Ferrite: A Judgmental Embedding of Session Types in Rust

Ruo Fei Chen
Independent Researcher

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 SILLs, 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.

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-kindred 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 SILLS together with their operational meaning. Subscripts L and S denote linear and shared sessions, resp., where \( m, n \in \{ L, S \} \).

<table>
<thead>
<tr>
<th>Session type current</th>
<th>Process term current</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L )</td>
<td>( P )</td>
<td>provider sends label ( l_A ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( Q )</td>
<td>client receives label ( l_A ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( P )</td>
<td>provider receives label ( l_A ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( Q )</td>
<td>client sends label ( l_A ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( P )</td>
<td>provider sends channel ( d_{m,A} ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( Q )</td>
<td>client receives channel ( d_{m,A} ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( P )</td>
<td>provider sends channel ( d_{m,A} ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( Q )</td>
<td>client sends channel ( d_{m,A} ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( P )</td>
<td>provider sends channel ( d_{m,A} ) along ( c )</td>
</tr>
<tr>
<td>( L )</td>
<td>( Q )</td>
<td>client sends channel ( d_{m,A} ) along ( c )</td>
</tr>
</tbody>
</table>

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 SILLS [1], which Ferrite builds upon. We consider the protocol governing the interaction between a queue and its client:

\[
\text{queue A} = \& \{ \text{enq : } A \rightarrow \text{queue A}, \text{deq} : \oplus \{ \text{none} : 1, \text{some} : A \odot \text{queue A} \} \}
\]

Table 1 provides an overview of the types used in the example. Since SILLS is based on a Curry-Howard correspondence between intuitionistic linear logic and the session-typed \( \pi \)-calculus [4, 5] it uses linear logic connectives (\( \oplus, \&\), \( \odot\), \( \rightarrow\), \( 1\)) 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 \( \text{queue A} \) imposes the following recursive protocol: A client may either send the label \( \text{enq} \) or \( \text{deq} \) 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 (\( \text{none} \)) or not (\( \text{some} \)), 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 \( \text{queue A} \) is limited to a single client. To support safe sharing of
stateful channel references while upholding session fidelity. SILL₅ extends linear session types with shared session types (L₅₁Aₛ, L₅₁Aₘ). 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 L₅₁Aₛ 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 Aₛ. A type L₅₁Aₘ, 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 Aₘ.

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

```plaintext
squeue Aₛ = squeue Aₛ \& \{ \text{enq} : Aₛ \rightarrow \text{squeue Aₛ}, \text{deq} : \oplus \{ \text{none} : \text{squeue Aₛ}, \text{some} : Aₘ \otimes \text{squeue Aₛ} \} \}
```

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, SILL₅ allows the connectives \& and \(\rightarrow\) to be used to transport both linear and shared channels along a linear carrier channel.

To provide a flavor of session-typed programming in SILL₅, 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 \(\rightarrow\)) and the type of the providing channel variable (right of \(\rightarrow\)). The second line binds the channel variables. In SILL₅, \(\leftarrow\) 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 squeue Aₛ.

```plaintext
\begin{align*}
\text{\texttt{\_\_ empty \:: q : squeue Aₛ}} & \quad x : Aₛ, t : squeue Aₛ \rightarrow elem \:: q : squeue Aₛ \\
q \leftarrow \text{empty} & \quad q \leftarrow \text{elem} \leftarrow x, t \\
q' \leftarrow \text{accept q} & \quad q' \leftarrow \text{accept q} \\
\text{case q' of} & \\
| \text{enq} \rightarrow x \leftarrow \text{recv q'} ; & \quad | \text{enq} \rightarrow y \leftarrow \text{recv q'} ; \\
q \leftarrow \text{detach q'} & \quad t' \leftarrow \text{acquire t} ; \\
q' \leftarrow \text{empty} ; q \leftarrow \text{elem} \leftarrow x, e & \quad t' \leftarrow \text{send t'} y ; \\
| \text{deq} \rightarrow q'.\text{none} ; & \quad t \leftarrow \text{release t'}, q \leftarrow \text{detach q'} ; \\
q \leftarrow \text{detach q'} & \quad q \leftarrow \text{elem} \leftarrow x, t \\
q \leftarrow \text{empty} & \quad | \text{deq} \rightarrow q'.\text{some} ; q' \leftarrow \text{send q'} x ; \\
& \quad q \leftarrow \text{detach q'} ; qd \leftarrow q t
\end{align*}
```

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 SILL₅ 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 squeue Aₛ is strictly equi-synchronizing whereas the type invalid = squeue Aₛ \& \{ left : \text{squeue Aₛ} \oplus \{ \text{yes} : \text{squeue Aₛ}, \text{no} : 1 \}, right : \text{squeue Aₛ} \} is not.

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

1. \((T, Tₐₜ; L) \quad \Psi, xₛ : \text{squeue Aₛ} ; \Delta, yₛ : Aₛ \vdash Qₐₜ :: (zₛ : Cₛ) \) \quad \Psi, xₛ : \text{squeue Aₛ} ; \Delta \vdash yₛ \leftarrow \text{acquire xₛ ; Qₐₜ :: (zₛ : Cₛ)}

2. \((T, Tₐₜ; L) \quad \Psi, xₛ : Aₛ ; \Delta \vdash \text{xₛ \rightarrow release yₛ ; Qₐₜ :: (zₛ : Cₛ)} \) \quad \Psi, \Delta, yₛ : \text{squeue Aₛ} \vdash \text{xