Ferrite: A Judgmental Embedding of Session Types in Rust

Ruo Fei Chen
Independent Researcher, Leipzig, Germany

Stephanie Balzer
Carnegie Mellon University, Pittsburgh, PA, USA

Bernardo Toninho
NOVA LINCS, Nova University Lisbon, Portugal

Abstract

Session types have proved viable in expressing and verifying the protocols of message-passing systems. While message passing is a dominant concurrency paradigm in practice, real world software is written without session types. A limitation of existing session type libraries in mainstream languages is their restriction to linear session types, precluding application scenarios that demand sharing and thus aliasing of channel references. This paper introduces Ferrite, a shallow embedding of session types in Rust that supports both *linear* and *shared* sessions. The formal foundation of Ferrite constitutes the shared session type calculus SILL_S, which Ferrite encodes via a novel *judgmental embedding* technique. The fulcrum of the embedding is the notion of a typing judgment that allows reasoning about shared and linear resources to type a session. Typing rules are then encoded as functions over judgments, with a valid typing derivation manifesting as a well-typed Rust program. This Rust program generated by Ferrite serves as a *certificate*, ensuring that the application will proceed according to the protocol defined by the session type. The paper details the features and implementation of Ferrite and includes a case study on implementing Servo's canvas component in Ferrite.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Type theory; Software and its engineering → Domain specific languages; Software and its engineering → Concurrent programming languages

Keywords and phrases Session Types, Rust, DSL

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.22

Related Version *Technical Report*: `https://arxiv.org/abs/2009.13619` [7]

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.14`

Funding *Stephanie Balzer*: National Science Foundation Award No. CCF-1718267.
Bernardo Toninho: FCT/MCTES grant NOVALINCS/BASE UIDB/04516/2020.

1 Introduction

Message-passing is a dominant concurrency paradigm, adopted by mainstream languages such as Erlang, Scala, Go, and Rust, putting the slogan *"Do not communicate by sharing memory; instead, share memory by communicating"* [13] into practice. In this setting, messages are exchanged along channels, which can be shared by several senders and receivers. Type systems in such languages typically allow channels to be typed, specifying and constraining the types of messages they may carry (e.g. integers, strings, sums, references, etc.).

An aspect inherent to message-passing concurrency that is not captured in mainstream type systems, however, is the idea of a *protocol*. Protocols dictate the sequencing and types of messages to be exchanged. To express and enforce such protocols, *session types* [14, 15, 16]

were introduced. Session typing disciplines assign types to channel endpoints according to their intended usage protocols in terms of sequencing of input/output actions (e.g. "send an integer and, afterwards, receive a string") and branching/selection actions (e.g. "receive either a buy message and process the payment; or a cancellation message and abort the transaction"), ensuring the action sequence is followed correctly and thus, adherence to the protocol. Thanks to their correspondence to *linear* logic [4, 44, 43, 42, 26, 5] session types enjoy a strong logical foundation and ensure, in addition to protocol adherence (*session fidelity*), the existence of a communication partner (*progress*). Session types have also been extended with safe *sharing* [1, 2, 3] to accommodate multi-client scenarios that are rejected by exclusively linear session types.

Despite these theoretical advances, session types have not (yet) been adopted at scale. While various session type embeddings exist in mainstream languages such as Java [18, 17], Scala [39], Haskell [38, 34, 20, 27], OCaml [32, 19], and Rust [21, 25, 8, 9], all of these embeddings lack support for multi-client scenarios that mandate controlled aliasing in addition to linearity.

This paper introduces *Ferrite* [6], a shallow embedding of session types in Rust. In contrast to prior work, Ferrite supports *both* linear and shared session types, with protocol adherence guaranteed statically by the Rust compiler. Ferrite's underlying theory is based on the calculus SILL_S introduced in [1], which develops the logical foundation of shared session types. As a matter of fact, Ferrite encodes SILL_S typing derivations as Rust functions, through a technique we dub *judgmental embedding*. Through our judgmental embedding, a type-checked Ferrite program yields a Rust program that corresponds to a SILL_S typing derivation and thus the *proof* of protocol adherence.

In order to faithfully encode SILL_S typing in Rust, this paper further makes several technical contributions to emulate advanced typing features, such as higher-kinded types, by a skillful combination of traits (type classes) and associated types (type families). For example, Ferrite supports recursive (session) types in this way, which are limited to recursive structs of a fixed size in plain Rust. A combination of type-level natural numbers with ideas from profunctor optics [33] are also used to support named channels and labeled choices. We adopt the idea of *lenses* [11] for selecting and updating individual channels in an arbitrary-length linear context. Similarly, we use *prisms* for selecting a branch out of arbitrary-length choices. Whereas `session-ocaml` [32] has previously explored the use of n-ary choice through extensible variants in OCaml, we are the first to connect n-ary choice to prisms and non-native implementation of extensible variants. Notably, the Ferrite codebase remains entirely in the *safe* fragment of Rust, with no (direct) use of unsafe features.

Given its support of both linear and shared session types, Ferrite is capable of expressing any session-typed program in Rust. We substantiate this claim by providing an implementation of Servo's canvas component with the communication layer within Ferrite.

This work makes the following contributions: **(i)** the design and implementation of *Ferrite*, an embedded domain-specific language (EDSL) for writing session-typed programs in Rust; **(ii)** with support of both *linear* and *shared* sessions, guaranteed to be observed by type checking; **(iii)** a novel *judgmental embedding* of custom typing rules in a host language with the resulting program carrying the proof of successful type checking; **(iv)** an encoding of *arbitrary-length choice* in terms of prisms and extensible variants in Rust; **(v)** an *empirical evaluation* based on a full implementation of Servo's canvas component in Ferrite.

All typing rules and their encoding as well as further materials of interest to an inquisitive reader are provided in our companion technical report [7].

Table 1 Overview of session types and terms in $\mathsf{SILL_S}$ together with their operational meaning. Subscripts L and S denote linear and shared sessions, resp., where $m, n \in \{\mathsf{L}, \mathsf{S}\}$.

Session type current	cont	Process term current	cont	Description
$c_\mathsf{L}{:}\oplus\{\overline{l{:}A_\mathsf{L}}\}$	$c_\mathsf{L}{:}A_{\mathsf{L}_h}$	$c_\mathsf{L}.l_h; P$	P	provider sends label l_h along c_L
		$\mathsf{case}\ c_\mathsf{L}\ \mathsf{of}\ \overline{l \Rightarrow Q}$	Q_h	client receives label l_h along c_L
$c_\mathsf{L}{:}\&\{\overline{l{:}A_\mathsf{L}}\}$	$c_\mathsf{L}{:}A_{\mathsf{L}_h}$	$\mathsf{case}\ c_\mathsf{L}\ \mathsf{of}\ \overline{l \Rightarrow P}$	P_h	provider receives label l_h along c
		$c_\mathsf{L}.l_h; Q$	Q	client sends label l_h along c_L
$c_\mathsf{L}{:}A_m \otimes B_\mathsf{L}$	$c_\mathsf{L}{:}B_\mathsf{L}$	$\mathsf{send}\ c_\mathsf{L}\ d_m; P$	P	provider sends channel $d_m{:}A_m$ along c_L
		$y_m \leftarrow \mathsf{recv}\ c_\mathsf{L}; Q_{y_m}$	Q_{d_m}	client receives channel $d_m{:}A_m$ along c_L
$c_\mathsf{L}{:}A_m \multimap B_\mathsf{L}$	$c_\mathsf{L}{:}B_\mathsf{L}$	$y_m \leftarrow \mathsf{recv}\ c_\mathsf{L}; P_{y_m}$	P_{d_m}	provider receives channel $d_m{:}A_m$ along c_L
		$\mathsf{send}\ c_\mathsf{L}\ d_m; Q$	Q	client sends channel $d_m{:}A_m$ along c_L
$c_\mathsf{L}{:}\mathbf{1}$	-	$\mathsf{close}\ c_\mathsf{L}$	-	provider sends "end" along c_L
		$\mathsf{wait}\ c_\mathsf{L}; Q$	Q	provider receives "end" along c_L
$c_\mathsf{L}{:}\downarrow^\mathsf{S}_\mathsf{L} A_\mathsf{S}$	$c_\mathsf{S}{:}A_\mathsf{S}$	$c_\mathsf{S} \leftarrow \mathsf{detach}\ c_\mathsf{L}; P_{x_\mathsf{S}}$	P_{c_S}	provider sends "$\mathsf{detach}\ c_\mathsf{S}$" along c_L
		$x_\mathsf{S} \leftarrow \mathsf{release}\ c_\mathsf{L}; Q_{x_\mathsf{S}}$	Q_{c_S}	client receives "$\mathsf{detach}\ c_\mathsf{S}$" along c_L
$c_\mathsf{S}{:}\uparrow^\mathsf{S}_\mathsf{L} A_\mathsf{L}$	$c_\mathsf{L}{:}A_\mathsf{L}$	$c_\mathsf{L} \leftarrow \mathsf{acquire}\ c_\mathsf{S}; Q_{x_\mathsf{L}}$	Q_{c_L}	client sends "$\mathsf{acquire}\ c_\mathsf{L}$" along c_S
		$x_\mathsf{L} \leftarrow \mathsf{accept}\ c_\mathsf{S}; P_{x_\mathsf{L}}$	P_{c_L}	provider receives "$\mathsf{acquire}\ c_\mathsf{L}$" along c_S
$c_m : A_m$	$c_m : A_m$	$z_n \leftarrow X \leftarrow \overline{d_m}; P_{z_n}$	P_{z_n}	spawn ("cut") X along $z_n{:}B_n$ with $\overline{d_m}{:}\overline{D_m}$
$c_m : A_m$	-	$\mathsf{fwd}\ c_m\ d_m$	-	forward to channel $d_m{:}A_m$ and terminate

2 Background

This section gives a brief tour of linear and shared session types. The presentation is based on the intuitionistic session-typed process calculus $\mathsf{SILL_S}$ [1], which Ferrite builds upon. We consider the protocol governing the interaction between a queue and its client:

$$\mathsf{queue}\ A = \&\{\mathsf{enq} : A \multimap \mathsf{queue}\ A,\ \mathsf{deq} : \oplus\{\mathsf{none} : \mathbf{1},\ \mathsf{some} : A \otimes \mathsf{queue}\ A\}\}$$

Table 1 provides an overview of the types used in the example. Since $\mathsf{SILL_S}$ is based on a Curry-Howard correspondence between intuitionistic linear logic and the session-typed π-calculus [4, 5] it uses linear logic connectives ($\oplus$, $\&$, $\otimes$, $\multimap$, $\mathbf{1}$) as session types. The remaining connectives concern shared sessions, a feature we remark on shortly. A crucial – and probably unusual – characteristic of session-typed processes is that a process *changes* its typing along with the messages it exchanges. As a result, a process' typing always reflects the current protocol state. Table 1 lists state transitions inflicted by a message exchange in the first and second column and corresponding process terms in the third and fourth column. The fifth column provides the operational meaning of a type.

Consulting Table 1, we gather that the above polymorphic session type $\mathsf{queue}\ A$ imposes the following recursive protocol: A client may either send the label enq or deq to the queue, depending on whether the client wishes to enqueue or dequeue an element of type A, resp. In the former case, the client sends the element to be enqueued, after which the queue recurs. In the latter case, the queue indicates to the client whether it is empty (none) or not (some), and proceeds by either terminating or sending the dequeued element and recurring, resp.

A linear typing discipline is beneficial because it immediately guarantees session fidelity – even in the presence of perpetual protocol change – by ensuring that a channel connects exactly two processes. Unfortunately, linearity also rules out various practical programming scenarios that demand sharing and thus aliasing of channel references. For example, the above linear session type $\mathsf{queue}\ A$ is limited to a *single* client. To support safe sharing of

stateful channel references while upholding session fidelity, $\mathsf{SILL_S}$ extends linear session types with shared session types ($\downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}$, $\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$). These two connectives mediate between shared and linear sessions by requiring that clients of shared sessions interact in *mutual exclusion* from each other. Concretely, a type $\uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}$ mandates a client to *acquire* the process offering the shared session. If the request is successful, the client receives a linear channel to the acquired process along which it must proceed as detailed by the session type A_{L}. A type $\downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}}$, on the other hand, mandates a client to *release* the linear process, relinquishing ownership of the linear channel and only being left with a shared alias to the now shared process at type A_{S}.

Using these connectives, we can turn the above linear queue into a shared one, bracketing enqueue and dequeue operations within acquire-release:

$$\mathsf{squeue}\ A_{\mathsf{S}} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \& \{\mathsf{enq} : A_{\mathsf{S}} \multimap \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{squeue}\ A_{\mathsf{S}},\ \mathsf{deq} : \oplus\{\mathsf{none} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{squeue}\ A_{\mathsf{S}},\ \mathsf{some} : A_{\mathsf{S}} \otimes \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{squeue}\ A_{\mathsf{S}}\}\}$$

In contrast to the linear queue, the above version recurs in the none branch and thus keeps the queue alive to serve the next client. For convenience, $\mathsf{SILL_S}$ allows the connectives $\otimes$ and $\multimap$ to be used to transport both linear and shared channels along a linear carrier channel.

To provide a flavor of session-typed programming in $\mathsf{SILL_S}$, we briefly comment on the below processes *empty* and *elem*, which implement the shared queue session type as a sequence of *elem* processes, ended by an *empty* process. A process implementation consists of its signature (first two lines) and body (after $=$). The first line indicates the typing of channel variables used by the process (left of $\vdash$) and the type of the providing channel variable (right of $\vdash$). The second line binds the channel variables. In $\mathsf{SILL_S}$, $\leftarrow$ generally denotes variable bindings. We leave it to the reader to convince themselves, consulting Table 1, that the code in the body of the two processes executes the protocol defined by session type $\mathsf{squeue}\ A_{\mathsf{S}}$.

```
· ⊢ empty :: q : squeue A_S
q ← empty ← · =
  q' ← accept q ;
  case q' of
  | enq → x ← recv q' ;
          q ← detach q' ;
          e ← empty ; q ← elem ← x, e
  | deq → q'.none ;
          q ← detach q' ;
          q ← empty
```

```
x : A_S, t : squeue A_S ⊢ elem :: q : squeue A_S
q ← elem ← x, t =
  q' ← accept q ;
  case q' of
  | enq → y ← recv q' ;
          t' ← acquire t ;
          t'.enq ; send t' y ;
          t ← release t' ; q ← detach q' ;
          q ← elem ← x, t
  | deq → q'.some ; send q' x ;
          q ← detach q' ; fwd q t
```

Imposing acquire-release not only as a programming methodology but also as a *typing discipline* has the advantage of recovering session fidelity for shared sessions. To this end, shared session types in $\mathsf{SILL_S}$ must be *strictly equi-synchronizing* [1, 3], imposing the invariant that an acquired session is released to the type at which previously acquired. For example, the shared session type $\mathsf{squeue}\ A_{\mathsf{S}}$ is strictly equi-synchronizing whereas the type $\mathsf{invalid} = \uparrow^{\mathsf{S}}_{\mathsf{L}} \& \{\mathsf{left} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \uparrow^{\mathsf{S}}_{\mathsf{L}} \oplus \{\mathsf{yes} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{invalid}, \mathsf{no} : \mathbf{1}\}, \mathsf{right} : \downarrow^{\mathsf{S}}_{\mathsf{L}} \mathsf{invalid}\}$ is not.

It is instructive to review the typing rules for acquire-release:

$$(\text{T-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\text{L})\quad \frac{\Psi, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}; \Delta, y_{\mathsf{L}} : A_{\mathsf{L}} \vdash Q_{y_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Psi, x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}}; \Delta \vdash y_{\mathsf{L}} \leftarrow \mathsf{acquire}\, x_{\mathsf{S}}\, ; Q_{y_{\mathsf{L}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}$$

$$(\text{T-}\uparrow^{\mathsf{S}}_{\mathsf{L}}\text{R})\quad \frac{\Psi; \cdot \vdash P_{y_{\mathsf{L}}} :: (y_{\mathsf{L}} : A_{\mathsf{L}})}{\Psi \vdash y_{\mathsf{L}} \leftarrow \mathsf{accept}\, x_{\mathsf{S}}\, ; P_{y_{\mathsf{L}}} :: (x_{\mathsf{S}} : \uparrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{L}})}$$

$$(\text{T-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\text{L})\quad \frac{\Psi, x_{\mathsf{S}} : A_{\mathsf{S}}; \Delta \vdash Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}{\Psi; \Delta, y_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}} \vdash x_{\mathsf{S}} \leftarrow \mathsf{release}\, y_{\mathsf{L}}\, ; Q_{x_{\mathsf{S}}} :: (z_{\mathsf{L}} : C_{\mathsf{L}})}$$

$$(\text{T-}\downarrow^{\mathsf{S}}_{\mathsf{L}}\text{R})\quad \frac{\Psi \vdash P_{x_{\mathsf{S}}} :: (x_{\mathsf{S}} : A_{\mathsf{S}})}{\Psi; \cdot \vdash x_{\mathsf{S}} \leftarrow \mathsf{detach}\, y_{\mathsf{L}}\, ; P_{x_{\mathsf{S}}} :: (y_{\mathsf{L}} : \downarrow^{\mathsf{S}}_{\mathsf{L}} A_{\mathsf{S}})}$$

Due to its foundation in intuitionistic linear logic, $\mathsf{SILL_S}$' typing rules are phrased using a *sequent calculus*, leading to *left* and *right* rules for each connective. Left rules describe the interaction from the point of view of the client, right rules from the point of view of the provider. The typing judgments $\Psi; \Delta \vdash P :: (x_{\mathsf{L}} : A_{\mathsf{L}})$ and $\Psi \vdash P :: (x_{\mathsf{S}} : A_{\mathsf{S}})$ read as "process P offers a session of type A along channel x using sessions offered along channels in Ψ (and Δ)." The typing contexts Ψ and Δ provide the typing of shared and linear channels, resp. Whereas Ψ is a structural context, Δ is a linear context, forbidding channels to be dropped (weakened) or duplicated (contracted). In contrast to linear processes, shared processes must not use any linear channels, a requirement crucial for type safety. The notions of acquire and release are naturally formulated from the point of view of a client, so these terms appear in the left rules. The right rules use the terms *accept* and *detach* with the meaning that an accept accepts an acquire and a detach initiates a release. The rules are read bottom-up, where the premise denotes the next action to be taken after the message exchange.

3 Key Ideas

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

3.1 $\mathsf{SILL_R}$ – A stepping stone from $\mathsf{SILL_S}$ to Ferrite

In Section 2, we reviewed $\mathsf{SILL_S}$ and its typing judgment. Our goal with Ferrite is to faithfully and compositionally encode $\mathsf{SILL_S}$ typing derivations in Rust. However, when viewed under the lens of a general purpose programming language, most readers will find $\mathsf{SILL_S}$ a prohibitively austere formalism, lacking most facilities needed to write realistic programs (e.g. basic data types, pattern matching, etc.) and provided by a convenient and usable programming language like Rust. From an ergonomics standpoint alone it would be unreasonably prohibitive for our embedding to forbid the use of Rust features such as functions, traits and enumerations, only for the sake of precisely mirroring $\mathsf{SILL_S}$. Moreover, to realize such an embedding we must be able to account for both $\mathsf{SILL_S}$' linear session discipline (i.e. the *linear* context Δ) and shared session discipline (i.e. the *structural* context Ψ) within Rust's usage discipline. Since Rust's typing discipline is essentially *affine*, its treatment of variable usage is neither linear nor purely structural, and so both shared and linear channels must be treated explicitly in the encoding.

The two points above naturally lead us to the language $\mathsf{SILL_R}$ as a formal stepping stone between $\mathsf{SILL_S}$ and our embedding, Ferrite. $\mathsf{SILL_R}$ is, in its essence, a pragmatic extension of $\mathsf{SILL_S}$ with Rust (type and term) constructs, allowing us to intersperse Rust code with the communication primitives of $\mathsf{SILL_S}$. In $\mathsf{SILL_R}$ we use the judgment $\Gamma; \Delta \vdash \mathit{expr} :: A$, denoting that expression *expr* has session type A, using the sessions tracked by Γ and Δ.

This judgment differs from that of $\mathsf{SILL_S}$ in its context region Γ and term *expr*, with the latter permitting arbitrary Rust expressions in addition to $\mathsf{SILL_S}$ primitives. Whereas $\mathsf{SILL_S}$'s structural context Ψ exclusively tracks shared channels, $\mathsf{SILL_R}$'s Γ tracks *both* shared sessions (subject to weakening and contraction) and plain Rust (affine) variables. A shared channel type in both $\mathsf{SILL_R}$ and $\mathsf{SILL_S}$ is always of the form $\uparrow^{\mathsf{S}}_{\mathsf{L}} A$, so there is no confusion among the affine and shared contents of Γ. As we discuss in Section 5.2, the distinction between a plain Rust variable, which is treated as affine, and a shared channel, which is treated structurally, is modelled in Ferrite by making shared channels implement Rust's `Clone` trait.

Table 2 Overview of SILL_R types and terms and their encoding in Ferrite. Note that SILL_R uses $\tau \triangleleft A_\mathsf{L}$ and $\tau \triangleright A_\mathsf{L}$ for shared channel output and input, resp., and ϵ for termination.

Type **Ferrite**	SILL_R	**Terms** (SILL_R) **provider**	**client**
`InternalChoice<Row>`	$\oplus\{\overline{l_i : A_{\mathsf{L}_i}}\}$	offer l_i; K	case a $\{\overline{l_i : K_i}\}$
`ExternalChoice<Row>`	$\&\{\overline{l_i : A_{\mathsf{L}_i}}\}$	offer_choice$\{\overline{l_i : K_i}\}$	choose a l_i; K
`SendChannel<A,B>`	$A_\mathsf{L} \otimes B_\mathsf{L}$	send_channel_from a; K	$a \leftarrow$ receive_channel_from f a; K
`ReceiveChannel<A,B>`	$A_\mathsf{L} \multimap B_\mathsf{L}$	$a \leftarrow$ receive_channel; K	send_channel_to f a; K
`SendValue<T,A>`	$\tau \triangleleft A_\mathsf{L}$	send_value x; K	$x \leftarrow$ receive_value_from a x; K
`ReceiveValue<T,A>`	$\tau \triangleright A_\mathsf{L}$	$x \leftarrow$ receive_value; K	send_value_to a x; K
`End`	ϵ	terminate	wait a; K
`SharedToLinear<A>`	$\downarrow^\mathsf{S}_\mathsf{L} A_\mathsf{S}$	detach_shared_session; K_s	release_shared_session a; K_l
`LinearToShared<A>`	$\uparrow^\mathsf{S}_\mathsf{L} A_\mathsf{L}$	accept_shared_session; K_l	$a \leftarrow$ acquire_shared_session s; K_l

Table 2 provides an overview of SILL_R types and terms and their Ferrite encoding. SILL_R types stand in direct correspondence with SILL_S types (see Table 1), apart from shared channel output and input. The SILL_S types for sending and receiving shared channels ($A_\mathsf{S} \otimes A_\mathsf{L}$ and $A_\mathsf{S} \multimap A_\mathsf{L}$) correspond to SILL_R types for sending and receiving values ($T \triangleleft A$ and $T \triangleright A$, resp.), which support *both* Rust values and shared channels. Their typing rules are:

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

Rule $\mathrm{T}\triangleleft_\mathsf{R}$ indicates that the value bound to variable x of type τ will be sent, after which the continuation K will execute, offering type A. Dually, rule $\mathrm{T}\triangleleft_\mathsf{L}$ states that using such a provider bound to a will bind x of type τ in continuation K, which must now use the channel bound to a according to A.

3.2 Judgmental Embedding

Having introduced the SILL_R typing judgment and illustrated some of its typing rules, we can now clarify the idea behind our notion of *judgmental embedding*, which enables the Rust compiler to typecheck SILL_R programs by encoding typing derivations 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<...>
  ( cont: PartialSession<C2, A2> )
  -> PartialSession<C1, A1>
```

On the left we show a SILL_R typing rule and on the right its encoding in Ferrite. Ferrite encodes a SILL_R *typing judgment* $\Gamma;\Delta \vdash \mathit{expr} :: A$ as a value of Rust *type* `PartialSession<C, A>`, where `C` encodes the linear context Δ and `A` the session type A, standing for any of the Ferrite types of Table 2. Ferrite then encodes a SILL_R *typing rule* for an expression *expr* as a Rust *function* `expr` that accepts a `PartialSession<C2, A2>` and returns a `PartialSession<C1, A1>`, where *expr* stands for any of the SILL_R terms of Table 2. The encoding makes use of *continuation passing style* (arising from the sequent calculus-based formulation of SILL_R), with the return type being the conclusion of the rule and the argument type being its premise. Table 3 summarizes the judgmental embedding; Section 4.1 provides further details. Whereas Ferrite explicitly performs a type-level encoding of the linear context Δ, the representation of the shared and affine context region Γ is achieved through Rust's normal

Table 3 Judgmental embedding of SILL_R in Ferrite.

SILL_R	Ferrite	Description
$\Gamma\,;\cdot \vdash A$	`Session<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 / Affine context; delegated to Rust.
A	`A: Protocol`	Session type.

binding structure, with the obligation that shared channels implement Rust's `Clone` trait to permit contraction. To type a closed program, Ferrite defines the type `Session<A>`, which stands for a SILL_R judgment with an empty linear context.

Adopting a judgmental embedding technique for implementing a DSL delivers the benefits of proof-carrying code: the `PartialSession<C1, A1>` returned from a well-typed Ferrite `expr` *is* the typing derivation of the corresponding SILL_R term. In case the SILL_R term is a SILL_S term, its typing derivation certifies protocol adherence by virtue of the type safety proof of SILL_S [1]. In case the SILL_R term includes Rust code, its typing derivation certifies protocol adherence modulo the possibility of a panic raised by the Rust code. A fully general type safety result for SILL_R, possibly building upon existing formalizations of Rust [22], is an avenue of future work.

3.3 Recursive and Shared Session Types in Ferrite

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 `Rec<F>` to obtain the fixed point of a type function `F`. Since Rust lacks higher-kinded types such as `Type` → `Type`, we use *defunctionalization* [36, 46] by accepting any Rust type `F` implementing the trait `RecApp` with a given associated type `F::Applied`, as shown below. Section 5.1 provides further details.

```
trait RecApp<X> { type Applied; }
struct Rec<F: RecApp<Rec<F>>> { unfold: Box<F::Applied> }
```

Recursive types are also vital for encoding shared session types. In line with [3], we restrict shared session types to be recursive, making sure that a shared component is continuously available. To guarantee type preservation, recursive session types must be *strictly equi-synchronizing* [1, 3], requiring an acquired session to be released to the same type at which it was previously acquired. Ferrite enforces this invariant by defining a specialized trait `SharedRecApp` which omits an implementation for `End`:

```
trait SharedRecApp<X> { type Applied; }   trait SharedProtocol { ... }
struct SharedToLinear<F> { ... }    struct SharedChannel<S: SharedProtocol> { ... }
struct LinearToShared<F: SharedRecApp<SharedToLinear<LinearToShared<F>>>> { ... }
```

Ferrite achieves safe communication for shared sessions by imposing an acquire-release discipline [1] on shared sessions, establishing a critical section for the linear portion of the process enclosed within acquire and release. `SharedChannel` denotes the shared process running in the background, and clients with a reference to it can *acquire* an exclusive linear channel to communicate with it. As long as the linear channel exists, the shared process is locked and cannot be acquired by any other client. With the strictly equi-synchronizing constraint in place, the now linear process must eventually be released (`SharedToLinear`) back to the same shared session type at which it was previously acquired, giving turn to another client waiting to acquire. Section 5.2 provides further details on the encoding.

3.4 N-ary Choice and Linear Context

Ferrite implements n-ary *choices* and linear typing *contexts* as extensible *sums* and *products* of session types, resp. Ferrite uses heterogeneous lists [23] to annotate a list of session types of arbitrary length. The notation HList![$A_0, A_1, ..., A_{N-1}$] denotes a heterogeneous list of N session types, with A_i being the session type at the i-th position of the list. The HList! macro acts as syntactic sugar for the heterogeneous list, which in its raw form is encoded as $(A_0, (A_1, (..., (A_{N-1}, ()))))$. Ferrite uses the Rust tuple constructor `(,)` for HCons, and unit `()` for HNil. The heterogeneous list itself can be directly used to represent an n-ary product. Using an associated type, the list can moreover be transformed into an n-ary sum.

One disadvantage of using heterogeneous lists is that its elements have to be addressed by position rather than a programmer-chosen label. To recover labels for accessing list elements, we use optics [33]. More precisely, Ferrite uses *lenses* [11] to access a channel in a linear context and *prisms* to select a branch of a choice. We further combine the optics abstraction with *de Bruijn levels* and implement lenses and prisms using type level natural numbers. Given an inductive trait definition of natural numbers as zero (`Z`) and successor (`S<N>`), a natural number `N` implements the lens to access the N-th element in the linear context, and the prism to access the N-th branch in a choice. Schematically, the lens encoding can be captured as follows:

$$\frac{\Gamma\,;\Delta,\, l_n : B_2 \;\vdash\; K :: A_2}{\Gamma\,;\Delta,\, l_n : B_1 \;\vdash\; \mathit{expr}\ l_n; K :: A_1}$$

```
fn expr<...>
  ( l: N, cont: PartialSession<C1, A2> )
  -> PartialSession<C2, A1>
where N: ContextLens<C1, B1, B2, Target=C2>
```

The index `N` amounts to the type of the variable `l` that the programmer chooses as a name for a channel in the linear context. Ferrite handles the mapping, supporting random access to programmer-named channels. Section 4.2 provides further details, including the support of higher-order channels. Similarly, prisms allow choice selection in constructs such as `offer_case` to be encoded as follows:

$$\frac{\Gamma;\Delta \;\vdash\; K :: A_n}{\Gamma;\Delta \;\vdash\; \mathsf{offer\_case}\ l_n;\; K :: \oplus\{..., l_n : A_n, ...\}}$$

```
fn offer_case<N, Row, C, A>
  ( l: N, cont: PartialSession<C, A> )
  -> PartialSession<C, InternalChoice<Row>>
where N: Prism<Row, Elem=A>, ...
```

Ferrite maps a choice label to a constant having the singleton value of a natural number `N`, which implements the prism to access the N-th branch of a choice. In addition to prisms, Ferrite implements a version of *extensible variants* [28] to support polymorphic operations on arbitrary sums of session types representing choices. Finally, the `define_choice!` macro is used as a helper to export type aliases as programmer-friendly identifiers. Details are reported in Section 6 and in our companion technical report [7].

4 Ferrite – A Judgmental Embedding of SILL$_R$

Having introduced some of the key concepts underlying the implementation of Ferrite, we now cover in detail the implementation of Ferrite's core constructs, building up the knowledge required for Section 5 and Section 6. Ferrite, like any other DSL, has to tackle the various technical challenges encountered when embedding a DSL in a host language. In doing so, we take inspiration from the range of embedding techniques developed for Haskell and adjust them to the Rust setting. The lack of higher-kinded types, limited support of recursive types, and presence of weakening, in particular, make the development far from trivial. A more conceptual contribution of this work is thus to demonstrate how existing Rust features can be combined to emulate many of the missing features that are beneficial to DSL embeddings

and how to encode custom typing rules in Rust or any similarly expressive language. The techniques described in this and subsequent sections also serve as a reference for embedding other DSLs in a host language like Rust.

4.1 Encoding Typing Rules via Judgmental Embedding

A distinguishing characteristic of Ferrite is its *propositions as types* approach, yielding a direct correspondence between SILL_R notions and their Ferrite encoding. This correspondence was introduced in Section 3.2 (see Table 3) and we now discuss it in more detail. To this end, let's consider the typing of value input. We remind the reader of Table 2 in Section 3, which provides a mapping between SILL_R and Ferrite session types. Interested readers can find a corresponding mapping on the term level in the companion technical report [7].

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

The SILL_R right rule $\mathrm{T}\triangleright_\mathsf{R}$ types expression $a \leftarrow \mathsf{receive\_value};\, K$ with session type $\tau \triangleright A$ and the continuation K with session type A, where a is now in scope with type τ. Following the schema hinted in Section 3.2, 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`.

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

The function yields a value of type `PartialSession<C, ReceiveValue<T, A>>`, i.e. the conclusion of the rule, given an (affine) closure of type `T` $\rightarrow$ `PartialSession<C, A>`, encoding the premise of the rule. Notably, Ferrite uses plain Rust binding (through function types) to encode the contents of Γ, as illustrated for the received value above. 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 affine closure implements the `FnOnce` trait, ensuring that it can only be called once.

The type `PartialSession` is a core construct 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`, corresponding to the SILL_R typing judgment $\Gamma;\Delta \vdash \mathit{expr} :: A$. 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, resp.:

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

For each SILL_R session type, Ferrite defines a corresponding Rust struct that implements the trait `Protocol`, yielding the listing shown in Table 2. 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 also implement `Protocol`:

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

Thus, while Ferrite delegates the handling of the shared/structural context Γ to Rust, the encoding of the linear context Δ is explicit. Being affine, the Rust type system permits weakening, a structural property rejected by linear logic. Ferrite encodes a linear context as a heterogeneous (type-level) list [23] of the form `HList![`$\mathtt{A}_0$`, `$\mathtt{A}_1$`, ..., `$\mathtt{A}_{N-1}$`]`, with all its type

elements A_i implementing `Protocol`. Internally, the `HList` macro desugars the type-level list into a nested tuple $(A_0, (A_1, (\ldots, (A_{N-1}, ()))))$. The unit type `()` is used as the empty list (`HNil`) and the tuple constructor `(,)` is used as the `HCons` constructor. The implementation for `Context` is defined inductively as follows:

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

To represent a closed program, i.e. a program without free channel variables, we define a type alias `Session<A>` for `PartialSession<C, A>`, with `C` restricted to the empty context:

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

A complete session type program in Ferrite is thus of type `Session<A>` and amounts to the SILL_R typing derivation proving that the program adheres to the defined protocol. Below we show a "hello world"-style program in Ferrite:

```
let hello_provider = receive_value(|name| { println!("Hello, {}", name); terminate() });
```

The Ferrite program `hello_provider` has an inferred Rust type `Session<ReceiveValue<String , End>>`. It offers the type `ReceiveValue<String, End>` by first receiving a string value using `receive_value`, binding it to `name` in the continuation closure. Upon receiving the name string, It prints out the name with a `"Hello"` greeting, and terminates using `terminate()`.

4.2 Manipulating the Linear Context

Context Lenses

The use of a type-level list to encode the linear context has the advantage of allowing contexts of arbitrary length. However, the list imposes an order on the context's elements, disallowing exchange. To allow exchange, we make use of the concept of *lenses* [11] to define a `ContextLens` trait, which is implemented using type-level natural numbers.

```
#[derive(Copy)] struct Z;     #[derive(Copy)] struct S<N>(PhantomData<N>);
trait ContextLens<C: Context, A1: Protocol, A2: Protocol> { type Target: Context; ... }
```

The `ContextLens` trait defines the read and update operations on a linear context, such that given a *source* context `C = HList![..., A_N, ...]`, the source element of interest, A_N at position N, can be updated to the target element `B` to form the *target* context `Target = HList![..., B, ...]`, with the remaining elements unchanged. We use natural numbers to inductively implement `ContextLens` at each position in the linear context, such that it satisfies all constraints of the form:

```
N: ContextLens<HList![..., A_N, ...], A_N, B, Target=HList![..., B, ...]>
```

The implementation of natural numbers as context lenses is done by first considering the base case, with `Z` used to access the first element of any non-empty linear context:

```
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); ... }
```

In the inductive case, for any natural number `N` implementing the context lens for a context `HList![A_0, ..., A_N, ...]`, it's successor `S<Z>` implements the context lens for `HList![A_-1, A_0, ..., A_N, ...]`, with a new element A_{-1} appended to the head of the linear context. Using context lenses, we can encode the SILL_R left rule $\mathrm{T}\triangleright_\mathsf{L}$ shown below, which types sending an ambient value x to a channel a in the linear context that expects to receive a value.

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

In Ferrite, $\mathrm{T}\triangleright_{\mathrm{L}}$ is implemented as the function `send_value_to`, which uses a context lens `N` to send a value of type `T` to the `N`-th channel in the linear context `C1`. This requires the `N`-th channel to have type `ReceiveValue<T,A>`. A continuation `cont` is then given with the linear context `C2`, which has the `N`-th channel updated to type `A`.

```
fn send_value_to<N, T, C1: Context, C2: Context, A: Protocol, B: Protocol>
  ( n: N, x: T, cont: PartialSession<C2, B> ) -> PartialSession <C1, B>
where N: ContextLens<C1, ReceiveValue<T, A>, A, Target=C2>
```

Channel Removal

The above definition of a context lens is suited for *updating* channel types in a context. However, we have not addressed how channels can be *removed* or *added* to the linear context. These operations are required to implement session termination and higher-order channel constructs such as $\otimes$ and $\multimap$. To support channel removal, we introduce a special `Empty` element to denote the *absence* of a channel at a given position in the linear context:

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

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`.

$$\frac{\Gamma\,;\,\Delta \vdash K :: A}{\Gamma\,;\,\Delta,\, a : \epsilon \vdash \mathsf{wait}\, a;\; K :: A}\;(\mathrm{T1}_{\mathrm{L}}) \qquad \frac{}{\Gamma\,;\,\cdot \vdash \mathsf{terminate};\; :: \epsilon}\;(\mathrm{T1}_{\mathrm{R}})$$

Using `Empty`, it is straightforward to implement $\mathsf{SILL}_{\mathsf{R}}$'s session termination. Rule $\mathrm{T1}_{\mathrm{L}}$ is encoded via a context lens that replaces a channel of session type `End` with the `Empty` slot. The function `wait` shown below does not really remove a slot from a linear context, but merely replaces the slot with `Empty`. The use of `Empty` is necessary, because we want to preserve the position of channels in a linear context in order for the context lens for a channel to work across continuations.

```
fn wait<C1: Context, C2: Context, A: Protocol, N>
  ( n: N, cont: PartialSession<C2, A> ) -> PartialSession<C1, A>
where N: ContextLens<C1, End, Empty, Target=C2>
```

With `Empty` introduced, an empty linear context may now contain any number of `Empty` slots (e.g., `HList![Empty, Empty]`). We introduce an `EmptyContext` trait to abstract over the different forms of empty linear contexts and provide an inductive definition as its implementation:

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

Given the empty list `()` as the base case, the inductive case `(Empty, C)` is an empty linear context, if `C` is also an empty linear context. Using the definition of an empty context, the $\mathsf{SILL}_{\mathsf{R}}$ right rule $\mathrm{T}\mathbf{1}_{\mathrm{R}}$ can then be easily encoded as the function `terminate`, which works generically for all contexts that implement `EmptyContext` as shown below:

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

Channel Addition

The Ferrite function `wait` removes a channel from the linear context by replacing it with `Empty`. Dually, the function `receive_channel`, adds a new channel to the linear context. The SILL_R rule $\mathrm{T}{\multimap_\mathsf{R}}$ 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.

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

To encode $\mathrm{T}{\multimap_\mathsf{R}}$, an append operation on contexts is defined via the `AppendContext` trait:

```
trait AppendContext<C: Context>: Context { type Appended: Context; ... }
impl<C: Context> AppendContext<C> for () { type Appended = C; ... }
impl<A: Slot, C1: Context, C2: Context, C3: Context> AppendContext<C2>
  for (A, C1) where C1: AppendContext<C2, Appended=C3> { type Appended = (A, C3); ... }
```

The `AppendContext` trait is parameterized by a linear context `C` and an associated type `Appended`. If a linear context `C1` implements the trait `AppendContext<C2>`, it means that context `C2` can be appended to `C1`, with `C3` = `C1::Appended` being the result of the append operation. The implementation of `AppendContext` is defined inductively, with the empty list `()` implementing the base case and the cons cell `(A, C)` implementing the inductive case.

Using `AppendContext`, a channel `B` can be appended to the end of a linear context `C`, if `C` implements `AppendContext<HList![B]>`. The new linear context after the append operation is given in the associated type `C::Appended`. We then 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 obtaining the length of `C`. In Ferrite, the length operation is implemented by adding an associated type `Length` to the `Context` trait. The implementation of `Context` for `()` and `(A, C)` is updated correspondingly.

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

The SILL_R right rule $\mathrm{T}{\multimap_\mathsf{R}}$ is then encoded as follows:

```
fn receive_channel<A: Protocol, B: Protocol, C1: Context, C2: Context>(
  cont: impl FnOnce(C1::Length) -> PartialSession<C2, B>) ->
  PartialSession<C1, ReceiveChannel<A, B>> where C1: AppendContext<(A, ()), Appended=C2>
```

The function `receive_channel` is parameterized by a linear context `C1` implementing `AppendContext` to append the session type `A` to `C1`. The continuation argument `cont` is a closure that is given a context lens `C::Length`, and returns a `PartialSession` with `C2=C1::Appended` as its linear context. The function returns a `PartialSession` with linear context `C1`, offering session type `ReceiveChannel<A, B>`.

We note that in the type signature of `receive_channel`, the type `C1::Length` is not shown to have any `ContextLens` implementation. However when `C1::Length` is instantiated to the concrete types `Z`, `S<Z>`, etc in the continuation body, Rust will use the appropriate implementations of `ContextLens` so that they can be used to access the appended channel in the linear context.

The use of `receive_channel` is illustrated with the `hello_client` example below:

```
let hello_client = receive_channel(|a| {
  send_value_to(a, "Alice".to_string(), wait(a, terminate())) });
```

The `hello_client` program is inferred to have the Rust type `Session<ReceiveChannel< ReceiveValue<String, End>, End>>`. It is written to communicate with the `hello_provider` program defined earlier in Section 4.1. The interaction is achieved by having `hello_client` offering the session type `ReceiveChannel<ReceiveValue<String, End>, End>`. In its body, `hello_client` uses `receive_channel` to receive channel `a` of type `ReceiveValue<String, End>` from

`hello_provider`. The continuation closure is given an argument `a:Z`, denoting the context lens generated by `receive_channel` for accessing the received channel in the linear context. The context lens `a:Z` is then used for sending a string value, after which we `wait` for `hello_provider` to terminate. We note that the type `Z` of channel `a` (i.e. the channel position in the context) is automatically inferred by Rust and not exposed to the user.

4.3 Communication

At this point we have defined the necessary constructs to build and typecheck both `hello_provider` and `hello_client`, but the two are separate Ferrite programs that are yet to be linked with each other and executed.

$$\frac{\Gamma\,;\,\Delta_1 \vdash K_1 :: A \qquad \Gamma\,;\,\Delta_2, a : A \vdash K_2 :: B}{\Gamma\,;\,\Delta_1, \Delta_2 \vdash a \leftarrow \mathsf{cut}\ K_1\,;\ K_2 :: B}\ (\text{T-CUT}) \qquad \frac{}{\Gamma\,; a : A \vdash \mathsf{forward}\ a :: A}\ (\text{T-FWD})$$

In SILL_R, rule T-CUT allows two session-typed programs to run in parallel, with the channel offered by K_1 added to the linear context of program K_2. Together with the forward rule T-FWD, we can use cut twice to run both `hello_provider` and `hello_client` in parallel, and have a third program that sends the channel offered by `hello_provider` to `hello_client`. The program `hello_main` would have the following pseudo code in SILL_R:

$$\begin{aligned} \mathit{hello\_main} : \epsilon \ = \ & f \leftarrow \mathsf{cut}\ \mathit{hello\_client}; a \leftarrow \mathsf{cut}\ \mathit{hello\_provider}; \\ & \mathsf{send\_channel\_to}\ f\ a; \mathsf{forward}\ f \end{aligned}$$

To implement `cut` in Ferrite, we need a way to split a linear context $\texttt{C} = \Delta_1, \Delta_2$ into two sub-contexts $\texttt{C1} = \Delta_1$ and $\texttt{C2} = \Delta_2$ so that they can be passed to the respective continuations. Moreover, since Ferrite programs use context lenses to access channels, the ordering of channels inside `C1` and `C2` must be preserved. We can preserve the ordering by replacing the corresponding slots with `Empty` during the splitting. Ferrite defines the `SplitContext` trait to implement the splitting as follows:

```
enum L {}     enum R {}
trait SplitContext<C: Context> { type Left: Context; type Right: Context; ... }
```

We first define two (uninhabited) marker types `L` and `R`. We then use type-level lists consisting of elements `L` and `R` to implement the `SplitContext` trait for a given linear context `C`. The `SplitContext` implementation contains the associated types `Left` and `Right`, representing the contexts `C1` and `C2` after splitting. As an example, the type `HList![L, R, L]` would implement `SplitContext<HList![A1, A2, A3]>` for any slot `A1`, `A2` and `A3`, with the associated type `Left` being `HList![A1, Empty, A3]` and `Right` being `HList![Empty, A2, Empty]`. We omit the implementation details of `SplitContext` for brevity. Using `SplitContext`, the function `cut` can be implemented as follows:

```
fn cut<XS, C: Context, C1: Context, C2: Context, C3: Context, A: Protocol, B: Protocol>
  ( cont1: PartialSession<C1, A>,
    cont2: impl FnOnce(C2::Length) -> PartialSession<C3, B> ) -> PartialSession<C, B>
where XS: SplitContext<C, Left=C1, Right=C2>, C2: AppendContext<HList![A], Appended=C3>
```

The function `cut` works by using the heterogeneous list `XS` that implements `SplitContext` to split a linear context `C` into `C1` and `C2`. To pass on the channel `A` that is offered by `cont1` to `cont2`, `cut` uses a similar technique to `receive_channel` to append the channel `A` to the end of `C2`, resulting in `C3`. Using `cut`, we can write `hello_main` in Ferrite as follows:

```
let hello_main: Session<End> = cut::<HList![]>(hello_client, |f| {
  cut::<HList![R]>(hello_provider, |a| { send_channel_to(f, a, forward(f)) }) });
```

Due to ambiguous instances for `SplitContext`, the type parameter `XS` has to be annotated explicitly for Rust to know in which context a channel should be placed. In the first use of `cut`, the context is empty, so we call `cut` with the empty list `HList![]`. We pass `hello_client` as the first continuation to run in parallel, and name the channel offered by `hello_client` as `f`. In the second use of `cut`, the linear context would be `HList![ReceiveValue<String, End>]`, with one channel `f`. We then have `cut` move `f` to the right side using `HList![R]`. On the left continuation, we have `hello_provider` run in parallel, and name the offered channel as `a`. In the right continuation, we use `send_channel_to` to send channel `a` to `f`. Finally, we forward the continuation of `f`, which now has type `End`.

Although `cut` provides the primitive way for Ferrite programs to communicate, its use can be cumbersome and requires a lot of boilerplate. For simplicity, we provide a specialized `apply_channel` construct that abstracts over the common usage pattern of `cut`. `apply_channel` takes a client program `f` offering session type `ReceiveChannel<A, B>` and a provider program `a` offering session type `A`, and sends `a` to `f` using `cut`. The use of `apply_channel` is akin to regular function application, making it more intuitive for programmers to use:

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

4.4 Executing Ferrite Programs

To actually *execute* a Ferrite program, the program must offer some specific session types. In the simplest case, Ferrite provides the function `run_session` for running a top-level Ferrite program offering `End`, with an empty linear context:

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

Function `run_session` executes the session *asynchronously* using Rust's async/await infrastructure. Internally, `PartialSession<C, A>` implements the dynamic semantics of the Ferrite program, which is only accessible by public functions such as `run_session`. Ferrite currently uses the `tokio` [41] runtime for asynchronous execution, as well as the one shot channels from `tokio::sync::oneshot` to implement the low-level communication of Ferrite channels.

Since `run_session` accepts an argument of type `Session<End>`, this means that programmers must first use `cut` or `apply_channel` to fully link Ferrite programs with free channel variables, or Ferrite programs that offer session types other than `End` before they can be executed. This restriction ensures that all linear channels created in a Ferrite program are consumed. For example, the programs `hello_provider` and `hello_client` cannot be executed individually, but the program resulting from composing `hello_provider` with `hello_client` can be executed:

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

We omit the implementation details of the dynamics of Ferrite, which use low-level primitives such as Rust channels while carefully ensuring that the requirements and invariants of session types are satisfied. Interested readers can find more details in our companion technical report [7].

5 Recursive and Shared Session Types

Many real world applications, such as web services and instant messaging, implement protocols that are recursive in nature. As a result, it is essential for Ferrite to support recursive session types. In this section, we report on Rust's limited support for recursive types and how Ferrite addresses this limitation. We then discuss our encoding of *shared*, recursive session types.

5.1 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 write a Ferrite program that offers such a session type, we may attempt to define the counter session type as `type Counter = SendValue<u64, Counter>`. If we try to use such a type definition, the compiler will emit the error "cycle detected when processing `Counter`". The issue with the definition is that it is a directly self-referential type alias, which is not supported in Rust. Rust imposes various restrictions on the legal forms of recursive types to ensure that the memory layout of data is known at compile-time.

Type-Level Fixed Points

To address this limitation, we implement type-level fixed points using *defunctionalization* [36, 46]. This is done by introducing a `RecApp` trait that is implemented by defunctionalized types that can be "applied" with a type parameter:

```
trait RecApp<X> { type Applied; }   type AppRec<F, X> = <F as RecApp<X>>::Applied;
struct Rec<F: RecApp<Rec<F>>> { unfold: Box<AppRec<F, Rec<F>>> }
```

The `RecApp` trait is parameterized by a type `X`, which serves as the type argument to be applied to. This makes it possible for a Rust type `F` that implements `RecApp` to act as if it has the higher-kinded type Type → Type, and be "applied" to type `X`. We define a type alias `AppRec<F, X>` to refer to the associated type `Applied` resulting from "applying" `F` to `X` via `RecApp`. Using `RecApp`, we can now define a type-level recursor `Rec` as a struct parameterized by a type `F` that implements `RecApp<Rec<F>>`. The body of `Rec` contains a boxed value `Box<AppRec<F, RecApp<Rec<F>>>>` to make it have a fixed size in Rust.

Ferrite implements `RecApp` for all `Protocol` types, with the type `Z` used to denote the recursion point. With that, the example `Counter` type would be defined as `type Counter = Rec<SendValue<u64, Z>>`. The type `Rec<SendValue<T, Z>>` is unfolded into `SendValue<T, Rec<SendValue<T, Z>>>` through generic implementations of `RecApp` for `SendValue` and `Z`:

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

Inside `RecApp`, `Z` simply replaces itself with the type argument `X`. `SendValue<T, A>` delegates the type application of `X` to `A`, provided that the session type `A` also implements `RecApp` for `X`.

The session type `Counter` is iso-recursive, as the rolled type `Rec<SendValue<u64, Z>>` and the folded type `SendValue<u64, Rec<SendValue<u64, Z>>` are considered distinct types in Rust. As a result, Ferrite provides the constructs `fix_session` and `unfix_session` for converting between the rolled and unfolded versions of a recursive session type.

Nested Recursive Session Types

The use of `RecApp` is akin to emulating the higher-kinded type (HKT) Type → Type in Rust. As of this writing, HKTs are only available in the nightly (unstable) version of Rust through *generic associated types*. However, even with support for HKTs, our defunctionalization-based approach via `RecApp` allows us to generalize to *nested* recursive types.

To account for a recursive type with multiple recursion points, we introduce a *recursion context* `R` as a type-level list of elements (c.f. the linear context of Section 4.2). The type-level natural numbers `Z`, `S<Z>`, etc. are now used as de Bruijn indices to unfold to the elements in the recursion context. The type-level fixed point combinator `Rec` is redefined as `RecX`, containing the recursion context:

```
struct RecX<R, F: RecApp<(RecX<R, F>, R)>> { unfix: Box<AppRec<F, (RecX<R, F>, R)>> }
type Rec<F> = RecX<(), F>;
impl<R, F: RecApp<(RecX<R, F>, R)>> RecApp<R> for RecX<(), F>  {
    type Applied = RecX<R, F>; }
```

A recursive session type is defined starting with an empty recursion context. Since nested recursive session types allow a `RecX` to be embedded inside another `RecX`, we have `RecX` also implement `RecApp`, provided it has an empty recursion context. When unfolded from another recursion context `R`, `RecX` simply saves `R` as its own recursion context and does not unfold further in `F`. The inner type `F` is only unfolded once with the full recursion context after all surrounding `RecX` types are unfolded.

The recursive marker `Z` is modified to unfold to the first element of the recursion context. We then implement `S<N>` to unfold to the (N+1)-th position in the recursion context:

```
impl<A, R> RecApp<(A, R)> for Z { type Applied = A; }
impl<A, R, N: RecApp<R>> RecApp<(A, R)> for S<N> { type Applied = N::Applied; }
```

5.2 Shared Session Types

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

Shared Session Types in Ferrite

As introduced in Section 2, the SILL_S (and SILL_R) notion of shared session types is recursive in nature, as a shared session type must offer the same linear critical section to all clients that acquire a shared resource. For instance, a shared version of the `Counter` type in SILL_R is:

$$\mathsf{SharedCounter} = \uparrow^\mathsf{S}_\mathsf{L} \mathsf{Int} \triangleleft \downarrow^\mathsf{S}_\mathsf{L} \mathsf{SharedCounter}$$

The linear portion of SharedCounter in between $\uparrow^\mathsf{S}_\mathsf{L}$ (acquire) and $\downarrow^\mathsf{S}_\mathsf{L}$ (release) amounts to a critical section. When a SharedCounter is *acquired*, it offers a linear session type $\mathsf{Int} \triangleleft \downarrow^\mathsf{S}_\mathsf{L}\mathsf{SharedCounter}$, willing to send an integer value, after which it must be *released* to become available again as a SharedCounter to the next client.

The recursive aspect of shared session types in SILL_R means that we can reuse the implementation technique that we use for recursive session types. The type `SharedCounter` can be defined in Ferrite as follows:

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

Compared to linear recursive session types, the main difference is that instead of using `Rec`, a shared session type is defined using the `LinearToShared` construct. This corresponds to $\uparrow^\mathsf{S}_\mathsf{L}$ in SILL_R, with the inner type `SendValue<u64, Release>` corresponding to the linear portion of the shared session type. At the point of recursion, the type `Release` is used in place of $\downarrow^\mathsf{S}_\mathsf{L}\mathsf{SharedCounter}$. As a result, the type `LinearToShared<SendValue<u64, Release>>` is unfolded into `SendValue<u64, SharedToLinear<LinearToShared<SendValue<u64, Release>>>>` after being acquired. Type unfolding is implemented as follows:

```
trait SharedRecApp<X> { type Applied; }   trait SharedProtocol { ... }
struct SharedToLinear<F> { ... }          struct LinearToShared<F> { ... }
impl<F> Protocol for SharedToLinear<LinearToShared<F>>
```

```
  where F: SharedRecApp<SharedToLinear<LinearToShared<F>>> { ... }
impl<F> SharedProtocol for LinearToShared<F>
  where F: SharedRecApp<SharedToLinear<LinearToShared<F>>> { ... }
```

The struct `LinearToShared` is parameterized by a linear session type `F` that implements the trait `SharedRecApp<SharedToLinear<LinearToShared<F>>>`. It uses the `SharedRecApp` trait instead of the `RecApp` trait to ensure that the session type is *strictly equi-synchronizing* [3], 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 `SharedRecApp` for `End`, ruling out invalid shared session types such as `LinearToShared<SendValue<u64, End>>`. We note that the type argument to `F`'s `SharedRecApp` is another struct `SharedToLinear`, which corresponds to $\downarrow^{S}_{L}$ in $\mathsf{SILL}_{\mathsf{R}}$. A `SharedProtocol` trait is also defined to identify 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 use of shared channel:

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

The code above shows the definition of the `SharedChannel` struct. Unlike linear channels, shared channels follow structural typing, i.e. they can be weakened or contracted. This means that we can delegate the handling of shared channels to Rust, given that `SharedChannel` implements Rust's `Clone` trait to allow contraction. Whereas $\mathsf{SILL}_{\mathsf{S}}$ provides explicit constructs for sending and receiving shared channels, Ferrite's shared channels can be sent as regular Rust values using `Send/ReceiveValue`.

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

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 is itself affine and cannot be shared. To enable sharing, the shared Ferrite program must first be executed with `run_shared_session`. The function `run_shared_session` takes a shared Ferrite program of type `SharedSession<S>` and starts 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 sent to multiple clients for access to the shared process.

Below we demonstrate how a shared session can be defined and used by multiple clients:

```
type SharedCounter = LinearToShared<SendValue<u64, Release>>;
fn counter_producer(current_count: u64) -> SharedSession<SharedCounter> {
  accept_shared_session(async move {
    send_value(current_count, detach_shared_session(
        counter_producer(current_count + 1))) }) }
fn counter_client(counter: SharedChannel<SharedCounter>) -> Session<End> {
  acquire_shared_session(counter, move | chan | {
    receive_value_from(chan, move | count | { println!("received count: {}", count);
      release_shared_session(chan, terminate()) }) }) }
```

The recursive function `counter_producer` creates a `SharedSession` program that, when executed, offers a shared channel of session type `SharedCounter`. On the provider side, a shared session is defined using the `accept_shared_session` construct, with a continuation given as an async thunk that is executed when a client acquires the shared session and enters

the linear critical section (of type `SendValue<u64, SharedToLinear<SharedCounter>>`). Inside the closure, the producer uses `send_value` to send the current count to the client and then uses `detach_shared_session` to exit the linear critical section. The construct `detach_shared_session` offers the linear session type `SharedToLinear<SharedCounter>` and expects a continuation that offers the shared session type `SharedCounter` to serve the next client. We generate the continuation by recursively calling the `counter_producer` function.

The `counter_client` function takes a shared channel of session type `SharedCounter` and returns a session type program that acquires the shared channel and prints the received count value to the terminal. A linear Ferrite program can acquire a shared session using the `acquire_shared_session` construct, which accepts a `SharedChannel` object and adds the acquired linear channel to the linear context. In this case, the continuation closure is given the context lens `Z`, which provides access to the linear channel of session type `SendValue <u64, SharedToLinear<SharedCounter>>` in the first slot of the linear context. It then uses `receive_value_from` to receive the value sent by the shared provider and then prints the value. On the client side, the linear session of type `SharedToLinear<SharedCounter>` must be released using the `release_shared_session` construct. After releasing the shared session, other clients will then be able to acquire the shared session.

```
async fn main () {
  let counter1: SharedChannel<SharedCounter> = run_shared_session(counter_producer(0));
  let counter2 = counter1.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; }
```

To illustrate a use of `SharedCounter`, we have a `main` function that initializes a shared producer with an initial value of 0 and then runs the shared provider using the `run_shared_session` construct. The returned `SharedChannel` is then cloned, making the shared counter accessible via aliases `counter1` and `counter2`. It then uses `task::spawn` to spawn two async tasks that run `counter_client` twice. A key observation is that multiple Ferrite programs that are executed independently can access *the same* shared producer through a reference to the shared channel.

A follow up example of `SharedQueue`, which demonstrates the Ferrite implementation of the SILL_S shared queue example in Section 2 is available in our companion technical report [7].

6 Choice

Session types support *internal* and *external* choice, leaving the choice among several options to the provider or the client, resp. (see Table 2). When restricted to binary choice, the implementation is relatively straightforward, as shown below by the two right rules for internal choice in SILL_R. The offer_left and offer_right constructs allow a provider to offer an internal choice $A \oplus B$ by offering either A or B, resp.

$$\frac{\Gamma\,;\,\Delta \vdash K :: A}{\Gamma\,;\,\Delta \vdash \mathsf{offer\_left};\, K :: A \oplus B}\ (\mathrm{T}\oplus 2\mathsf{L}_\mathsf{R}) \qquad \frac{\Gamma\,;\,\Delta \vdash K :: B}{\Gamma\,;\,\Delta \vdash \mathsf{offer\_right};\, K :: A \oplus B}\ (\mathrm{T}\oplus 2\mathsf{R}_\mathsf{R})$$

It is straightforward to implement the two versions of the right rules by writing the two respective functions `offer_left` and `offer_right`:

```
fn offer_left<C: Context, A: Protocol, B: Protocol>
  ( cont: PartialSession<C, A> ) -> PartialSession<C, InternalChoice2<A, B>>
fn offer_right < C: Context, A: Protocol, B: Protocol >
  ( cont: PartialSession<C, B> ) -> PartialSession<C, InternalChoice2<A, B>>
```

However, this approach does not scale if we want to generalize choice beyond two options. To support n-ary choice, the functions would have to be explicitly reimplemented N times. Instead, we implement a single `offer_case` function which allows selection from n-ary branches.

In Section 4.2, we explored heterogeneous lists to encode the linear context, i.e. *products* of session types of arbitrary lengths. We then implemented context *lenses* to access and update individual channels in the linear context. Observing that n-ary choices can be encoded as *sums* of session types, we now use *prisms* to implement the selection of an arbitrary-length branch. Ferrite also supports an n-ary choice type `InternalChoice<HList![...]>`, with `InternalChoice<HList![A, B]>` being the special case of a binary choice. To select a branch out of the heterogeneous list, we define the `Prism` trait as follows:

```
trait Prism<Row> {type Elem; ...}  impl<A, R> Prism<(A, R)> for Z {type Elem = A; ... };
impl<N, A, R> Prism<(A, R)> for S<N> where N: Prism<R> { type Elem = N::Elem; ... }
```

The `Prism` trait is parameterized over a row type `Row=HList![...]`, with the associated type `Elem` being the element type that has been selected from the list by the prism. We then inductively implement `Prism` using type-level natural numbers, with the number `N` used for selecting the N-th element of the heterogeneous list. The definition of `Prism` is similar to `ContextLens`, with the main difference being that we only need `Prism` to support extraction and injections operations on the sum types that are derived from the heterogeneous list. Using `Prism`, a generalized `offer_case` function is implemented as follows:

```
fn offer_case<C: Context, A: Protocol, Row, N: Prism<Row, Elem=A>>
  (n: N, cont: PartialSession<C, A>) -> PartialSession<C, InternalChoice<Row>>
```

The function accepts a natural number `N` as the first parameter, which acts as the *prism* for selecting a session type A_N out of the row type `Row=HList![..., A_N, ...]`. Through the associated type `A=N::Elem`, `offer_case` forces the programmer to provide a continuation that offers the chosen session type `A`.

While `offer_case` is a step in the right direction, it only allows the selection of a specific choice, but not the provision of *all* possible choices. The latter, however, is necessary to encode the SILL_R left rule of internal choice and right rule of external choice. To illustrate the problem, let's consider the right rule of a binary external choice, $\mathrm{T}\&2_\mathsf{R}$:

$$\frac{\Gamma\,;\,\Delta \vdash K_l :: A \qquad \Gamma\,;\,\Delta \vdash K_r :: B}{\Gamma\,;\,\Delta \vdash \mathsf{offer\_choice\_2}\ K_l\ K_r :: A\&B}\ (\mathrm{T}\&2_\mathsf{R})$$

The `offer_choice_2` construct has two possible continuations K_l and K_r, with only one of them being executed, depending on the selection by the client. In a naive implementation, we can define the construct to accept two continuations as follows:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont_left: PartialSession<C, A>, cont_right: PartialSession<C, B> )
  -> PartialSession<C, ExternalChoice2<A, B>>
```

While the above implementation works in most languages, it is not adequate in Rust. Since Rust's type system is *affine*, variables can only be captured by one of the continuation closures, but not both. As far as the compiler is aware, both closures can potentially be called, and we cannot state that one of the branches is guaranteed to never run.

In order for `offer_choice_2` to work in Rust's affine typing, it has to accept only one continuation closure and have it return either `PartialSession<C, A>` or `PartialSession<C, B>`, depending on the client's selection. It is not as straightforward to express such behavior as a valid type in a language like Rust. If Rust supported dependent types, `offer_choice_2` could be implemented along the following lines:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont: impl FnOnce(first: bool) -> if first { PartialSession<C, A> }
     else { PartialSession<C, B> } ) -> PartialSession<C, ExternalChoice2<A, B>>
```

That is, the return type of the `cont` closure depends on the whether the *value* of the `first` argument is true or false. However, since Rust does not support dependent types, we emulate a dependent sum in a non-dependent language, using a CPS transformation:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont: impl FnOnce(InjectSum2<C, A, B>) -> ContSum2<C, A, B> )
    -> PartialSession<C, ExternalChoice2<A, B>>
```

The function `offer_choice_2` accepts a continuation function `cont` that is given a value of type `InjectSum2<C, A, B>` and returns a value of type `ContSum2<C, A, B>`. We will now look at the definitions of `ContSum2` and `InjectSum2`. First, we observe that the different return types for the two branches can be unified with a type `ContSum2`:

```
struct ContSum2<C: Context, A: Protocol, B: Protocol> { ... }
async fn run_cont_sum<C: Context, A: Protocol, B: Protocol>(cont: ContSum2<C, A, B>)
```

The type `ContSum2` contains the necessary data for executing either a `PartialSession<C, A>` or a `PartialSession<C, B>`, together with the runtime data for the linear context `C`. For brevity, the implementation details of `ContSum2` are omitted, with the private function `run_cont_sum` provided as an abstraction for Ferrite to execute the continuation.

We then define `InjectSum2` as a sum of boxed closures that would construct a `ContSum2` from either a `PartialSession<C, A>` or a `PartialSession<C, B>`:

```
enum InjectSum2<C, A, B> {
  InjectLeft(Box<dyn FnOnce(PartialSession<C, A>) -> ContSum2<C, A, B>>),
  InjectRight(Box<dyn FnOnce(PartialSession<C, B>) -> ContSum2<C, A, B>>) }
```

When the `cont` passed to `offer_choice_2` is given a value of type `InjectSum2<C, A, B>`, it has to branch on it and match on whether the `InjectLeft` or `InjectRight` constructors are used. Since the return type of `cont` is `ContSum2<C, A, B>` and the constructor for `ContSum2` is private, there is no other way for `cont` to construct the return value other than to call either `InjectLeft` or `InjectRight` with the appropriate continuation.

The use of `InjectSum2` prevents the programmer from providing the wrong branch in the continuation by keeping the constructor private. However, a private constructor alone cannot prevent two uses of `InjectSum2` to be *deliberately* interchanged, causing a protocol violation. To fully ensure that there is no way for the user to provide a `ContSum2` from elsewhere, we instead use a technique from GhostCell [47] that uses *higher-ranked trait bounds* (HTRB) to mark a phantom invariant lifetime on both `InjectSum2` and `ContSum2`:

```
fn offer_choice_2<C: Context, A: Protocol, B: Protocol>
  ( cont: for <'r> impl FnOnce(InjectSum2<'r, C, A, B>) -> ContSum2<'r, C, A, B> )
    -> PartialSession<C, ExternalChoice2<A, B>>
```

The use of HRTB ensures that each call of `offer_choice_2` would generate a unique lifetime `'r` for the continuation. Using that, Ferrite can ensure that a value of type `InjectSum2<'r1, C, A, B>` cannot be used to construct the return value of type `ContSum2<'r2, C, A, B>`, if the lifetimes `<'r1>` and `<'r2>` are different. An example use of `offer_choice_2` is as follows:

```
let choice_provider: Session<ExternalChoice2<SendValue<u64, End>, End>>
  = offer_choice_2(|b| { match b { InjectLeft(ret) => ret(send_value(42, terminate())),
                                   InjectRight(ret) => ret(terminate()) } });
```

To free the programmer from writing such boilerplate, Ferrite also provides macros that translates into the underlying pattern matching syntax. The macros allow the same example to be written as follows:

```
1  enum CanvasMsg { Canvas2d(Canvas2dMsg, CanvasId), Close(CanvasId), ... }
2  enum Canvas2dMsg { LineTo(Point2D), GetTransform(Sender<Transform2D>),
3    IsPointInPath(f64, f64, FillRule, IpcSender<bool>), ... }
4  enum ConstellationCanvasMsg { Create { id_sender: Sender<CanvasId>, size: Size2D } }
5  struct CanvasPaintThread { canvases: HashMap<CanvasId, CanvasData>, ... }
6  impl CanvasPaintThread { ...
7    fn start() -> (Sender<ConstellationCanvasMsg>, Sender<CanvasMsg>) {
8      let (msg_sender, msg_receiver) = channel();  let (create_sender, create_receiver) = channel();
9      thread::spawn(move || { loop { select! {
10         recv(canvas_msg_receiver) -> { ...
11           CanvasMsg::Canvas2d(message, canvas_id) => { ...
12             Canvas2dMsg::LineTo(point) => self.canvas(canvas_id).move_to(point),
13             Canvas2dMsg::GetTransform(sender) =>
14               sender.send(self.canvas(canvas_id).get_transform()).unwrap(), ... }
15           CanvasMsg::Close(canvas_id) => canvas_paint_thread.canvases.remove(&canvas_id) }
16         recv(create_receiver) -> { ... ConstellationCanvasMsg::Create { id_sender, size } => {
17             let canvas_id = ...; self.canvases.insert(canvas_id, CanvasData::new(size, ...));
18             id_sender.send(canvas_id); } } } } });
19     (create_sender, msg_sender) }
20   fn canvas(&mut self, canvas_id: CanvasId) -> &mut CanvasData {
21     self.canvases.get_mut(&canvas_id).expect("Bogus canvas id") } }
```

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

```
define_choice!{ CustomChoice; Left: SendValue<u64, End>, Right: End }
let choice_provider: Session<ExternalChoice<CustomChoice>> = offer_choice! {
  Left => send_value(42, terminate()), Right => terminate() };
```

The `define_choice!` macro allows defining named n-ary branches of choice. The `offer_choice!` macro allows the choice provider to branch without the boilerplate used in the earlier example. The generalization from binary to n-ary choice is omitted for conciseness. The details can be found in our companion technical report [7].

7 Evaluation

The Ferrite library is more than just a research prototype. It is designed for practical use in real world applications. To evaluate the design and implementation of Ferrite, we re-implemented the communication layer of the canvas component of Servo [29] entirely in Ferrite. Servo is an under development browser engine that uses message-passing for heavy task parallelization. Canvas provides 2D graphic rendering, allowing clients to create new canvases and perform operations on a canvas such as moving the cursor and drawing shapes.

The canvas component is a good target for evaluation as it is sufficiently complex and also very demanding in terms of performance. Canvas is commonly used for animations in web applications. For an animation to look smooth, a canvas must render at least 24 frames per second, with potentially thousands of operations to be executed per frame.

The changes we made are fairly minimal, consisting of roughly 750 lines of additions and 620 lines of deletions, out of roughly 300,000 lines of Rust code in Servo. The sources of our implementation are provided as an artifact. To differentiate the two versions of code snippets, we use blue for the original code, and green for the code using Ferrite.

7.1 Servo Canvas Component

Figure 1 provides a sketch of the main communication paths in Servo's canvas component [30]. The canvas component is implemented by the `CanvasPaintThread`, whose function `start` contains the main communication loop running in a separate thread (lines 9–18). This loop processes client requests received along `canvas_msg_receiver` and `create_receiver`, which are the receiving endpoints of the channels created prior to spawning the loop (lines 8–8). 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, resp. When a client sends a message that expects a response from the recipient, such as `GetTransform` and `IsPointInPath` (lines 2–3), it sends a channel along with the message to be used by the recipient to send back the result. Canvases are identified by an id, which is generated upon canvas creation (line 17) and stored in the thread's `canvases` hash map (line 5). If a client requests an invalid id, for example after prior termination and removal of the canvas (line 15), the failed assertion `expect("Bogus canvas id")` (line 21) will result in a `panic!`, causing the canvas component to crash and subsequent calls to fail.

The 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. Nonetheless, Rust's type system is not expressive enough to enforce 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 "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. In fact, while refactoring Servo to use Ferrite, we were able to uncover a protocol violation in Servo, caused by one of the nested match arms of the provider doing an early return before sending back any result to the client.

7.2 Canvas Protocol in Ferrite

In the original canvas component, the provider `CanvasPaintThread` accepts messages of type `CanvasMsg`, made up of a combination of smaller sub-message types such as `Canvas2dMsg`. We note that the majority of the sub-message types have the following trivial form:

```
enum CanvasMsg { Canvas2d(Canvas2dMsg, CanvasId), Close(CanvasId), ... }
enum Canvas2dMsg { BeginPath, ClosePath, Fill(FillOrStrokeStyle), ... }
```

The trivial sub-message types such as `BeginPath`, `Fill`, and `LineTo` do not require a response from the provider, so the client can simply fire them and proceed. Although we can offer all sub-message types as separate branches in an external choice, it is more efficient to keep trivial sub-messages in a single enum. In our implementation, we define `CanvasMessage` to have similar sub-messages as `Canvas2dMsg`, with non-trivial messages such as `IsPointInPath` moved to separate branches.

```
enum CanvasMessage { BeginPath, ClosePath, Fill(FillOrStrokeStyle), ... }
define_choice! { CanvasOps; Message: ReceiveValue<CanvasMessage, Release>, ... }
type Canvas = LinearToShared<ExternalChoice<CanvasOps>>;
```

We use the `define_choice!` macro described in Section 6 to define an n-ary choice `CanvasOps`. The first branch of `CanvasOps` is labelled `Message`, and the only action is for the provider to receive a `CanvasMessage`. The choices are offered as an external choice, and the session type `CanvasProtocol` is defined as a shared protocol that offers the choices in the critical section.

The original design of the `CanvasPaintThread` would be sufficient if the only messages being sent were trivial messages. However, `Canvas2dMsg` also contains non-trivial sub-messages, such as `GetImageData` and `IsPointInPath`, demanding a response from the provider:

```
enum Canvas2dMsg { ..., GetImageData(Rect<u64>, Size2D<u64>, IpcBytesSender),
  IsPointInPath(f64, f64, FillRule, IpcSender<bool>), ... }
```

To obtain the result from the original canvas, clients must create a new inter-process communication (IPC) channel and bundle the channel's sender endpoint with the message. In our implementation, we define separate branches in `CanvasOps` to handle non-trivial cases:

```
define_choice! { CanvasOps; Message: ReceiveValue<CanvasMessage, Release>,
  GetImageData: ReceiveValue<(Rect<u64>, Size2D<u64>), SendValue<ByteBuf, Release>>,
  IsPointInPath: ReceiveValue<(f64, f64, FillRule), SendValue<bool, Release>>, ... }
```

Table 4 MotionMark Benchmark scores in fps (higher is better).

Benchmark Name	Servo	Servo/Ferrite	Firefox	Chrome
Arcs	12.21 ± 6.75%	11.83 ± 11.49%	52.61 ± 32.88%	46.00 ± 9.00%
Paths	43.76 ± 10.66%	40.98 ± 18.94%	55.59 ± 28.80%	59.50 ± 14.90%
Lines	7.48 ± 7.06%	11.47 ± 12.74%	14.35 ± 6.65%	32.43 ± 6.48%
Bouncing clipped rects	18.43 ± 7.06%	18.23 ± 11.00%	34.82 ± 7.76%	58.07 ± 19.85%
Bouncing gradient circles	8.02 ± 7.74%	7.72 ± 12.63%	58.79 ± 21.03%	59.77 ± 10.07%
Bouncing PNG images	7.97 ± 5.91%	6.31 ± 10.26%	24.61 ± 6.35%	59.94 ± 13.04%
Stroke shapes	10.60 ± 3.95%	10.35 ± 10.96%	51.21 ± 11.25%	59.38 ± 16.87%
Put/get image data	60.01 ± 3.81%	32.08 ± 10.83%	59.66 ± 20.16%	60.00 ± 5.00%

The original `GetImageData` accepts an `IpcBytesSender`, which sends raw bytes back to the client. In Ferrite, we translate the use of `IpcBytesSender` to the type `SendValue<ByteBuf, Z>`, which sends the raw bytes wrapped in a `ByteBuf` type.

Aside from the `Canvas` protocol, we also redesign the use of `ConstellationCanvasMsg` into its own shared protocol, `ConstellationCanvas`:

```
type ConstellationCanvas = LinearToShared<ReceiveValue<Size2D,
    SendValue<SharedChannel<Canvas>, Release>>>;
```

To create a new canvas, a client first acquires the shared channel of type `SharedChannel<ConstellationCanvas>`. Afterwards, the client sends the `Size2D` parameter to specify the canvas size. The constellation canvas provider then spawns a new canvas shared process through `run_shared_session` and sends back the shared channel of type `SharedChannel<Canvas>` as a value. Finally, the session is released, allowing other clients to acquire the shared provider.

7.3 Performance Evaluation

To evaluate the performance of the canvas component, we use the MotionMark benchmark suite [45]. MotionMark is a web benchmark that focuses on graphics performance of web browsers. It contains benchmarks for various web components, including canvas, CSS, and SVG. As MotionMark does not yet support Servo, we modified the benchmark code to make it work in the absence of features that are not implemented in Servo (details on the benchmarks can be found in the companion artifact).

For the purpose of this evaluation, we focused on benchmarks that target the canvas component and skipped benchmarks that fail in Servo due to missing features. We ran each benchmark in a fixed 1600x800 resolution for 30 seconds, on a Core i7 Linux desktop machine. We ran the benchmarks against the original Servo, modified Servo with Ferrite canvas (Servo/Ferrite), Firefox (v98), and Chrome (v99). Our performance scores are measured in the fixed mode version of MotionMark, which measures frames per second (fps) performance of executing the same set of canvas operations per frame.

The benchmark results are shown in Table 4, with the performance scores in fps (higher fps is better). It is worth noting that a benchmark can achieve at most 60 fps. Our goal in this benchmark is to keep the scores of Servo/Ferrite close to those of Servo, *not* to achieve better performance than the original. This is shown to be the case in most of the benchmarks.

The only benchmark with a large difference between Servo and Servo/Ferrite is *Put/get image data*, with Ferrite performing 2x worse. This is because in Servo/Ferrite, we use `ByteBuf` to transfer the images as raw bytes within the same shared channel. Servo uses a specialized structure `IpcBytesSender` for transferring raw bytes in parallel to other messages. As a result, communication in Servo/Ferrite is congested during the transfer of the image data, while the original Servo can process new messages in parallel with the image transmission.

