding of SILL_R typing judgments and rules and linear context in Ferrite. Section 5 discusses recursive and shared session types.

3.1 Judgmental Embedding

A distinguishing characteristics of Ferrite is its *propositions as types* approach, yielding a direct correspondence between SILL_R notions and their Ferrite encodings. We introduced this correspondence in Section 2.1 (see Table 2), next we discuss it in more detail. To this end, let's consider the typing of value input. We remind the reader of Table 1 in Section 1, which provides a mapping between SILL_R and Ferrite session types. The interested reader can find a corresponding mapping on the term level in Table 3 in the appendix.

$$\frac{\Gamma,\, a : \tau\,;\Delta \vdash K :: A}{\Gamma\,;\Delta \vdash a \leftarrow \text{receive\_value}; K :: \tau \triangleright A}\ (\text{T} \triangleright_\text{R})$$

```
fn receive_value < T, C: Context, A: Protocol >
  ( cont : impl FnOnce
      ( T ) -> PartialSession < C, A > )
  -> PartialSession < C, ReceiveValue < T, A > >
```

The SILL_R right rule $(\text{T} \triangleright_\text{R})$ types the expression $a \leftarrow \text{receive\_value}; K$ as the session type $\tau \triangleright A$ and the continuation K as the session type A. Ferrite encodes this rule as the function `receive_value`, parameterized by a value type `T` (τ), a linear context `C` (Δ), and an offered session type `A`. The function yields a value of type `PartialSession< C, ReceiveValue<T, A> >`, the conclusion of the rule, given a closure of type `FnOnce(T) -> PartialSession<C, A>`, encapsulating the premise of the rule. The use of a closure reveals the continuation-passing-style of the encoding, where the received value of type `T` is passed to the continuation closure. The closure implements the `FnOnce(T)` trait, ensuring that it can only be called once.

`PartialSession<C, A>` is one of the core constructs of Ferrite that enables the judgmental embedding of SILL_R. A Rust value of type `PartialSession<C, A>` represents a Ferrite program that guarantees linear usage of session type channels in the linear context `C` and offers the linear session type `A`. A `PartialSession<C, A>` in Ferrite thus corresponds to a SILL_R typing judgment. The type parameters `C` and `A` are constrained to implement the traits `Context` and `Protocol`, two other Ferrite constructs representing a linear context and linear session type, respectively:

```
struct PartialSession
  < C: Context, A: Protocol > { ... }
```

```
trait Context { ... }
trait Protocol { ... }
```

For each SILL_R session type, Ferrite defines a corresponding Rust struct that implements the trait `Protocol`, yielding the listing shown in Table 1. Corresponding implementations for ϵ (`End`) and $\tau \triangleright A$ (`ReceiveValue<T, A>`) are shown below. When a session type is nested within another session type, such as in the case of `ReceiveValue<T, A>`, the constraint to implement `Protocol` is propagated to the inner session type, requiring `A` to implement `Protocol` too:

```
struct End { ... }
impl Protocol for End { ... }
```

```
struct ReceiveValue < T, A > { ... };
impl < A: Protocol > for
  ReceiveValue < T, A > { ... }
```

Whereas Ferrite delegates the handling of the shared context Γ to Rust with the obligation that shared channels implement Rust's trait `Clone` to permit contraction, it encodes the linear context Δ explicitly. Being affine, the Rust type system permits weakening, a structural property rejected by linear logic. Ferrite encodes a linear context as a type-level list of the form `(A0, (A1, ()))`, with all its type elements $\texttt{A}_i$ implementing `Protocol`. Using the unit type `()` for the empty list and the tuple constructor `(_,_)` for the cons cell, we can implement the `Context` trait inductively as follows:

```
impl Context for () { ... }
impl < A: Protocol, C: Context > Context for ( A, C ) { ... }
```

The use of a type-level list for encoding the linear context has the advantage that it allows the context to be of arbitrary length. Its disadvantage is that it imposes an order on the context's elements, disallowing exchange. In the next two sections, we discuss how to make up for the loss of exchange and support random access to context elements using lenses. In prepration of this discussion, we find it helpful to introduce a variant of SILL_R typing rules that use an ordered

context Σ instead of the linear context Δ and a lens for random access to the elements in Σ. We call this variant $\mathsf{SILL}_{\mathsf{R}\Sigma}$ and label $\mathsf{SILL}_{\mathsf{R}\Sigma}$ typing rules with the subscript Σ to set them apart from regular $\mathsf{SILL}_{\mathsf{R}}$ typing rules. In $\mathsf{SILL}_{\mathsf{R}\Sigma}$, the context Σ is inductively defined as follows, mirroring the inductive definition of `Context`:

$$\frac{}{\cdot\ \text{ctx}} \qquad \frac{A\ \text{sessiontype} \qquad \Sigma\ \text{ctx}}{(A, \Sigma)\ \text{ctx}}$$

To represent a closed program, that is a program without any free variables, Ferrite defines a type alias `Session<A>` for `PartialSession<C, A>` that is restricted to an empty linear context:

```
type Session < A > = PartialSession < (), A >;
```

A complete session type program in Ferrite is thus of type `Session<A>` and amounts to the $\mathsf{SILL}_{\mathsf{R}}$ typing derivation proving that the program adheres to the defined protocol.

As an illustration of how to program in Ferrite we use a "hello world"-style session type program that receives a string value as input, prints it to the screen, and then terminates. On the left, we show the corresponding program in $\mathsf{SILL}_{\mathsf{R}}$, on the right in Ferrite. The session type offered is of $\mathsf{SILL}_{\mathsf{R}}$ type `String` $\triangleright\ \epsilon$, which translates into the Ferrite type `ReceiveValue<String, End>`.

hello_provider : `String` $\triangleright\ \epsilon =$
  name $\leftarrow$ receive_value ;
  println "Hello, {}" *name* ;
  terminate ;

```
let hello_provider :
  Session < ReceiveValue < String, End > >
= receive_value ( | name | {
    println!("Hello, {}", name);
    terminate () });
```

The full derivation tree of the `hello_provider` Ferrite program is available in appendix B.1.

3.2 Linear Context

Next, we discuss how we establish exchange for our encoding of a linear context using the notion of a lens. In this section, we focus on updating the type of a channel in the linear context due to protocol transition, in the next section we address the addition or removal of a channel to and from the linear context to account for higher-order channel constructs such as $\multimap$ and $\otimes$.

We illustrate the update of the type of a channel in the linear context based on the left rule for value input. The rule updates the type of channel a from $\tau \triangleright A$ in the conclusion to A in the premise as a consequence of sending a value of type τ along channel a:

$$\frac{\Gamma\ ;\ \Delta, a : A\ \vdash\ K :: B}{\Gamma, x : \tau;\ \Delta, a : \tau \triangleright A\ \vdash\ \text{send\_value\_to}\ a\ x;\ K :: B}\ (\mathrm{T}\triangleright_{\mathsf{L}})$$

Following the same approach that we used for embedding ($\mathrm{T}\triangleright_{\mathsf{R}}$) in the previous section, we can sketch the encoding of ($\mathrm{T}\triangleright_{\mathsf{L}}$) as the function `send_value_to` shown below, with a few type holes prefixed with `?` to be filled in:

```
fn send_value_to < T, A: Protocol, ... >
  ( l: ?L, x: T, cont: PartialSession < ?C2, A > )
  -> PartialSession < ?C1, A >
```

In addition to the type holes `?C1` and `?C2` for the linear contexts $\Delta, a : \tau \triangleright A$ and $\Delta, a : A$ for the conclusion and premise of the rule, respectively, the encoding introduces the type hole `?L` as a means to access the channel `a` inside the linear contexts `?C1` and `?C2`.

A naive approach to filling in the type holes would simply be to plug in `(ReceiveValue<T, A>, C)` for `?C1` and `(A, C)` for `?C2`, while ignoring `?L`, as shown below:

```
fn send_value_to < T, C: Context, A: Protocol, B : Protocol >
  ( x: T, cont: PartialSession < (A, C), B > )
  -> PartialSession< ( ReceiveValue < T, A >, C ), B >
```

This approach, however, only works if the channel to be updated is at the head of the linear context, which is exactly the restriction that we want to lift. Let's instead make actual use of the type hole `?L`. What we need is a way to refer to an element in the context and to provide a transformation to be applied to that element. A *lens* [Foster et al. 2007; Pickering et al. 2017] exactly provides this capability: it is an abstraction that generalizes access and modification of a data structure's component. We thus extend $\mathsf{SILL}_{\mathsf{R}\Sigma}$ with the following lens judgment:

$$L \Rightarrow \phi(\Sigma, A, A', \Sigma')$$

The judgment indicates that type L implements the context lens ϕ, which provides access to a channel a in the linear context Σ to update a's type from A to A', resulting in the context Σ'.

Using a context lens ϕ, we can define the $\mathsf{SILL}_{\mathsf{R}\Sigma}$ -variant of rule ($\mathsf{T}\triangleright_{\mathsf{L}}$) as follows:

$$\frac{\Gamma, l : L\ ;\ \Sigma' \vdash K :: B \qquad L \Rightarrow \phi(\Sigma, \tau \triangleright A, A, \Sigma')}{\Gamma, l : L, x : \tau\ ;\ \Sigma \vdash \mathsf{send\_value\_to}\ l\ x; K :: B}\ (\mathsf{T}_{\Sigma}\triangleright_{\mathsf{L}})$$

While rule ($\mathsf{T}\triangleright_{\mathsf{L}}$) uses *exchange* to locate the channel $a : \tau \triangleright A$ in Δ, rule ($\mathsf{T}_{\Sigma}\triangleright_{\mathsf{L}}$) treats Σ opaquely and instead uses the context lens $\phi(\Sigma, \tau \triangleright A, A, \Sigma')$ provided by the type L to locate the channel $a : \tau \triangleright A$ in Σ. The continuation K is given the linear context Σ', without the inference rule having to know about the internals of Σ and Σ'.

Ferrite implements the context lens ϕ as the `ContextLens` trait shown below. The `ContextLens` trait is defined with three type parameters `C`, `A1`, and `A2` corresponding to Σ, A, and A' of ϕ, respectively. The updated context Σ' is defined as an *associated type* `Target` because it is determined by the other type parameters `C`, `A1`, and `A2`.

```
trait ContextLens < C: Context, A1: Protocol, A2: Protocol >
{ type Target: Context; ... }
```

Using the trait `ContextLens` we can now encode the inference rule ($\mathsf{T}_{\Sigma}\triangleright_{\mathsf{L}}$) as shown below. The type `L` of the first argument `l` implements the trait `ContextLens<C, ReceiveValue<T, A>, A>`, for updating a target channel in the linear context `C` from session type `ReceiveValue<T, A>` to `A`. The returned `PartialSession` uses the original linear context `C`, while the continuation in the argument `cont` uses the updated linear context, as specified by the associated type `L::Target`.

```
fn send_value_to
  < T, C: Context, A: Protocol, B: Protocol,
    L: ContextLens < C, ReceiveValue < T, A >, A > >
  ( l: L, x: T, cont: PartialSession < L::Target, B > )
  -> PartialSession< C, B >
```

What remains to be done is to associate with the lens the actual channel that is the target of the update. Since we represent the context Σ as an ordered, type-level list, an element in that list can be uniquely identified by its position. This suggests the idea to use type-level natural numbers as implementations of context lenses to access channels at their respective position:

$$\frac{}{\mathsf{Z} \Rightarrow \phi((A, \Sigma), A, A', (A', \Sigma))} \qquad \frac{N \Rightarrow \phi(\Sigma, A, A', \Sigma')}{\mathsf{S}(N) \Rightarrow \phi((B, \Sigma), A, A', (B, \Sigma'))}$$

The base case zero (Z) implements the context lens for accessing the first channel in a non-empty linear context. The inductive case successor ($S(N)$) states that for any given natural number N, if N implements $\phi(\Sigma, A, A', \Sigma')$, then $\mathsf{S}(N)$ implements $\phi((B, \Sigma), A, A', (B, \Sigma'))$. In other words, a natural number $S(N)$ can implement a context lens for a linear context (B, Σ), if it can delegate the access of the channel A in Σ to its predecessor N.

To finally implement trait `ContextLens` in Ferrite, we first define the natural numbers as the structs `Z` and `S<N>`. The structs have a copy semantics, indicated by the pragma `#[derive(Copy)]`, allowing their values to be used more than once. Since the types do not have any meaningful contents at the

value level, the struct `Z` has an empty body, while `S<N>` uses `PhantomData<N>` to discard the unused type argument `N`.

```
#[derive(Copy)] struct Z ();
#[derive(Copy)] struct S < N > ( PhantomData<N> );
```

The implementation of `ContextLens` using `Z` and `S<N>` is shown below, mirroring the inductive definition in $\text{SILL}_{\text{R}\Sigma}$ shown above. The base case `Z` implements the `ContextLens` for any linear context in the form of `(A1, C)`. The inductive case `S<N>` implements the `ContextLens< (B, C), A1, A2 >` for any `B`, provided that `N` implements the `ContextLens< C, A1, A2 >`. The `Target` associated type for `S<N>` is simply `N::Target` prepended with `B`.

```
impl < A1: Protocol, A2: Protocol,
       C: Context >
  ContextLens < (A1, C), A1, A2 >
  for Z
{ type Target = ( A2, C ); ... }
```

```
impl < A1: Protocol, A2: Protocol,
       B: Protocol, C: Context,
       N: ContextLens < C, A1, A2 > >
  ContextLens < (B, C), A1, A2 >
  for S < N >
{ type Target = ( B, N::Target ); ... }
```

3.3 Higher-Order Channels

The current definition of a context lens works well for accessing a channel in a linear context to update its session type. However we have not addressed how channels can be *removed* from or *added* to the linear context can be done using context lenses. These operations are required to account for higher-order channel constructs such as $\otimes$ and $\multimap$ or session termination.

3.3.1 Removal. To support channel removal, we introduce a special empty element $\emptyset$ to denote the *absence* of a channel at a particular position in the linear context Σ. Below, we show the SILL_{R} inference rule ($\text{T}\mathbf{1}_{\text{L}}$) for termination, which uses exchange for locating the channel $l{:}\epsilon$ for removal from Δ, and contrast it with the corresponding $\text{SILL}_{\text{R}\Sigma}$ rule ($\text{T}_{\Sigma}\mathbf{1}_{\text{L}}$), which uses a context lens to update $l{:}\epsilon$ in Σ to $l{:}\emptyset$ in Σ'.

$$\frac{\Gamma;\, \Delta \vdash K :: A}{\Gamma;\, \Delta, l : \epsilon \vdash \text{wait}\, l;\, K :: A}\ (\text{T}\mathbf{1}_{\text{L}}) \qquad \frac{\Gamma,\, n : N;\, \Sigma' \vdash K :: A \qquad N \Rightarrow \phi(\Sigma, \epsilon, \emptyset, \Sigma')}{\Gamma,\, n : N;\, \Sigma \vdash \text{wait}\, n;\, K :: A}\ (\text{T}_{\Sigma}\mathbf{1}_{\text{L}})$$

Ferrite implements $\emptyset$ as the `Empty` struct. To allow `Empty` to be present in a linear context, we introduce a new `Slot` trait and make both `Empty` and `Protocol` implement `Slot`. The original definition of `Context` is then updated to allow types that implement `Slot` instead of `Protocol`.

```
struct Empty { }
trait Slot { ... }
```

```
impl Slot for Empty { ... }
impl < A: Protocol > Slot for A { ... }
```

Using `Empty`, it is straightforward to implement rule ($\text{T}_{\Sigma}\mathbf{1}_{\text{L}}$) using a context lens that replaces a channel of session type `End` with the `Empty` slot:

```
fn wait < C: Context, A: Protocol, N: ContextLens < C, End, Empty > >
  ( n: N, cont: PartialSession < N::Target, A > )
  -> PartialSession < C, A >
```

The function `wait` does not really remove a slot from a linear context, but merely replaces the slot with `Empty`. As a result, an empty linear context may contain any number of `Empty` slots, such as `(Empty, (Empty, ()))`. We introduce a new `EmptyContext` trait to abstract over the different forms an empty linear context can take and provide an inductive definition as its implementation, with the empty list `()` as the base case. The inductive case `(Empty, C)` then is an empty linear context, if `C` is an empty context. Using the definition of an empty context, the right rule ($\text{T}_{\Sigma}\mathbf{1}_{\text{R}}$) can then be easily encoded as the function `terminate` shown below:

```
trait EmptyContext : Context { ... }
fn terminate < C: EmptyContext >
  () -> PartialSession < C, End >
```

```
impl EmptyContext for () { ... }
impl < C: EmptyContext >
  EmptyContext for ( Empty, C ) { ... }
```

3.3.2 Addition. The Ferrite function `wait` removes a channel from the linear context by replacing it with $\emptyset$. The function `receive_channel`, on the other hand, adds a new channel to the linear context. The SILL_R right rule ($\mathsf{T}{\multimap_\mathsf{R}}$) for channel input is shown below. It binds the received channel of session type A to the channel variable a and adds it to the linear context Δ of the continuation.

$$\frac{\Gamma\,;\,\Delta, a : A \vdash K :: B}{\Gamma\,;\,\Delta \vdash a \leftarrow \mathsf{receive\_channel}; K :: A \multimap B}\ (\mathsf{T}{\multimap_\mathsf{R}})$$

To encode ($\mathsf{T}{\multimap_\mathsf{R}}$) in $\mathsf{SILL}_{\mathsf{R}\Sigma}$, we need to first think of how to add a new channel to a linear context Σ. From the previous section we know that context lenses are implemented as natural numbers, serving as an index for accessing channels by their position in a linear context. With this in mind we want to avoid adding a new channel A to the front of a linear context Σ to form (A, Σ), as it would *invalidate* any existing context lenses used in the continuation. Instead, we want to define an operation for *appending* a new channel A to the *end* of a linear context Σ.

The append operation in $\mathsf{SILL}_{\mathsf{R}\Sigma}$ can be defined inductively by the rules shown below. We use the judgment $\Sigma * \Sigma' \doteq \Sigma''$ to denote that the result of appending Σ to Σ' is Σ''.

$$\frac{}{\cdot * \Sigma \doteq \Sigma} \qquad \frac{\Sigma * \Sigma' \doteq \Sigma''}{(A, \Sigma) * \Sigma' \doteq (A, \Sigma'')}$$

Similarly in Ferrite, the append operation is defined as the `AppendContext` trait shown below. The trait `AppendContext` is parameterized by a linear context `C`, has `Context` as its super-trait, and an associated type `Appended`. If a linear context `C1` implements the trait `AppendContext<C2>`, it means that the linear context `C2` can be appended to `C1`, with the associated type `C1::Appended` being the result of the append operation. The implementation of `AppendContext` follows the same inductive definition as in $\mathsf{SILL}_{\mathsf{R}\Sigma}$, with the empty list `()` implementing the base case and the cons cell `(A, C)` implementing the inductive case.

```
trait AppendContext < C: Context > : Context
{ type Appended: Context; ... }
impl < C: Context >
  AppendContext < C > for ()
{ type Appended = C; ... }
```

```
impl < A: Slot, C2: Context,
       C1: AppendContext < C2 > >
  AppendContext < C2 >
  for ( A, C1 )
{ type Appended = ( A, C1::Appended ); ... }
```

Using `AppendContext`, a channel `B` can be appended to the end of a linear context `C`, if `C` implements `AppendContext<(B, ())>`. The new linear context after the append operation is then given in the associated type `C::Appended`. At this point we know that the channel `B` can be found in `C::Appended`, but there is not yet any way to access channel `B` in `C::Appended`. To provide access, a context lens has first to be generated. We observe that the position of channel `B` in `C::Appended` is the same as the length of the original linear context `C`. In other words, the context lens for channel `B` in `C::Appended` can be generated by getting the length of `C`.

We first provide an inductive definition for the length of a linear context Σ in $\mathsf{SILL}_{\mathsf{R}\Sigma}$ using the type-level natural numbers defined earlier and $|\Sigma|$ to denote the size of the type-level list Σ.

$$\frac{}{|\cdot| \doteq \mathsf{Z}} \qquad \frac{|\Sigma| \doteq N}{|(A, \Sigma)| \doteq \mathsf{S}(N)}$$

In Ferrite, the length operation can be implemented by adding an associated type `Length` to the `Context` trait. Then the implementation of `Context` for `()` and `(A, C)` simply follows the same inductive definition as shown above in $\mathsf{SILL}_{\mathsf{R}\Sigma}$.

```
trait Context { type Length; ... }
impl Context for () { type Length = Z; ... }
```

```
impl < A: Slot, C: Context > Context for (A, C)
{ type Length = S < C::Length >; ... }
```

With the append and length operations in place, we can now define the right rule ($\mathsf{T}_\Sigma{\multimap_\mathsf{R}}$) in $\mathsf{SILL}_{\mathsf{R}\Sigma}$ as shown below. The rule uses $\Sigma * (A, \cdot)$ to append the channel A to Σ and identifies Σ' as

the result. Other than that it also uses $|\Sigma|$ to determine the length of Σ as N. The structural context Γ has a new variable a of type N added to it, and Σ' is used as the linear context.

$$\frac{\Gamma, a : N \,;\, \Sigma' \vdash K :: B \qquad |\Sigma| \doteq N \qquad \Sigma * (A, \cdot) \doteq \Sigma'}{\Gamma \,;\, \Sigma \vdash a \leftarrow \text{receive\_channel}; K :: A \multimap B} \; (\mathrm{T}_{\Sigma} \multimap_{\mathrm{R}})$$

Ferrite encodes the right rule ($\mathrm{T}_{\Sigma} \multimap_{\mathrm{R}}$) as the `receive_channel` function shown below. The function is parameterized by a linear context `C` implementing `AppendContext` to append the session type `A` to `C`. The continuation argument `cont` is a closure that is given a context lens `C::Length`, and returns a `PartialSession` with `C::Appended` as its linear context. The function returns a `PartialSession` with `C` being the linear context and offers a session of type `ReceiveChannel<A, B>`.

```
fn receive_channel
  < A: Protocol, B: Protocol, C: AppendContext <( A, () )> >
  ( cont: impl FnOnce ( C::Length ) -> PartialSession < C::Appended, B > )
  -> PartialSession < C, ReceiveChannel < A, B > >
```

The use of `ContextLens` and `send_value_to` can be demonstrated with an example client shown below. The program `hello_client` is written to communicate with the `hello_provider` program defined earlier in section 3.1. The communication is achieved by having `hello_client` offer the session type `ReceiveChannel < ReceiveValue<String, End>, End >`. Inside the body, `hello_client` uses `receive_channel` to receive a channel of session type `ReceiveValue<String, End>` provided by `hello_provider`. The continuation closure is given an argument `a` of a type `Z`, denoting the context lens generated by `receive_channel` for accessing the received channel in the linear context. Following that, the context lens `a: Z` is used for sending a string value, after which `hello_client` waits for `hello_provider` to terminate. We note that the type `Z` of channel `a` and thus the positioning of `a` within the context is not exposed to the user but managed internally by Ferrite.

hello_client : (String ▷ ϵ) $\multimap$ ϵ =
  $a \leftarrow$ receive_channel;
  send_value_to a "Alice";
  wait a;
  terminate;

```
let hello_client : Session <
  ReceiveChannel <
    ReceiveValue < String, End >, End > >
= receive_channel ( | a | {
    send_value_to ( a, "Alice".to_string(),
      wait ( a, terminate() ) ) });
```

The full derivation tree of the `hello_client` Ferrite program is available in appendix B.1.

3.4 Communication

At this point we have defined the necessary constructs to build and type check both `hello_provider` and `hello_client`, but the two are separate Ferrite programs that are yet to be linked with each other and executed. Ferrite provides a special construct `apply_channel` to facilitate such linking. The typing rule for `apply_channel` is as follows:

$$\frac{\Gamma \,;\, \cdot \vdash f :: A \multimap B \qquad \Gamma \,;\, \cdot \vdash a :: A}{\Gamma \,;\, \cdot \vdash \text{apply\_channel}\ f\ a :: B} \; (\text{T-APP})$$

```
fn apply_channel < A: Protocol, B: Protocol >
  ( f: Session < ReceiveChannel < A, B > >,
    a: Session < A > )
  -> Session < B >
```

The function `apply_channel` is defined as a construct that brings together *two* continuations `f` and `a`. The program `f` acts as the client expecting to receive a channel from a provider offering session type `A` and then continues as session type `B`. The program `a` acts as the provider offering session type `A`. Both `f` and `a` are linked by `apply_channel`, which sends the channel offered by `a` to `f` and then continues as `f`. The function `apply_channel` restricts `f` and `a` to be complete Ferrite programs with empty linear contexts, i.e., `f` and `a` cannot be closures with free channel variables.

To actually *execute* an entire Ferrite application, Ferrite provides the function `run_session`. The function `run_session` accepts a top-level Ferrite program of type `Session<End>`, which has an empty linear context, and runs it *asynchronously*.

```
async fn run_session ( session: Session < End > ) { ... }
```

The function `run_session` is the only public function Ferrite exposes to allow end users to run a Ferrite program. This means that partial Ferrite programs with free channel variables or Ferrite programs that offer session types other than `End` cannot be executed until they are linked to a main Ferrite program of type `Session<End>`. This restriction ensures that all channels created by a Ferrite application are indeed consumed. For example, the programs `hello_provider` and `hello_client` cannot be executed individually, but the linked program resulting from applying `hello_provider` to `hello_client` can be executed as shown below.

```
async fn main () { run_session ( apply_channel ( hello_client, hello_provider ) ).await; }
```

Relationship to Cut. Readers familiar with linear logic-based session types [Caires and Pfenning 2010; Wadler 2012] may wonder how `apply_channel` is related to *cut*. The cut rule is defined as follows, using the SILL_R syntax:

$$\frac{\Gamma\,;\,\Delta_1 \vdash a :: A \qquad \Gamma\,;\,\Delta_2, x : A \vdash b :: B}{\Gamma\,;\,\Delta_1, \Delta_2 \vdash x \leftarrow \text{cut}\ a\,;\,b :: B}\ (\text{T-CUT})$$

Compared to `apply_channel`, `cut` is less restrictive as it accepts arbitrary linear contexts Δ_1 and Δ_2, including non-empty ones, and an arbitrary session type A for the first premise. However, as we show in Appendix B.2, `apply_channel` is actually derivable using cut and does, as a result, not limit the expressiveness of Ferrite.

The benefit of supporting `apply_channel` rather than cut is the enforced linearization by introducing channels into a context subsequently, which uniquely determines the type-level list used internally in Ferrite for the linear context. Supporting cut directly in Ferrite would require the splitting of contexts for the premises of the cut rule, a type equation for which no unique solution may exist. For example, the equation $\Sigma_1 * \Sigma_2 \doteq (A_0, (A_1, (A_2, ())))$ has the solutions $\Sigma_1 \doteq (A_0, (A_1, (\emptyset, ())))$ and $\Sigma_2 \doteq (\emptyset, (\emptyset, (A_2, ())))$, $\Sigma_1' \doteq (A_0, (\emptyset, (A_2, ())))$ and $\Sigma_2' \doteq (\emptyset, (A_1, (\emptyset, ())))$, and $\Sigma_1'' \doteq (\emptyset, (A_1, (A_2, ())))$ and $\Sigma_2'' \doteq (A_0, (\emptyset, (\emptyset, ())))$ for splitting the context such that $|\Sigma_1| = 2$ and $|\Sigma_2| = 1$.

4 DYNAMICS

Section 3 introduced the type system of Ferrite, based on the constructs `End`, `ReceiveValue`, and `ReceiveChannel`. This section revisits those constructs and fills in the missing implementations to make the constructs executable, amounting to the *dynamic semantics* of Ferrite.

4.1 Rust Channels

Ferrite uses *Rust channels* as the basic building blocks for session type channels. A Rust channel is a pair of a sender and receiver, of type `Sender<P>` and `Receiver<P>`, respectively, denoting the two endpoints of the channel. The type parameter `P` is the *payload* type of the Rust channel, indicating the type of values that can be communicated over the channel.

Rust channels can be used for communication between two or more processes. For the communication to happen, we first need to decide what the payload type `P` should be and how to distribute the two endpoints among the processes. Internally, Ferrite adopts the convention to always give the sending endpoint of a Rust channel to the provider, and the receiving endpoint to the client.

We illustrate Ferrite's use of Rust channels based on the example below. The example shows a provider `int_provider` that offers to send an integer value and a client that is willing to receive an integer value. We choose `i32` as the payload type, allowing the provider to send a value of that type to the client.

```
fn int_provider ( sender: Sender<i32> )
{ sender.send(42); }
```

```
fn int_client ( receiver: Receiver<i32> )
{ let res = receiver.recv(); ... }
```

In the above example, the polarities of the session types of the provider and the client comply with our convention to give the sending endpoint of a channel to the provider and the receiving endpoint to the client. However, this setup cannot generally be expected, as demonstrated by the example below, in which a provider offers to *receive* an integer value and a client to send such value. To address the mismatch we must *reverse the polarity* of the provider and the client such that the former receives and the latter sends. We can easily achieve this by changing the payload type from `i32` to `Sender<i32>`, allowing the provider to send the sending endpoint of a newly created channel with reverse polarity to the receiver. This is demonstrated in the code below where the provider `receive_int_provider` first creates another Rust channel pair of payload type `i32`, and sends the sending endpoint of type `Sender<i32>` to the client `receive_int_client`. It then uses the receiving endpoint to receive the integer value sent by the client.

```
fn receive_int_provider
  ( sender: Sender < Sender<i32> > )
{ let (sender2, receiver) = channel();
  sender.send(sender2);
  let res = receiver.recv(); ... }
```

```
fn receive_int_client
  ( receiver: Receiver < Sender<i32> > )
{ let sender = receiver.recv().unwrap();
  sender.send(42); }
```

Given this brief introduction to Rust channels and the use of channel nesting for polarity reversal, we next address how these ideas are combined to implement the dynamics of Ferrite session types. Basically, the dynamics of a Ferrite session type is implemented by a Rust channel whose payload is of type `Protocol`. We recall from Section 3.1 that all protocols except for `End` are parameterized with the session type of their continuation. To provide an implementation that properly reflects the state transitions of a protocol induced by communication, it is essential to create a fresh channel to be used for communication by the continuation. For each implementation of a session type, it must be determined what the payload type of the channel to be used for the continuation should be. Again, the convention is to associate the sending endpoint with the provider, which may necessitate channel nesting if polarity reversal is required.

We illustrate this idea below, showing the implementations of the Ferrite protocols `SendValue<T, A>` and `ReceiveValue<T, A>`. `SendValue<T, A>` indicates that it sends a value of type `T` as well as a `Receiver<A>` endpoint for the continuation to be used by the client. `ReceiveValue<T, A>`, on the other hand, uses channel nesting for polarity reversal both for the transmission of a value of type `T` and the continuation type `A`.

```
struct SendValue < T, A >
( T, Receiver < A > );
```

```
struct ReceiveValue < T, A >
( Sender < ( T, Sender < A > ) > );
```

The examples above illustrate that it can become quite mind-boggling to determine the correct typing of channels. We emphasize that a Ferrite user is completely *shielded* from this complexity as Ferrite autonomously handles channel creation and thread spawning.

The linear context of a Ferrite program comprises the receiving endpoints of the channels implementing the session types. We define the associated types `Endpoint` and `Endpoints` for the `Slot` and `Context` traits, respectively, as shown below. From the client's perspective, a non-empty slot of session type `A` has the `Endpoint` type `Receiver<A>`. The `Endpoints` type of a `Context` is then a type level list of slot `Endpoints`.

```
trait Slot { type Endpoint : Send; }
impl Slot for Empty
  { type Endpoint = (); }
impl < A: Protocol > Slot for A
  { type Endpoint = Receiver<A>; }
```

```
trait Context { type Endpoints; ... }
impl Context for () { type Endpoints = (); ... }
impl < A: Slot, R: Context > Context
  for ( A, R ) { type Endpoints =
    ( A::Endpoint, R::Endpoints ); ... }
```

4.2 Session Dynamics

Ferrite generates session type programs by composing `PartialSession` objects generated by term constructors such as `send_value`. The `PartialSession` struct contains an internal `executor` field as shown below, for executing the constructed Ferrite program. The executor is a Rust *async closure* that accepts two arguments – the endpoints for the linear context `C::Endpoints` and the sender for the offered session type `Sender<A>`. The closure then executes asynchronously by returning a *future* with unit return type.

```
struct PartialSession < C: Context, A: Protocol >
{ executor : Box < dyn FnOnce( C::Endpoints, Sender < A > )
    -> Pin < Box < dyn Future < Output=() > > > > }
```

Ferrite keeps the `executor` field private within the library to prevent end users from constructing new `PartialSession` values or running the `executor` closure. This is because the creation and execution of `PartialSession` may be unsafe. The code below shows two examples of unsafe (i.e., non-linear) usage of `PartialSession`. On the left, a Ferrite program `p1` of type `Session< SendValue<String, End> >` is constructed, but in the `executor` closure both the linear context and the sender are ignored. As a result, `p1` violates the linearity constraint of session types and never sends any string value or signal for termination. On the right, an example client is shown, which calls a Ferrite program `p2` of type `ReceiveValue<String, End>` by directly running its `executor`. The client creates a Rust channel pair but ignores the receiver end of the channel, and then executes `p2` by providing the sender end. Because the receiver end is dropped, `p2` fails to receive any value, and the program results in a deadlock.

```
let p1 : Session <
  SendValue < String, End > >
= PartialSession { executor = Box::new (
    async | _ctx, _sender | { } ) };
```

```
let p2 : Session <
  ReceiveValue < String, End > > = ...;
let (sender, _receiver) = channel();
(p2.executor)( (), sender ).await;
```

From the examples above we can see that direct access to the `executor` field is unsafe. The `PartialSession` is used with care within Ferrite to ensure that linearity is enforced in the implementation. Externally, the `run_session` is provided for executing Ferrite programs of type `Session<End>`, as only such programs can be executed safely without additional safe guard.

We end this section by showing the dynamic implementation of `send_value`, which implements the right rule for `SendValue`. The function accepts a value `x` of type `T` to be sent, and a continuation `cont` of type `PartialSession<C, A>`.

```
fn send_value < T, C: Context, A: Protocol >
  ( x: T, cont: PartialSession < C, A > )
  -> PartialSession < C, SendValue < T, A > >
{ PartialSession { executor = Box::new (
    async move | ctx, sender1 | {
      let (sender2, receiver2) = channel(1);
      sender1.send ( SendValue ( x, receiver2 ) ).await;
      (cont.executor)( ctx, sender2 ).await; }) } }
```

In the function body, the `PartialSession` constructor is called to create the return value of type `PartialSession< C, SendValue<T, A> >`. The `executor` field is given an async closure, which has the first argument `ctx` of type `C::Endpoints`, and the second argument `sender1` of type `SendValue<T, A>`. The closure body creates a new Rust channel pair `(sender2, receiver2)` with `A` as the payload type for the continuation. It then constructs the `SendValue<T, A>` payload using the value `x` and the continuation receiver `receiver2`. Finally, the `executor` of the continuation is called with the original linear context `ctx` untouched and the continuation sender `sender2`.

5 ADVANCED FEATURES

Sections 3 and 4 introduce the statics and dynamics of core Ferrite constructs. This section discusses Ferrite's support of recursive and shared session types.

5.1 Recursive Session Types

Many real world applications, such as web services, databases, instant messaging, and online games, are recursive in nature. As a result, it is essential for Ferrite to support recursive session types to allow the expression of the communication protocols of these applications. In this section, we report on Rust's limited support for recursive types and how Ferrite addresses the restriction and successfully encodes recursive session types.

Consider a simple example of a counter session type, which sends an infinite stream of integer values, incrementing each by one. To build a Ferrite program that offers such a session type, we may attempt to define the counter session type as follows:

```
type Counter = SendValue < u64, Counter >;
```

If we try to compile our program using the type definition above, we will get a compiler error that says "cycle detected when processing `Counter`". The problem with the above definition is that it defines a recursive type alias that directly refers back to itself, which is not supported in Rust. Rust imposes various restrictions on the forms of recursive types that can be defined to ensure that the space requirements of data is known at compile-time.

To circumvent this restriction, we could use recursive structs. However, if Ferrite relied on recursive struct definitions, end users would have to explicitly wrap each recursive session type in a recursive struct. Such an approach would be not be very convenient, so we want to find a way for the end users to define recursive session types using type aliases. One possibility is to perform recursion at the *type level*. [Crary et al. 1999]

Functional languages such as ML, OCaml, and Haskell provide excellent support for type-level recursion. We can use type-level recursion to define a recursive type `Fix`, which serves as the type-level fixed point of a type function. With that, we can first define a non-recursive data structure and then use `Fix` to get the fixed point of that data structure, which becomes a recursive data structure. Such non-recursive definitions of recursive data structures have many useful applications in the domain of recursion schemes [Meijer et al. 1991]. As for Ferrite, our focus is on using `Fix` to define recursive session types in a non-recursive way.

In Haskell, we can define such a `Fix` data type as follows:

```
data Fix (f :: Type -> Type) = Fix ( f ( Fix f ) )
```

It is parameterized by a *higher-kinded* type `f`, which has the kind `Type -> Type`. In the constructor definition, a value of *rolled* type `Fix f` can be constructed by providing a value of the *unrolled* type `f (Fix f)`. Unfortunately at the time of writing this article, higher-kinded types are not supported in Rust. As a result, we cannot define the `Fix` type in the same way as in Haskell. Fortunately, there is still a way to achieve the same outcome by using *defunctionalization* [Reynolds 1972; Yallop and White 2014] to emulate higher-kinded polymorphism in Rust. This can be done by defining a `TypeApp` trait as follows:

```
trait TypeApp < X > { type Applied; }
```

The `TypeApp` trait is parameterized by a type parameter `X`, which serves as the type argument to be applied to. This makes it possible for a Rust type `F` that implements `TypeApp` to act as if it has kind `Type -> Type` and be "applied" to `X`. The associated type `Applied` is then used as the result type of "applying" `X` to `F`. Using `TypeApp`, we can now define `Fix` in Rust as follows:

```
struct Fix < F: TypeApp < Fix < F > > > { unfix : Box < F::Applied > }
```

The new version of `Fix` is now parameterized over a type `F` that implements `TypeApp< Fix<F> >`. The body of `Fix` now contains `Box<F::Applied>`, with `F::Applied` representing the result of applying `Fix<F>` to `F`. This time the Rust compiler accepts the definition of `Fix`.

To use `Fix` for recursive session types, we want to implement `TypeApp` for *all* session types in Ferrite. We also need to pick a type for the *recursion point* of the fixed point type, and we chose `Z` for that purpose. The implementation of `TypeApp` for `Z` is shown on the left below. It simply replaces itself with the type argument `X` when applied. The implementation of `TypeApp` for `SendValue<T, A>` is shown on the right below. It delegates the type application to `A`, provided that the continuation session type `A` also implements `TypeApp` for the type argument `X`.

```
impl < X > TypeApp < X > for Z
{ type Applied = X; }
```

```
impl < X, T, A : TypeApp < X > >
  TypeApp < X > for SendValue < T, A >
{ type Applied = SendValue < T, A::Applied >; }
```

With `TypeApp` implemented, we can define the earlier recursive session type `Counter` as the type:

```
type Counter = Fix < SendValue < u64, Z > >;
```

To make `Counter` a valid session type, we must implement `Protocol` for `Fix` and `Z`. Moreover, since `Fix` is iso-recursive, Ferrite provides constructs for rolling and unrolling session types. Below is the function `fix_session`, which rolls up an offered unrolled session type into `Fix`:

```
fn fix_session
  < C: Context, F: Protocol + TypeApp < Fix < F > > >
  ( cont: PartialSession < C, F::Applied > )
  -> PartialSession < C, Fix < F > >
```

The function `fix_session` is used for rolling an offered session type `Fix<F>` from its unrolled version `F::Applied`. The continuation offering the unrolled type `F::Applied` is given to `fix_session`, and the Ferrite program returned offers the rolled up session type `Fix<F>`.

An example stream producer is shown below, which offers the recursive `Counter` session type that we defined realier.

```
fn stream_producer (count: u64) -> Session < Counter >
{ fix_session ( send_value_async ( async move || {
    task::sleep ( Duration::from_secs(1) ).await;
    ( count,
      stream_producer ( count + 1 ) ) }) ) }
```

We define `stream_producer` as a recursive function that recursively generates the Ferrite program of type `Session<Counter>` for each iteration. The recursive function is given a `count` argument for the integer to be sent. In the body, it uses `fix_session` to roll up the continuation `send_value_async(...)`, which offers the unrolled session type `SendValue<u64, Counter>`. The `send_value_async` function is an *asynchronous* version of `send_value`, by which the sent value can be constructed asynchronously. The asynchronous send allows the counter value to be produced *lazily*, that is only when the Ferrite program is ready to send, instead of eagerly producing all values before the Ferrite program can ever start. After that, the producer sleeps for one second to make the produced value observable and then sends the `count` argument by returning it to `send_value_async`. At the same time it recurs back to itself with `count+1` as an argument to produce the next count.

On the other side of `fix_session` and the `stream_producer` example, there is also a `unfix_session_for` construct for unrolling recursive session types in the linear context, as well as an example `stream_client` that consumes the integer stream from `stream_producer`. This is explained in detail in the extended example for recursive session types in Appendix B.3.

5.2 Shared Session Types

In the previous section we explored a recursive session type `Counter`, which is defined non-recursively using `Fix` and `Z`. Since `Counter` is defined as a linear session type, it cannot be shared among multiple clients. Shared communication, however, is essential for implementing many practical applications. For example, we may want to implement a simple counter web service using session types, to send a unique count for each request. To support such shared communication, we introduce *shared session types* in Ferrite, enabling safe shared communication among multiple clients.

Ferrite implements shared session types as introduced in Balzer and Pfenning [2017], which provide language primitives for the *acquiring* and *releasing* of shared processes and stratify session types into a linear and shared layer with two *modalities* connecting the layers. The SILL_R types extended with shared session types are as follows:

$$\begin{array}{lcl} S & \triangleq & \uparrow^S_L A \\ A, B & \triangleq & \downarrow^S_L S \mid \epsilon \mid \tau \triangleright A \mid \tau \triangleleft A \mid A \multimap B \mid A \otimes B \mid A \oplus B \mid A \,\&\, B \end{array}$$

The type system of SILL_R is extended to have one additional layer S for *shared session types*. At the shared layer, there is one connective $\uparrow^S_L A$, which represents a *linear to shared* modality for shifting a linear session type A up to the shared layer. This modality amounts to the acquiring of a shared process. A linear session type $\downarrow^S_L S$ is added to the linear layer, which represents a *shared to linear* modality for shifting a shared session type S down to the linear layer. This modality amounts to the releasing of an acquired process.

The usage of the two modalities can be demonstrated with a recursive definition of a shared counter example in SILL_R , as shown below. The code defines a shared session type SharedCounter, which is a shared version of the linear counter example. The linear portion of SharedCounter in between $\uparrow^S_L$ (acquire) and $\downarrow^S_L$ (release) resembles a critical section. The definition shows that SharedCounter is a shared session type that, when being *acquired*, offers a linear session type $\mathsf{Int} \triangleleft \downarrow^S_L \mathsf{SharedCounter}$ in the critical section. In other words, when acquired SharedCounter first sends an integer value and then continues as the linear session type $\downarrow^S_L \mathsf{SharedCounter}$. Once the linear critical section is *released*, the shared session type SharedCounter becomes available to other clients.

$$\mathsf{SharedCounter} = \uparrow^S_L \mathsf{Int} \triangleleft \downarrow^S_L \mathsf{SharedCounter}$$

5.2.1 Shared Session Types in Ferrite. Shared session types are recursive in nature, as they have to offer the same linear critical section to all clients that acquire a shared process. As a result, we can use the same technique that we use for defining recursive session types also for shared recursive session types. Below we show how the shared session type `SharedCounter` can be defined in Ferrite:

```
type SharedCounter = LinearToShared < SendValue < u64, Z > >;
```

Compared to linear recursive session types, the main difference is that instead of using `Fix`, a shared session type is defined using a new `LinearToShared` construct. This corresponds to the $\uparrow^S_L$ in SILL_R , with the inner type `SendValue<u64, Z>` corresponding to the linear portion of the shared session type. At the point of recursion, the type `Z` is used in place of $\downarrow^S_L \mathsf{SharedCounter}$, which is then unrolled during type application. The way of how the type unrolling works is shown below:

```
trait SharedTypeApp < X > { type Applied; }
struct SharedToLinear < F > { ... }
struct LinearToShared < F:
  SharedTypeApp < SharedToLinear<F> > > { ... }
```

```
trait SharedProtocol { ... }
impl < F > Protocol for SharedToLinear < F > ...
impl < F > SharedProtocol for LinearToShared < F > ...
```

The struct `LinearToShared` is parameterized by a linear session type `F` that implements `SharedTypeApp< SharedToLinear<F> >`. It uses the `SharedTypeApp` trait instead of the `TypeApp` trait to ensure that the session type is *strictly equi-synchronizing* [Balzer and Pfenning 2017; Balzer et al. 2019], requiring

an acquired session to be released to the same type at which it was previously acquired. Ferrite enforces this requirement by omitting an implementation of `SharedTypeApp` for `End`, ruling out invalid shared session types such as `LinearToShared< SendValue<u64, End> >`. We note that the type argument to `F`'s `SharedTypeApp` is another struct `SharedToLinear`, which corresponds to $\downarrow_{\mathsf{L}}^{\mathsf{S}}$ in $\mathsf{SILL}_{\mathsf{R}}$.

The struct `SharedToLinear` is also parameterized by `F`, but without the `TypeApp` constraint. Since `SharedToLinear` and `LinearToShared` are mutually recursive, the type parameter `F` alone is sufficient for reconstructing the type `LinearToShared<F>` from the type `SharedToLinear<F>`. The existing `Protocol` trait is implemented for linear session types in Ferrite, including `SharedToLinear`. A new trait `SharedProtocol` is also defined for identifying shared session types, i.e. `LinearToShared`.

Once a shared process is started, a shared channel is created to allow multiple clients to access the shared process through the shared channel. The code below shows the definition of the `SharedChannel` struct in Ferrite. Unlike linear channels, shared channels follow structural typing rules in the same way as functional variables, i.e., they can be weakened or contracted. This means that we can delegate the handling of shared channels to Rust as long as shared channels implement Rust's trait `Clone` to permit contraction.

```
struct SharedChannel < S: SharedProtocol > { ... }
impl < S: SharedProtocol > Clone for SharedChannel < S > { ... }
```

On the client side, a `SharedChannel` serves as an endpoint for interacting with a shared process running in parallel. To start the execution of such a shared process, a corresponding Ferrite program has to be defined and executed. Similar to `PartialSession`, we define `SharedSession` as shown below to represent such a shared Ferrite program.

```
struct SharedSession < S: SharedProtocol > { ... }
```

Just as `PartialSession` encodes linear Ferrite programs without executing them, `SharedSession` encodes shared Ferrite programs without executing them. Since `SharedSession` does not implement the `Clone` trait, the shared Ferrite program itself is affine and cannot be shared. To enable sharing, the shared Ferrite program must first be executed using the `run_shared_session` defined below. The function takes a shared Ferrite program of type `SharedSession<S>` and starts running it in the background as a shared process. Then in parallel, the shared channel of type `SharedChannel<S>` is returned to the caller, which can then be passed to multiple clients for accessing the shared process.

```
fn run_shared_session < S: SharedProtocol >
  ( session : SharedSession < S > ) -> SharedChannel < S >
```

Below is a high level overview of how a shared session type program is defined and used in Ferrite. The shared client `counter_client` is defined as a function that accepts a shared channel `counter` of type `SharedChannel<SharedCounter>` and constructs a Ferrite program of type `Session<End>` that makes use of the shared channel. We define the client as a function so that it can be called multiple times to construct multiple clients for demonstration purposes. Following that, a shared program `producer` of type `SharedSession<SharedCounter>` is defined. It is then passed to `run_shared_session` to start executing the producer in the background, and a shared channel `counter1` of type `SharedChannel<SharedCounter>` is returned. The shared channel is then cloned as `counter2`.

```
type SharedCounter = LinearToShared < SendValue < u64, Z > >;
fn counter_client ( counter: SharedChannel < SharedCounter > ) -> Session < End > { ... }
fn main () {
  let producer: SharedSession < SharedCounter > = ...;
  let counter1 = run_shared_session ( producer ); let counter2 = counter.clone();
  let child1 = task::spawn ( async move { ...; run_session( counter_client(counter1) ).await; });
  let child2 = task::spawn ( async move { ...; run_session( counter_client(counter2) ).await; });
  join!(child1, child2).await; }
```

The main application then executes two concurrent tasks, at some point in each of the child tasks, a counter client is constructed using `counter_client` and then executed using `run_session`. Finally the main function waits for the two child tasks to terminate using `join!`. A key observation is that multiple Ferrite programs that are executed independently are able to access the same shared producer through a reference to the shared channel.

5.2.2 Acquire and Release. Based on the work by Balzer and Pfenning [2017], Ferrite achieves safe communication for shared session types by treating the linear portion of the process as a critical section and enclosing it within acquire and release. The `SharedChannel` works as an alias to the shared process running in the background, and clients having a reference to the `SharedChannel` can *acquire* an exclusive linear channel to communicate with the shared process. During the lifetime of the linear channel, the shared process is locked and cannot be acquired by other clients. With the strictly equi-synchronizing constraint in place, the linear channel is eventually released through the `SharedToLinear` session type back to the same shared session type at which it was acquired. At this point one of the other clients that are waiting to acquire the shared process will get the next exclusive access, and the cycle repeats.

6 DISCUSSION

This section reflects on our choice of host language and the guarantees provided by Ferrite and concludes with directions for future work.

6.1 Choice of Host Language

We chose Rust as the host language for Ferrite, as we believe in Rust's potential for writing session type applications. Having the tagline "Fearless Concurrency" as one of its selling points, Rust has put much effort in its language design to ensure that concurrent programs can be written safely and efficiently. Rust has many unique features such as ownership, borrowing, and lifetimes, amounting to an affine type system, which make it well suited for high performance applications. For instance, Rust allows values to be safely modified without having to do expensive copying, under specific conditions enforced by the type system. Rust also provides a broad selection of high-level concurrency libraries, such as futures, channels, async/await, which Ferrite makes use of without having to implement the concurrency primitives from scratch.

Particularly conducive for the purpose of implementing an embedded DSL is Rust's support of functional programming constructs, such as parametric polymorphism, traits (type classes), and associated types (type families). Although Rust still lacks some more advanced language features, such as higher-kinded types and data kinds, we managed to find alternative ways, such as using traits as indirection to implement recursive session types in Ferrite. Rust moreover provides excellent support for type inference, making it possible for Ferrite programs to be written with minimal type annotations needed.

6.2 Guarantees

A natural question to ask is what the guarantees are that Ferrite provides. Since Ferrite is a direct embedding of SILL_R , it enjoys the properties of SILL_R , assuming the absence of implementation errors. SILL_R in turn comprises the language SILL_S [Balzer and Pfenning 2017], including value input and output from its precursor SILL [Toninho et al. 2013], extended with Rust statements. Both SILL_S and SILL are proved safe, ensuring protocol adherence in particular. As such these guarantees directly translate into SILL_R and Ferrite, given Ferrite's judgmental embedding of SILL_R . This means that as long as a programmer only uses the Ferrite library constructs made available, the session type programs are guaranteed to comply with their defined protocol. However, since

SILL_R includes arbitrary Rust statements and allows the input and output of arbitrary Rust values, a Ferrite session type program is subject to all the errors that a standard Rust program can suffer from. As a result, a Rust `panic!` may be the result of executing a Ferrite session type program.

The current implementation of Ferrite simply propagates panics to the main application. A future improvement would be to provide explicit handling of panics raised within a Ferrite program, so that end users are given the possibility to recover from runtime errors. For example, one option would be to catch panics from certain Ferrite sub-programs and convert the offered session type to an internal choice. Alternatively, we may consider extending Ferrite's session type language to add explicit language constructs for safe error handling in the spirit of Fowler et al. [2019].

An interesting endeavor beyond the scope of this work is the proving of properties of interest of combined Ferrite/Rust code. For example, even though the linear fragment of SILL_S is proved to be deadlock-free, deadlocks may still come about from embedded Rust code. In order to reason about deadlock-freedom of a combined Ferrite/Rust program, the semantics of SILL_R must be integrated with a semantics of Rust. The RustBelt [Jung et al. 2018] or Oxide [Weiss et al. 2019] verification efforts for Rust offer valuable starting points for such an endeavor.

6.3 Future Work

N-ary Choice. Although not mentioned in this article, the current implementation of Ferrite supports *binary* versions of both internal and external *choice*. There are some fine details of how Ferrite deals with the affinity constraints for branching operations and how we work around the lack of support for higher-rank types in Rust to encapsulate the continuation variants in different branches. The page limit unfortunately precludes a discussion thereof. Currently under development is the support of *n-ary* choice, of which we have a working prototype using *row polymorphism* and *prisms*. In profunctor optics, prisms are the duals of lenses and as such allow "injection of a session type into a sum" as opposed to its "projection from a product". Viewing the linear context as a product of session types and a choice as a sum of session types, the notions of lenses and prims provide powerful abstractions for context and choice manipulations.

Deep Embedding. Ferrite currently uses the `async-std` library for spawning Ferrite processes as lightweight async tasks, and uses async channels for communication. There exist many competing concurrency runtimes for Rust other than `async-std`, and it may be useful if end users could choose from different runtimes when running Ferrite programs. Some users may even want to opt out of asynchronous programming and instead use native threads for Ferrite processes.

Unfortunately, such flexibility is not possible with Ferrit's current implementation because Ferrite is a *shallowly embedded* DSL. This means that Ferrite programs cannot be interpreted in alternative ways, such as using a different concurrency library or channel implementation. We plan to explore a *deep embedding* of Ferrite to allow multiple interpretations of Ferrite programs. Such an approach has been explored by Lindley and Morris [2016] in the context of Haskell. A corresponding deep embedding in Rust, however, may face challenges due to limitations of Rust's type system mentioned in Section 6.1.

Nested Recursive Session Types. Section 5.1 discusses how the type `Z` is used as the type-level recursion point for a recursive session type definition inside `Fix`. Generalizing this idea further, it is possible to define *nested* recursive session types with one `Fix<...>` session type nested inside another `Fix`. The support of such nested recursive types is possible because we can implement `TypeApp` for `Fix` itself, essentially lifting `Fix` from `Type` to `Type -> Type`. A nested recursive session type would have multiple recursion points, and requires the usage of `S<Z>` etc. to fold and unfold from the outer `Fix`. We currently have a working prototype for nested recursive session types in Ferrite,

with some details left for improving its usability. We plan to further investigate possible use cases for nested recursive session types.

7 RELATED WORK

Static vs. Dynamic Guarantees. Session type programs require linear usage of channels to achieve session fidelity. Since the majority of host languages do not have a linear type system, the linear usage of channels has to be enforced in some other ways. Ferrite follows the approach by Pucella and Tov [2008], Imai et al. [2010], Padovani [2015], Lindley and Morris [2016], Imai et al. [2019], and statically type-checks session type programs, i.e., non-linear use of channels will raise a compile-time error. In contrast, Jespersen et al. [2015] and Kokke [2019] rely on Rust's affine type system to statically prevent session type channels to be used more than once, but rely on dynamic detection of channels that are closed without being used.

In addition to enforcing linear usage of channels, there are also other static guarantees of interest. In particular, the linear use of higher-order channels is enforced by Ferrite, Imai et al. [2010] and Imai et al. [2019]; while Padovani [2015] relies on dynamic checking to ensure that sent channels are used linearly. Ferrite also ensures statically that session type programs can only be executed when fully linked. Existing works such as Pucella and Tov [2008], Imai et al. [2010], and Imai et al. [2019], on the other hand, require manual linking of session type programs, making it possible for a provider to run without any client to communicate with running at the same time.

Intuitionistic vs. Classical Session Types. The works by Pucella and Tov [2008], Imai et al. [2010], Lindley and Morris [2016], and Imai et al. [2019] are all based on classical linear logic formulations of session types [Gay and Vasconcelos 2010; Wadler 2012]. As a result, users are required to *dualize* a session type for two session type programs to communicate. For example, a provider offering `ReceiveValue<String, End>` will require a client to offer the session type `SendValue<String, End>`. The need to keep track of two session types that are dual to each other can increase the cognitive load on programmers, especially when the session types become more complex. In particular when higher-order channels are involved, the complexity of dual session types increases, requiring additional constructs such as the use of *polarized* session types by Imai et al. [2019] to keep track of the polarity of sent/received channels.

In contrast, Ferrite is based on intuitionistic linear logic session types [Caires and Pfenning 2010], which avoid the need for dualization and instead provide right rules for providers offering a session type and left rules for clients consuming the session type. The polarity of session type channels in Ferrite are implicit, with channels in the linear context on the left side of a judgment having the client side polarity, and the offered session type on the right side of a judgment having the provider side polarity. When higher-order channels are involved, the sent and received channels always have the client side polarity.

Continuation Passing Style vs. Indexed Monad. The works by Pucella and Tov [2008], Imai et al. [2010] Lindley and Morris [2016], and Imai et al. [2019] use an *indexed (parameterized)* monad to track the linear usage of session type channels. An indexed monad is used to encode partial session type programs, which can be composed using the bind operator. The encoding requires to keep the result type as well as the pre- and post-conditions in the indexed monad. For example, a monadic encoding of Ferrite would be something like `PartialSession<C1, A1, C2, A2, X>`, where `C1` and `A1` are the initial linear context and session type, and `C2` and `A2` the final linear context and session type, and `X` the result type of the computation, respectively.

In contrast, Ferrite uses a continuation-passing-style (CPS) approach to pass the continuation as closure to term constructors. While it is true that Rust's support for monadic-style programming is limited, we chose a continuation-passing-style encoding because it stands in a one-to-one

correspondence with the sequent calculus-based typing rules in SILL_R. Compared to the type signature of indexed monads, the `PartialSession` struct we have in Ferrite only requires two type parameters, making it easier for end users to learn and understand the type signature. Should Rust provide better support for monads in the future, it will be straightforward to build an additional abstraction layer to translate between the current CPS-style and a new monad-style encoding, thus allowing end users to choose the approach that is best suited for them.

Shared Session Types. Ferrite is the first library that implements shared session types with acquire-release semantics [Balzer and Pfenning 2017]. This is in contrast to the copying semantics available through the linear exponential. Few of the related works address the need for shared session types. One reason being that for languages with structural rules such as Haskell and Ocaml, shared sessions based on a copy semantics can be trivially achieved by running different instances of the same session type program. Pucella and Tov [2008], Imai et al. [2010], and Imai et al. [2019] support some form of ad hoc sharing through non-linear channels that can be listened to by multiple providers and clients. This kind of sharing, however, is outside of the session type language and thus loses some of the safety guarantees provided by session types. In particular, they may be no provider listening on one side of the shared channel, causing clients on the other side to deadlock.

In contrast, Ferrite's shared channels provide the safety guarantee of session fidelity described by Balzer and Pfenning [2017], and allow true sharing of resources in a safe manner.

Linear Context. The approach of encoding the linear context as a type-level list is used by Pucella and Tov [2008], Imai et al. [2010], Lindley and Morris [2016], and Imai et al. [2019]. In their early works, Pucella and Tov [2008] provide operations to rearrange channels in the linear context in order to access a target channel as the first element. Imai et al. [2010] then introduce the use of context lenses for random access and update of channels in the linear context, and the same technique is later adopted by Imai et al. [2019]. Imai et al. [2010] also introduce the use of natural numbers to implement context lenses through type classes. Imai et al. [2019], in comparison, require a hand-written implementation of a context lens for each position in the linear context, and only provide context lenses for the first four slots by default.

Ferrite's design of the linear context and context lenses is close to the one by Imai et al. [2010], with some differences and improvements. Being based on intuitionistic linear logic session types, the linear context in Ferrite holds session type channels of client polarity. In contrast, the channels in the linear context of Imai et al. [2010] is of provider polarity, and channels in linear context of Imai et al. [2019] may be of either provider or client polarity. In the work by Imai et al. [2010], new context lenses are first generated together with an empty channel slot, and then the receive channel construct is used to bind the received channel to the empty channel slot. In contrast, Ferrite simplifies both operations into a single step done by `receive_channel`.

Higher-Order Channels. Ferrite is one of the few libraries that implements higher-order channels, also known as *session delegation*. Higher-order channels support the sending and receiving of channels along other channels. Jespersen et al. [2015] and Kokke [2019] support higher-order channels, but without enforcing the linear usage of the sent channels statically. Other than Ferrite, Imai et al. [2010] and Imai et al. [2019] also support higher-order channels with static linearity enforcement. For Imai et al. [2019], the sending of a channel needs to be accompanied with a polarity tag to identify whether the channel sent is of provider or client polarity.

Managed Concurrency. Many of the related works do not manage the concurrency aspects of spawning multiple session type processes and linking them for communication. Jespersen et al. [2015] and Kokke [2019] pass around channels as function arguments, and require the users to use external methods such as `fork` to run multiple processes in parallel. Pucella and Tov [2008] introduce

a `Rendezvous` type, which acts as a non-linear channel with two endpoints with session types that are dual to each other. The user then has to use `fork` to spawn a separate thread, and run either the provider or client by providing the respective endpoint of the channel. Since the `Rendezvous` type is non-linear, there is no guarantee that a provider must have exactly one counterpart client for communication. Imai et al. [2019] also provide a non-linear channel similar to `Rendezvous` for communication, with the two endpoints having the same session type but with opposite polarities.

In contrast, Ferrite manages the concurrency and communication on behalf of the end user, ensuring that a provider is always paired with exactly one client and that both proceses always start executing at the same time. Ferrite provides constructs such as `cut`, `include_session`, and `apply_channel` for linking multiple Ferrite programs. Imai et al. [2010] offer a similar way of linking multiple session type programs. Users need to first use `new` to allocate an empty slot in the linear context, and then use `fork` on a provider. The continuation then has the dual session type of the provider in the post type of the indexed monad, so that the program can continue as the client.

REFERENCES

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP (2017), 37:1–37:29.

Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *29th International Conference on Concurrency Theory (CONCUR) (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 30:1–30:18.

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *28th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Vol. 11423. Springer, 611–639.

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *21st International Conference on Concurrency Theory (CONCUR)*. Springer, 222–236.

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 50–63.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17. https://doi.org/10.1145/1232420.1232424

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20, 1 (2010), 19–50. https://doi.org/10.1017/S0956796809990268

Andrew Gerrand. 2010. The Go Blog: Share Memory By Communicating. https://blog.golang.org/share-memory-by-communicating

Kohei Honda. 1993. Types for Dyadic Interaction. In *4th International Conference on Concurrency Theory (CONCUR)*. Springer, 509–523.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *7th European Symposium on Programming (ESOP)*. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 273–284.

Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. 2010. Type-Safe Eventful Sessions in Java. In *24th European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 6183. Springer, 329–353. https://doi.org/10.1007/978-3-642-14107-2_16

Raymond Hu, Nobuko Yoshida, and Kohei Honda. 2008. Session-Based Distributed Programming in Java. In *22nd European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science)*, Vol. 5142. Springer, 516–541. https://doi.org/10.1007/978-3-540-70592-5_22

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-ocaml: a Session-based Library with Polarities and Lenses. *Science of Computer Programming* 172 (2019), 135–159. https://doi.org/10.1016/j.scico.2018.08.005

Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. 2010. Session Type Inference in Haskell. In *3rd Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES) 2010, Paphos, Cyprus, 21st March 201*

(EPTCS), Vol. 69. 74–91. https://doi.org/10.4204/EPTCS.69.6

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *11th ACM SIGPLAN Workshop on Generic Programming (WGP)*. https://doi.org/10.1145/2808098.2808100

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Steve Klabnik and Carol Nichols. 2018. The Rust Programming Language. https://doc.rust-lang.org/stable/book/

Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *12th Interaction and Concurrency Experience, ICE 2019*. 48–60.

Sam Lindley and J. Garrett Morris. 2016. Embedding Session Types in Haskell. In *9th International Symposium on Haskell*. ACM, 133–145. https://doi.org/10.1145/2976002.2976018

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. 124–144. https://doi.org/10.1007/3540543961_7

Luca Padovani. 2015. A Simple Library Implementation of Binary Sessions. (Oct. 2015). https://hal.archives-ouvertes.fr/hal-01216310 working paper or preprint.

Luca Padovani. 2017. A Simple Library Implementation of Binary Sessions. *Journal of Functional Programming* 27 (2017), e4. https://doi.org/10.1017/S0956796816000289

Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor Optics: Modular Data Accessors. *Programming Journal* 1, 2 (2017), 7. https://doi.org/10.22152/programming-journal.org/2017/1/7

Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. In *1st ACM SIGPLAN Symposium on Haskell*. ACM, 25–36. https://doi.org/10.1145/1411286.1411290

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *ACM Annual Conference*, Vol. 2. ACM, 717–740. https://doi.org/10.1145/800194.805852

Matthew Sackman and Susan Eisenbach. 2008. *Session Types in Haskell: Updating Message Passing for the 21st Century*. Technical Report. Imperial College. http://hdl.handle.net/10044/1/5918

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *30th European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 56. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 21:1–21:28.

Servo. 2020. Servo source code - canvas paint thread. https://github.com/servo/servo/blob/c67b3d71e23f10ba2049bdd9aed822b19ed8527f/components/canvas/canvas_paint_thread.rs

Bernardo Toninho. 2015. *A Logical Foundation for Session-based Concurrent Computation*. Ph.D. Dissertation. Carnegie Mellon University and New University of Lisbon.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: a Monadic Integration. In *22nd European Symposium on Programming (ESOP)*. Springer, 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

Philip Wadler. 2012. Propositions as Sessions. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 273–286.

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982

Jeremy Yallop and Leo White. 2014. Lightweight Higher-Kinded Polymorphism. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. 119–135. https://doi.org/10.1007/978-3-319-07151-0_8

Table 3. Overview of session terms in SILL_R and Ferrite.

SILL_R	Ferrite	Term	Description
ϵ	`End`	terminate;	Terminate session.
		wait a; K	Wait for channel a to close.
$\tau \triangleright A$	`ReceiveValue<T, A>`	$x \leftarrow$ receive_value; K	Receive value x of type τ.
		send_value_to a x; K	Send value x of type τ to channel a.
$\tau \triangleleft A$; K	`SendValue<T, A>`	send_value x	Send value of type τ.
		$x \leftarrow$ receive_value_from a x; K	Receive value of type τ from channel a.
$A \multimap B$	`ReceiveChannel<A, B>`	$a \leftarrow$ receive_channel; K	Receive channel a of session type A.
		send_channel_to f a; K	Send channel a to channel f of session type $A \multimap B$.
$A \otimes B$	`SendChannel<A, B>`	send_channel_from a; K	Send channel a of session type A.
		$a \leftarrow$ receive_channel_from f a; K	Receive channel a from channel f of session type $A \otimes B$.
$A \,\&\, B$	`ExternalChoice<A, B>`	offer_choice K_l K_r	Offer either continuation K_l or K_r based on client's choice.
		choose_left a; K	Choose the left branch offered by channel a
		choose_right a; K	Choose the right branch offered by channel a
$A \oplus B$	`InternalChoice<A, B>`	offer_left; K	Offer the left branch
		offer_right; K	Offer the right branch
		case a K_l K_r	Branch to either K_l or K_r based on choice offered by channel a.
$\uparrow^S_L A$	`LinearToShared<A>`	accept_shared_session; K_l	Accept an acquire, then continue as linear session K_l.
		$a \leftarrow$ acquire_shared_session s; K_l	Acquire shared channel s as linear channel a.
$\downarrow^S_L S$	`SharedToLinear<S>`	detach_shared_session; K_s	Detach linear session and continue as shared session K_s.
		release_shared_session a; K_l	Release acquired linear session.
-	`Fix<F>`	`fix_session(cont)`	Roll up session type `F::Applied` offered by `cont`.
		`unfix_session_for(a, cont)`	Unroll channel `a` to session type `F::Applied` in `cont`.

A TYPING RULES

A.1 Typing Rules for SILL_R

Following is a list of inference rules in SILL_R .

Communication

$$\frac{\Gamma\,;\,\Delta_1 \vdash a :: A \qquad \Gamma\,;\,\Delta_2, a' : A \vdash b :: B}{\Gamma\,;\,\Delta_1, \Delta_2 \vdash a' \leftarrow \text{cut}\ a\,;\ b :: B}\ (\text{T-CUT})$$

$$\frac{\Gamma\,;\,\cdot \vdash a :: A \qquad \Gamma\,;\,\Delta, a' : A \vdash b :: B}{\Gamma\,;\,\Delta \vdash a' \leftarrow \text{include}\ a\,;\ b :: B}\ (\text{T-INCL})$$

$$\frac{\Gamma\,;\,\cdot \vdash f :: A \multimap B \qquad \Gamma\,;\,\cdot \vdash a :: A}{\Gamma\,;\,\cdot \vdash \text{apply\_channel}\ f\ a :: B}\ (\text{T-APP})$$

$$\frac{}{\Gamma\,;\,a : A \vdash \text{forward}\ a :: A}\ (\text{T-FWD})$$

Termination

$$\frac{}{\Gamma\,;\,\cdot \vdash \text{terminate};: \epsilon}\ (\text{T1}_\text{R})$$

$$\frac{\Gamma\,;\,\Delta \vdash K :: A}{\Gamma\,;\,\Delta,\ a : \epsilon \vdash \text{wait}\ a;\ K :: A}\ (\text{T1}_\text{L})$$

Receive Value

$$\frac{\Gamma,\ x : \tau\,;\Delta \vdash K :: A}{\Gamma\,;\Delta \vdash x \leftarrow \text{receive\_value};\ K :: \tau \triangleright A}\ (\text{T}\triangleright_{\text{R}}) \qquad \frac{\Gamma\,;\,\Delta, a : A \vdash K :: B}{\Gamma,\ x : \tau;\ \Delta,\ a : \tau \triangleright A \vdash \text{send\_value\_to}\ a\ x;\ K :: B}\ (\text{T}\triangleright_{\text{L}})$$

Send Value

$$\frac{\Gamma\,;\,\Delta \vdash K :: A}{\Gamma,\ x : \tau;\ \Delta \vdash \text{send\_value}\ x;\ K :: \tau \triangleleft A}\ (\text{T}\triangleleft_{\text{R}}) \qquad \frac{\Gamma,\ a : \tau\,;\Delta, a : A \vdash K :: A}{\Gamma\,;\Delta, a : \tau \triangleright A \vdash x \leftarrow \text{receive\_value\_from}\ a;\ K :: B}\ (\text{T}\triangleleft_{\text{L}})$$

Receive Channel

$$\frac{\Gamma\,;\,\Delta, a : A \vdash K :: B}{\Gamma\,;\,\Delta \vdash a \leftarrow \text{receive\_channel};\ K :: A \multimap B}\ (\text{T}\multimap_{\text{R}}) \qquad \frac{\Gamma\,;\,\Delta, f : A_2 \vdash K :: B}{\Gamma\,;\,\Delta,\ f : A_1 \multimap A_2,\ a : A_1 \vdash \text{send\_channel\_to}\ f\ a;\ K :: B}\ (\text{T}\multimap_{\text{L}})$$

Send Channel

$$\frac{\Gamma\,;\,\Delta \vdash K :: B}{\Gamma\,;\,\Delta, a : A \vdash \text{send\_channel\_from}\ a;\ K :: A \otimes B}\ (\text{T}\otimes_{\text{R}}) \qquad \frac{\Gamma\,;\,\Delta, f : A_2,\ a : A_1 \vdash K :: B}{\Gamma\,;\,\Delta,\ f : A_1 \otimes A_2 \vdash a \leftarrow \text{receive\_channel\_from}\ f;\ K :: B}\ (\text{T}\otimes_{\text{L}})$$

External Choice

$$\frac{\Gamma\,;\,\Delta \vdash K_l :: A \qquad \Gamma\,;\,\Delta \vdash K_r :: B}{\Gamma\,;\,\Delta \vdash \text{offer\_choice}\ K_l\ K_r :: A \,\&\, B}\ (\text{T}\,\&_{\text{R}})$$

$$\frac{\Gamma\,;\,\Delta, a : A_1 \vdash K :: B}{\Gamma\,;\,\Delta,\ a : A_1 \,\&\, A_2 \vdash \text{choose\_left}\ a;\ K :: B}\ (\text{T}\,\&_{\text{L}})$$

$$\frac{\Gamma\,;\,\Delta, a : A_2 \vdash K :: B}{\Gamma\,;\,\Delta,\ a : A_2 \,\&\, A_2 \vdash \text{choose\_right}\ a;\ K :: B}\ (\text{T}\,\&_{\text{L}})$$

Internal Choice

$$\frac{\Gamma\,;\,\Delta \vdash K :: A}{\Gamma\,;\,\Delta \vdash \text{offer\_left};\ K :: A \oplus B}\ (\text{T}\oplus_{\text{R}})$$

$$\frac{\Gamma\,;\,\Delta \vdash K :: B}{\Gamma\,;\,\Delta \vdash \text{offer\_right};\ K :: A \oplus B}\ (\text{T}\oplus_{\text{R}})$$

$$\frac{\Gamma\,;\,\Delta, a : A_1 \vdash K_l :: B \qquad \Gamma\,;\,\Delta, a : A_2 \vdash K_r :: B}{\Gamma\,;\,\Delta,\ a : A_1 \oplus A_2 \vdash \text{case}\ a\ K_l\ K_r :: B}\ (\text{T}\,\&_{\text{L}})$$

Shared Session Types

$$\frac{\Gamma\,;\,\cdot \vdash K_l :: A}{\Gamma\,;\,\cdot \vdash \text{accept\_shared\_session};\ K_l :: \uparrow^{\text{S}}_{\text{L}} A}\ (\text{T}\uparrow^{\text{S}}_{\text{LR}}) \qquad \frac{\Gamma\,;\,\Delta,\ a : A \vdash K :: B}{\Gamma, s : \uparrow^{\text{S}}_{\text{L}} A\,;\,\Delta \vdash a \leftarrow \text{acquire\_shared\_session}\ s;\ K :: B}\ (\text{T}\uparrow^{\text{S}}_{\text{LL}})$$

$$\frac{\Gamma\,;\,\cdot \vdash K_s :: S}{\Gamma\,;\,\cdot \vdash \text{detach\_shared\_session};\ K_s :: \downarrow^{\text{S}}_{\text{L}} S}\ (\text{T}\downarrow^{\text{S}}_{\text{LR}}) \qquad \frac{\Gamma, s : S\,;\,\Delta \vdash K :: B}{\Gamma\,;\,\Delta,\ a : \downarrow^{\text{S}}_{\text{L}} S \vdash s \leftarrow \text{release\_shared\_session}\ a;\ K :: B}\ (\text{T}\downarrow^{\text{S}}_{\text{LL}})$$

A.2 Typing Constructs for Ferrite

Following is a list of function signatures of the term constructors provided in Ferrite.

Forward

```
fn forward < N, C, A > ( n : N ) -> PartialSession < C, A >
where A : Protocol, C : Context, N::Target : EmptyContext, N : ContextLens < C, A, Empty >
```

Termination

```
fn terminate < C: EmptyContext >
  () -> PartialSession < C, End >
```

```
fn wait < C: Context, A: Protocol,
  N: ContextLens < C, End, Empty > >
  ( n: N, cont: PartialSession < N::Target, A > )
  -> PartialSession < C, A >
```

Communication

```
fn cut < C1, C2, C3, C4, A, B >
  ( cont1 : PartialSession < C3, B >,
    cont2 : PartialSession < C2, A > )
  -> PartialSession < C4, B >
where A : Protocol, B : Protocol,
  C1 : Context, C2 : Context,
  C3 : Context, C4 : Context,
  C1 : AppendContext < ( A, () ), Appended = C3 >,
  C1 : AppendContext < C2, Appended = C4 >,
```

Receive Value

```
fn receive_value < T, C, A, Fut >
  ( cont : impl FnOnce (T) -> Fut  + Send + 'static )
  -> PartialSession <
    C, ReceiveValue < T, A > >
```

Send Value

```
fn send_value < T, C, A >
  ( val : T, cont : PartialSession < C, A > )
  -> PartialSession < C, SendValue < T, A > >
where T: Send + 'static, A: Protocol, C: Context

fn send_value_async < T, C, A, Fut >
  ( cont_builder: impl
      FnOnce () -> Fut + Send + 'static )
  -> PartialSession < C, SendValue < T, A > >
where T: Send + 'static, A: Protocol, C: Context,
  Fut : Future < Output =
    ( T,  PartialSession < C, A > ) > + Send
```

Receive Channel

```
fn receive_channel < C, A, B >
  ( cont : impl FnOnce ( C::Length ) ->
      PartialSession < C::Appended, B > )
  -> PartialSession < C, ReceiveChannel < A, B > >
where A : Protocol, B : Protocol,
  C : AppendContext < ( A, () ) >
```

Send Channel

```
fn send_channel_from < N, C, A, B >
  ( n : N,
    cont: PartialSession < N::Target, B > )
  -> PartialSession < C, SendChannel < A, B > >
where C : Context, A : Protocol, B : Protocol,
  N : ContextLens < C, A, Empty >
```

```
fn include_session < C, A, B >
  ( session : Session < A >,
    cont : impl FnOnce ( C :: Length )
      -> PartialSession < C::Appended, B > )
  -> PartialSession < C, B >
where A : Protocol, B : Protocol, C : Context,
  C : AppendContext < ( A, () ) >

fn apply_channel < A, B >
  ( f : Session < ReceiveChannel < A, B > >,
    a : Session < A > )
  -> Session < B >
where A : Protocol, B : Protocol
```

```
fn send_value_to < N, C, T, A, B >
  ( lens : N, value : T,
    cont : PartialSession < N::Target, A > )
  -> PartialSession < C, A >
where A : Protocol, B : Protocol, C : Context,
  T : Send + 'static,
  N : ContextLens <
    C, ReceiveValue < T, B >, B >
```

```
fn receive_value_from < N, C, T, A, B, Fut >
  ( n : N,
    cont : impl FnOnce (T) -> Fut + Send + 'static )
  -> PartialSession < C, B >
where A : Protocol, B : Protocol, C : Context,
  T : Send + 'static,
  Fut : Future < Output =
    PartialSession < N :: Target, B > > + Send,
  N : ContextLens < C, SendValue < T, A >, A >
```

```
fn send_channel_to < N1, N2, C, A1, A2, B >
  ( n1 : N1, n1 : N2,
    cont : PartialSession < N1::Target, B > )
  -> PartialSession < C, B >
where C : Context, B : Protocol,
  A1 : Protocol, A2 : Protocol,
  N1 : ContextLens < N2::Target,
    ReceiveChannel < A1, A2 >, A2 >,
  N2 : ContextLens < C, A1, Empty >
```

```
fn receive_channel_from < C1, C2, A1, A2, B, N >
  ( n : N,
    cont: impl FnOnce ( C2::Length )
      -> PartialSession < C2::Appended, B > )
  -> PartialSession < C1, B >
where A1 : Protocol, A2 : Protocol,
  B : Protocol, C1 : Context,
  C2 : AppendContext < ( A1, () ) >,
  N : ContextLens < C1,
    SendChannel < A1, A2 >, A2, Target = C2 >
```

Recursive Session Types

```
fn fix_session < F, A, C >
  ( cont: PartialSession < C, A > )
  -> PartialSession < C, Fix < F > >
where C : Context, F : Protocol, A : Protocol,
  F : TypeApp < Fix < F >, Applied = A >
```

```
fn unfix_session_for < N, C, A, B, F >
  ( n : N,
    cont : PartialSession < N::Target, B > )
  -> PartialSession < C, B >
where A : Protocol, B : Protocol, C : Context,
  F : Protocol + TypeApp < Fix < F >, Applied = A >,
  N : ContextLens < C, Fix < F >, A, >
```

Shared Session Types

```
fn accept_shared_session < F >
  ( cont : PartialSession <
      ( Lock < F >, () ), F::Applied > )
  -> SharedSession < LinearToShared < F > >
where F::Applied : Protocol,
  F : Protocol + SharedTypeApp < SharedToLinear < F > >

fn detach_shared_session < F, C >
  ( cont : SharedSession < LinearToShared < F > > )
  -> PartialSession <
      ( Lock < F >, C ), SharedToLinear < F > >
where C : EmptyContext, F::Applied : Protocol,
  F : Protocol + SharedTypeApp < SharedToLinear < F > >
```

```
fn acquire_shared_session < F, C, A, Fut >
  ( shared : SharedChannel < LinearToShared < F > >,
    cont : impl FnOnce ( C :: Length ) -> Fut
      + Send + 'static )
  -> PartialSession < C, A >
where C : Context, A : Protocol, F::Applied : Protocol,
  F : Protocol + SharedTypeApp < SharedToLinear < F > >,

  C : AppendContext < ( F::Applied , () ) >,
  Fut : Future < Output =
    PartialSession < C::Appended, A > > + Send,

fn release_shared_session < N, C, F, A >
  ( n : N,
    cont : PartialSession < N::Target, A > )
  -> PartialSession < C, A >
where A : Protocol, C : Context,
  F : Protocol + SharedTypeApp < SharedToLinear < F > >,

  N : ContextLens < C, SharedToLinear < F >, Empty >
```

External Choice

```
type InjectCont < C, A, B > =
  Either <
    Box < dyn FnOnce (
       PartialSession < C, A >
    ) -> ContSum < C, A, B > + Send >,
    Box< dyn FnOnce (
       PartialSession < C, B >
    ) -> ContSum < C, A, B > + Send > >;

struct ContSum < C, A, B >
where C: Context, A: Protocol, B: Protocol
{ result: Either <
    PartialSession < C, A >,
    PartialSession < C, B > > }

fn offer_choice < C, A, B >
  ( cont_builder : impl FnOnce
      ( InjectCont < C, A, B > )
      -> ContSum < C, A, B > + Send + 'static)
  -> PartialSession < C, ExternalChoice < A, B > >
where A : Protocol, B : Protocol, C : Context
```

```
fn choose_left < N, C, A1, A2, B >
  ( n: N,
    cont: PartialSession < N::Target, B > )
  -> PartialSession < C, B >
where C: Context, A1: Protocol,
  A2: Protocol, B: Protocol,
  N: ContextLens < C, ExternalChoice<A1, A2>, A1 >

fn choose_right < N, C, A1, A2, B >
  ( n: N,
    cont: PartialSession < N::Target, B > )
  -> PartialSession < C, B >
where C: Context, A1: Protocol,
  A2: Protocol, B: Protocol,
  N: ContextLens < C, ExternalChoice<A1, A2>, A2 >
```

Internal Choice

```
struct ContSum < C1, C2, A >
where C1 : Context, C2 : Context, A : Protocol
{ result: Either <
    PartialSession < C1, A >,
    PartialSession < C2, A >
  > }

fn offer_left < C, A, B >
  ( cont:  PartialSession < C, A > )
  -> PartialSession < C, InternalChoice < A, B > >
where A : Protocol, B : Protocol, C : Context

fn offer_right < C, A, B >
  ( cont:  PartialSession < C, B > )
  -> PartialSession < C, InternalChoice < A, B > >
where A : Protocol, B : Protocol, C : Context
```

```
type InjectCont < C2, C3, B > =
  Either <
    Box < dyn FnOnce ( PartialSession < C2, B > )
      -> ContSum < C2, C3, B > + Send >,
    Box < dyn FnOnce ( PartialSession < C3, B > )
      -> ContSum < C2, C3, B > + Send > >;

fn case < N, C1, C2, C3, C4, A1, A2, B >
  ( n : N,
    cont : impl FnOnce ( InjectCont < C2, C3, B > )
      -> ContSum < C2, C3, B > + Send + 'static )
  -> PartialSession < C1, B >
where
  C1 : Context, C2 : Context, C3 : Context, C4 :
     Context,
  A1 : Protocol, A2 : Protocol, B : Protocol,
  N : ContextLens < C1, InternalChoice < A1, A2 >,
    A1, Target = C2, Deleted = C4 >,
  N : ContextLens < C1, InternalChoice < A1, A2 >,
    A2, Target = C3, Deleted = C4 >
```

B EXTENDED EXAMPLES

B.1 Derivation Tree for Hello World Example

We can show how that the function `receive_value` embeds the inference rule ($\mathsf{T} \triangleright_{\mathsf{R}}$), by visualizing it as extending Rust's type system with an additional inference rule shown below. The judgments are given a Rust context Ψ, with variables in subject to Rust typing rules. From the inference rule we can see that it shares the the essence of ($\mathsf{T} \triangleright_{\mathsf{R}}$), with the same linear context `C` in both the premise and conclusion, and the offered session type changing from `A` in the premise to `ReceiveValue<T, A>` in the conclusion.

$$\frac{\Psi \vdash \texttt{cont : impl FnOnce (T) -> PartialSession < C, A >}}{\Psi \vdash \texttt{receive\_value(cont) : PartialSession < C, ReceiveValue < T, A > >}}$$

Similarly, we can think of the function `send_value_to` as introducing the inference rule shown below to Rust's type system. The notation `L` $\Rightarrow$ `ContextLens<...>` is used to denote that the type `L` implements a specific `ContextLens` instance in Rust. Notice that the type of `l` is not changed in the continuation, but the type `L` may implement a more than one instances of `ContextLens` to be used in the continuation.

$$\frac{\Psi, \texttt{l: L} \vdash \texttt{cont : PartialSession < L::Target, B >} \qquad \texttt{L} \Rightarrow \texttt{ContextLens < C, ReceiveValue <T, A>, A >}}{\Psi, \texttt{l: L, x: T} \vdash \texttt{send\_value\_to(l, x, cont) : PartialSession < C, B >}}$$

We can visualize the function `receive_channel` as an inference rule added to Rust as shown below. The rule follows the same structure as ($\mathsf{T}_{\Sigma} \multimap_{\mathsf{R}}$), with the constraint `C: Context` omitted for brievity.

$$\frac{\Psi \vdash \texttt{cont : impl FnOnce (C::Length) -> PartialSession< C::Appended, B >} \qquad \texttt{C1} \Rightarrow \texttt{AppendContext <(A, ())>}}{\Psi \vdash \texttt{receive\_channel(cont) : PartialSession< C, ReceiveChannel<A, B> >}}$$

By visualizing Ferrite term constructors as inference rules, we can better understand the hello world example in Section 3 by building derivation trees for the program. The code for `hello_provider` and `hello_client` are repeated below:

```
let hello_provider :
  Session < ReceiveValue < String, End > >
= receive_value ( | name | {
    println!("Hello, {}", name);
    terminate () });
```

```
let hello_client : Session <
  ReceiveChannel <
    ReceiveValue < String, End >, End > >
= receive_channel ( | a | {
    send_value_to ( a, "Alice".to_string(),
      wait ( a, terminate() ) ) });
```

We can build the derivation tree of `hello_provider` as follows:

$$\dfrac{\dfrac{}{\Psi, \texttt{name : String} \vdash \texttt{println!("Hello, \{\}!", name); terminate() : PartialSession < (), End >}}}{\dfrac{\Psi \vdash \texttt{|name| \{ println!("Hello, \{\}!", name); terminate() \} :}}{\Psi \vdash \texttt{receive\_value(|name|\{ println!("Hello, \{\}!", name); terminate() \}) :}}\ \texttt{impl FnOnce(String) -> PartialSession < (), End >}}\ \texttt{PartialSession < (), ReceiveValue<String, End> >}$$

The first part of derivation tree of `hello_client`, of how a received channel is bound to the linear context, is shown as follows:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\Psi, \texttt{a: Z} \vdash \texttt{send\_value\_to(a, ...) : PartialSession < (ReceiveValue<String, End>, ()), End >}}}{\Psi \vdash \texttt{|a|\{ send\_value\_to(a, ...) \} : impl FnOnce(Z) -> PartialSession < (ReceiveValue<String, End>, ()), End >}} \quad \dfrac{}{\texttt{() ⇒ AppendContext < (ReceiveValue<String, End>, ()), Appended= (ReceiveValue<String, End>,()) >}} \quad \dfrac{}{\texttt{() ⇒ Context < Length=Z >}}}{\Psi \vdash \texttt{receive\_channel(|a|\{ send\_value\_to(...) \}) : PartialSession < (), ReceiveChannel< ReceiveValue<String, End>, End > > >}}$$

With the received channel and context lens in the environment, The second part of derivation tree of `hello_client`, $\mathcal{D}$, is continued as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{(Empty, ()) ⇒ EmptyContext}}}{\Psi \vdash \texttt{terminate() : PartialSession < (Empty,()), End >}} \quad \dfrac{}{\texttt{Z ⇒ ContextLens < (End,()), End, Empty, Target=(Empty,()) >}}}{\Psi, \texttt{a: Z} \vdash \texttt{wait(a, terminate()) : PartialSession <(End, ()), End >}} \quad \dfrac{}{\texttt{Z ⇒ ContextLens < (ReceiveValue<String, End>,()), ReceiveValue<String, End>, End, Target=(End,()) >}}}{\Psi, \texttt{a: Z} \vdash \texttt{send\_value\_to(a, "Alice".to\_string(), wait(a, ...)) : PartialSession < (ReceiveValue<String, End>, ()), End >}}$$

B.2 Communication

B.2.1 Include Session. Other than the general `cut` rule, Ferrite also provides a more restricted version of `cut` called `include`:

$$\frac{\Gamma\,;\,\cdot \vdash a :: A \quad \Gamma\,;\,\Delta, x : A \vdash b :: B}{\Gamma\,;\,\Delta \vdash x \leftarrow \mathsf{include}\, a\,;\, b :: B}\ (\textsc{T-incl})$$

The typing rule T-INCL is nearly identical to T-CUT, with the additional restriction that the linear context for a must be empty. This restriction makes it much easier for the Rust compiler to infer the type for Δ, since there is no longer a need to split it into two parts for the two continuations. Without the linear context splitted, the existing context lenses can also be preserved, making them usable before and after an `include`. It is also clear that T-INCL is *derivable* from T-CUT, by simply unifiying Δ_1 in T-CUT with $\cdot$. As a result, introducing T-INCL does not affect the properties of session types. T-INCL is implemented in Ferrite as the `include_session` function as follow:

```
fn include_session < A: Protocol, B: Protocol, C: Context >
  ( a: Session < A >,
    cont: impl FnOnce ( C::Length )
      -> PartialSession < C::Appended, B >
  ) -> PartialSession < C, B >
where C : AppendContext < ( A, () ) >,
```

`include_session` takes a Ferrite program of type `Session<A>` and appends the offered channel to the linear context `C`. Similar to `receive_channel`, `include_session` generates a context lens of type `C::Length`, which is used by the continuation closure `cont` to access the appended channel `A` to `C`. `cont` returns a `PartialSession < C::Appended, B >`, indicating that it offers session type `B` using the linear context `C` appended with `A`. Finally `include_session` returns `PartialSession < C, B >`, which works on the original linear context `C` and offers session type `B`.

The `apply_channel` construct is implemented using `include_session`, `send_channel_to`, and `forward` as shown below.

```
fn apply_channel < A: Protocol, B: Protocol >
  ( f: Session < ReceiveChannel<A, B> >, a: Session < A > )
  -> Session < B >
{ include_session ( f, | chan_f | {
    include_session ( a, | chan_a | {
      send_channel_to ( chan_f, chan_a,
        forward ( chan_f ) ) }) }) }
```

We can prove that the typing rule T-APP is derivable from T-CUT, as shown below.

$$\dfrac{\dfrac{\dfrac{}{\Gamma\,;\,f' : B \vdash \mathsf{forward}\, f' :: B}}{\Gamma\,;\,f' : A \multimap B,\, a : A \vdash \mathsf{send\_channel\_to}\, f'\, a';\, \mathsf{forward}\, f' :: B} \quad \dfrac{\dots}{\Gamma\,;\,\cdot \vdash a :: A}}{\dfrac{\Gamma\,;\,f' : A \multimap B \vdash a' \leftarrow \mathsf{cut}\, a\,;\, \mathsf{send\_channel\_to}\, f'\, a';\, \mathsf{forward}\, f' :: B \qquad \dfrac{\dots}{\Gamma\,;\,\cdot \vdash f :: A \multimap B}}{\Gamma\,;\,\cdot \vdash f' \leftarrow \mathsf{cut}\, f\,;\, a' \leftarrow \mathsf{cut}\, a\,;\, \mathsf{send\_channel\_to}\, f'\, a';\, \mathsf{forward}\, f';:: B}}{\Gamma\,;\,\cdot \vdash \mathsf{apply\_channel}\, f\, a :: B}$$

B.3 Recursive Session Types

B.3.1 Unrolling Session Types. The function `unfix_session_for` shown below works with a context lens `N` operating on a linear context `C` to unroll a recursive session type `Fix<F>`. The inner protocol `F` implements `TypeApp< Fix <F> >`. The target linear context `N::Target` is the result of replacing `Fix<F>` in `C` with `F::Applied`, which is then given as the linear context for the continuation session `cont`.

```
fn unfix_session_for
  < N, C: Context, A: Protocol, F: Protocol >
  ( n : N, cont : PartialSession < N::Target, A > )
  -> PartialSession < C, A >
where
  F : TypeApp < Fix < F > >,
  N : ContextLens < C, Fix < F >, F::Applied, >
```

Using `unfix_session_for`, we can define a client `stream_client` to consume the `Counter` channel offered by `stream_producer` as follows:

```
1  fn stream_client () -> Session < ReceiveChannel < Counter, End > >
2  { receive_channel ( | stream | {
3      unfix_session_for ( stream,
4        receive_value_from ( stream,
5          async move | count | {
6            println!("Received value: {}", count);
7            include_session ( stream_client (),
8              | next | {
9                send_channel_to ( next, stream,
10                 forward ( next ) ) }) }) ) }) }
```

The function `stream_client` is a bit more involved than `stream_server`, so we will go through the steps one line at a time. In the first line, we define `stream_client` to be a recursive function with no argument that returns a `Session< ReceiveChannel<Counter, End> >`. In other words `stream_client` receives a channel of type `Counter` and then terminates. In the body, `receive_channel` is used to receive the `Counter` channel, and the continuation closure binds the `stream` context lens for accessing the `Counter` channel in the linear context. Inside the continuation closure (line 3), the expression `unfix_session_for (...)` have the type `PartialSession< (Counter, ()), End >`, so it has one channel of session type `Counter` in the linear context. Using the context lens `stream`, `unfix_session_for(stream, ...)` unrolls `Counter` so that the continuation at line 4 have the type `PartialSession< (SendValue<u64, Counter>, ()), End >`. With the recursive session type unrolled, `receive_value_from(stream, ...)` can be used to receive the integer value from `SendValue<u64, Counter>` using the `stream` context lens. In line 5 the continuation closure `async move | count | {...}` is passed to `receive_value_from` to bind the received integer value to `count`. Then the next line prints out the received count using `println!`.

Following that in line 7, the return type for the continuation closure has to be `PartialSession< (Counter, ()), End >`, which is the same type as in line 3. Experienced functional programmers may recognize that we could have done some inner recursion to get back to line 3, however this is not an option in Rust as inner recursive closure is not supported. Instead, we need to find some way of doing recursion back to `stream_client`, but this would require some way of converting the Rust type from `Session< ReceiveChannel<Counter, End> >` to `PartialSession< (Counter, ()), End >`. To do this, in line 7 we first call `include_session(stream_client (), ...)` to include a new copy of `Session< ReceiveChannel<Counter, End> >` into the current continuation by recursively calling `stream_client()`. The new channel is then bound to `next` inside the continuation closure in line 8. Following that, the continuation at line 9 have the type `PartialSession< (Counter, (ReceiveChannel<Counter, End>, ())), End >`, with the context lens `stream` bound to the first channel `Counter` and the context lens `next` bound to the second channel `ReceiveChannel < Counter, End >`. Comparing the session types of `stream` and `next`, we can notice that `stream` can be sent to `next`, which can be done using `send_channel_to(stream, next, ...)`. Finally in the continuation in line 10, we need to produce the Rust type `PartialSession< (Empty, (End, ())), End >`. This can be done by forwarding the channel `next` using `forward ( next )`, which then the empty linear context allows completion of the Ferrite program.