We also observe that there are significant performance differences in the scores between Servo and those in Firefox and Chrome, indicating that there exist performance bottlenecks in Servo unrelated to communication protocols.

8 Related and Future Work

Session type embeddings exist for various languages, including Haskell [34, 20, 27, 31], OCaml [32, 19], Java [18, 17], and Scala [39]. Functional languages like ML, OCaml, and Haskell, in particular, are ideal host languages for creating EDSLs thanks to their advanced features (e.g. type classes, type families, higher-rank and higher-kinded types and GADTs). [34] first demonstrated the feasibility of embedding session types in Haskell, with refinements done in later works [20, 27, 31]. Similar embeddings have also been contributed in the context of OCaml by `FuSe` [32] and `session-ocaml` [19].

Aside from Ferrite, there are other implementations of session types in Rust, including `session_types` [21], `sesh` [25], and `rumpsteak` [8, 9]. `session_types` were the first implementation to make use of affinity to provide a session type library in Rust. `sesh` emphasizes this aspect by embedding the affine session type system Exceptional GV [12] in Rust. Both `session_types` and `sesh` adopt a classical perspective, requiring the endpoints of a channel to be typed with dual types. `rumpsteak` develops an embedding of multiparty session types by generating Rust types derived from multiparty session types defined in Scribble [48].

Due to their reliance on Rust's affine type system, neither `session_types` nor `sesh` prevent a channel endpoint from being dropped prematurely, relegating the handling of such errors to the runtime. `rumpsteak` uses some type-level techniques similar to Ferrite to enforce a channel's linear usage in the continuation passed to the `try_session` function. This ensures that a linear channel in `rumpsteak` is always fully consumed, if it is ever consumed. However, prior to the call to `try_session`, the linear channel exists as an affine value, which may be dropped by without being consumed at all, resulting in a deadlock. Ferrite enforces linearity *at all levels*, including safe linking of multiple linear processes using `cut`.

In terms of concurrency, `session_types`, `sesh`, and `rumpsteak` all require the programmer to manually manage concurrency, either by spawning threads or async tasks. This introduces potential failure when the code fails follow the requirement to spawn all processes. On the other hand, the simplicity of such a model allows relatively few threads or async tasks to be spawned, thereby allowing the underlying runtime to execute the processes more efficiently. In comparison, Ferrite offers fully managed concurrency, without the programmer having to worry about how to spawn the processes and execute them in parallel.

In terms of performance, the downside of Ferrite's concurrency approach is that it aggressively spawns new async tasks in each use of `cut`. Although async tasks in Rust are much more lightweight than OS threads, there is still a significant overhead in spawning and managing many async tasks, especially in micro-benchmarks. As a result, Ferrite tends to perform slower than alternative Rust implementations in settings where only a fixed small number of processes need to be spawned. Nevertheless, it is worth noting that the async ecosystem in Rust is still relatively immature, with many potential improvements to be made. In practice, the overhead of the async runtime may also be negligible when compared to the core application logic. In such cases, Ferrite would also allow applications to scale more easily by allowing many more processes to be spawned and managed concurrently without requiring additional effort from the programmer.

In terms of DSL design, Ferrite is closely related to the embeddings in OCaml and Haskell, as it fully enforces a linear treatment of channels and thus *statically* rules out any panics arising from dropping a channel prematurely. However, Ferrite leverages Rust's affine type

system, which naturally extends to support linear types as compared to the structural type systems of OCaml or Haskell. As a result, Ferrite programs can reuse any existing Rust code without sacrificing the benefit of affine types. This is generally not the case with substructural EDSLs, which often require rewriting of libraries (e.g. LinearHaskell's `linear-base`).

Ferrite also differs from other libraries in that it adopts intuitionistic typing [4], allowing the typing of a channel rather than its two endpoints via type duality. While the use of dual types is convenient for simple types like `ReceiveValue<String, End>`, the mental overhead of computing the dual type becomes higher when higher-order channels are involved. For example, when implementing a process with type `ReceiveChannel<ReceiveValue<String, End>>`, the programmer would have to keep in mind that the received channel would have its session type flipped and become `SendValue<String, End>`. From an ergonomics point of view, we believe that intuitionistic session types provide a more familiar model of programming.

On the use of profunctor optics, our work is the first to connect n-ary choice to prisms, while prior work by `session-ocaml` [20] has only established the connection between lenses, the dual of prisms, and linear contexts. `FuSe` [32] and `session-ocaml` [19] have previously explored the use of n-ary (generalized) choice through extensible variants available only in OCaml. Our work demonstrates that it is possible to encode extensible variants, and thus n-ary choice, as type-level constructs using features available in Rust.

A major difference in terms of implementation is that Ferrite uses a continuation-passing style, whereas Haskell and OCaml embeddings commonly use (indexed) monads and do-notation. This technical difference amounts to a key conceptual one: a direct correspondence between the Rust programs generated from Ferrite constructs and the SILL_R typing derivation. As a result, the generated Rust code can be viewed as carrying the proof of protocol adherence.

The embeddings of `ESJ` [17] and `lchannels` [39] also adopt a continuation-passing style, but do not faithfully embed typing derivations (i.e. they do not statically enforce linearity). They follow an encoding of session types using linear types [10] first proposed by Kobayashi [24] in the setting of π-calculus. While session types are generally less powerful than the approaches of Kobayashi et al., they provide a useful compromise between expressiveness and simplicity, being more amenable to embeddings in general-purpose language constructs and type systems.

In terms of expressiveness, Ferrite contributes over all prior session-based works in its support for shared session types [1], allowing it to express real-world protocols, as demonstrated by our implementation of Servo's canvas component. Shared session types reclaim the expressiveness of the untyped asynchronous π-calculus in session-typed languages [2], at the cost of deadlock-freedom. Recent extensions of classical linear logic session types contribute another approach to softening the rigidity of linear session types to support multiple client sessions and nondeterminism [35] and memory cells and nondeterministic updates [37], resp.

Our technique of a judgmental embedding opens up new possibilities for embedding type systems other than session types in Rust. Although we have demonstrated that the judgmental embedding is sufficiently powerful to encode a type system like session types, the embedding is currently *shallow*, with the implementation hardcoded to use the channels and async run-time from `tokio`. Rust comes with unique features such as affine types and lifetimes that makes it especially suited for implementing concurrency primitives, as evidenced by the wealth of channel and async run-time implementations available. One of our future goals is to explore the possibility of making Ferrite a *deep* embedding of session types in Rust, so that users can choose from multiple low-level implementations. Although deep embeddings have extensively been explored for languages like Haskell [40, 27], it remains a open question to find suitable approaches that work well in Rust.

References

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

2 Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. A universal session type for untyped asynchronous communication. In *29th International Conference on Concurrency Theory (CONCUR)*, LIPIcs, pages 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

3 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *28th European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019. doi:10.1007/978-3-030-17184-1_22.

4 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR)*, pages 222–236. Springer, 2010.

5 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016.

6 Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite project website. https://github.com/ferrite-rs/ferrite.

7 Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: A judgmental embedding of session types in rust. *CoRR*, abs/2009.13619, 2022. arXiv:2009.13619.

8 Zak Cutner and Nobuko Yoshida. Safe session-based asynchronous coordination in rust. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 80–89. Springer, 2021. doi:10.1007/978-3-030-78142-2_5.

9 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. *CoRR*, abs/2112.12693, 2021. arXiv:2112.12693.

10 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *Principles and Practice of Declarative Programming (PPDP)*, pages 139–150, 2012.

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

12 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *Proceedings of the ACM on Programming Languages*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.

13 Andrew Gerrand. The go blog: Share memory by communicating, 2010. URL: https://blog.golang.org/share-memory-by-communicating.

14 Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR)*, pages 509–523. Springer, 1993.

15 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP)*, pages 122–138. Springer, 1998.

16 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

17 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *24th European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010. doi:10.1007/978-3-642-14107-2_16.

18 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *22nd European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008. doi:10.1007/978-3-540-70592-5_22.

19 Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: a session-based library with polarities and lenses. *Science of Computer Programming*, 172:135–159, 2019. doi:10.1016/j.scico.2018.08.005.

20 Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. 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*, volume 69 of *EPTCS*, pages 74–91, 2010. doi:10.4204/EPTCS.69.6.

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

22 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018. doi:10.1145/3158154.

23 Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In Henrik Nilsson, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004, Snowbird, UT, USA, September 22-22, 2004*, pages 96–107. ACM, 2004. doi:10.1145/1017472.1017488.

24 Naoki Kobayashi. Type systems for concurrent programs. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*, volume 2757 of *Lecture Notes in Computer Science*, pages 439–453. Springer, 2002. doi:10.1007/978-3-540-40007-3_26.

25 Wen Kokke. Rusty variation: Deadlock-free sessions with failure in rust. In *12th Interaction and Concurrency Experience, ICE 2019*, pages 48–60, 2019.

26 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *24th European Symposium on Programming (ESOP)*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584, 2015. doi:10.1007/978-3-662-46669-8_23.

27 Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *9th International Symposium on Haskell*, pages 133–145. ACM, 2016. doi:10.1145/2976002.2976018.

28 J. Garrett Morris. Variations on variants. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 71–81. ACM, 2015. doi:10.1145/2804302.2804320.

29 Mozilla. Servo, the Parallel Browser Engine Project. https://servo.org/, 2012.

30 Mozilla. Servo source code – canvas paint thread, 2021. URL: https://github.com/servo/servo/blob/d13a9355b8e66323e666dde7e82ced7762827d93/components/canvas/canvas_paint_thread.rs.

31 Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 568–581. ACM, 2016. doi:10.1145/2837614.2837634.

32 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017. doi:10.1017/S0956796816000289.

33 Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *Programming Journal*, 1(2):7, 2017. doi:10.22152/programming-journal.org/2017/1/7.

34 Riccardo Pucella and Jesse A. Tov. Haskell session types with (almost) no class. In *1st ACM SIGPLAN Symposium on Haskell*, pages 25–36. ACM, 2008. doi:10.1145/1411286.1411290.

35 Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *CoRR*, abs/2010.13926, 2020. arXiv:2010.13926.

36 John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, volume 2, pages 717–740. ACM, 1972. doi:10.1145/800194.805852.

37 Pedro Rocha and Luís Caires. Propositions-as-types and shared state. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.

38 Matthew Sackman and Susan Eisenbach. Session types in haskell: Updating message passing for the 21st century. Technical report, Imperial College, 2008. URL: http://hdl.handle.net/10044/1/5918.

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

40 Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2012. doi:10.1007/978-3-642-40447-4_2.

41 Tokio. Tokio Homepage. https://tokio.rs/, 2021.

42 Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation.* PhD thesis, Carnegie Mellon University and New University of Lisbon, 2015.

43 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: a monadic integration. In *22nd European Symposium on Programming (ESOP)*, pages 350–369. Springer, 2013. doi:10.1007/978-3-642-37036-6_20.

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

45 WebKit. MotionMark Homepage. https://browserbench.org/MotionMark/, 2021.

46 Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 119–135, 2014. doi:10.1007/978-3-319-07151-0_8.

47 Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. Ghostcell: separating permissions from data in rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473597.

48 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.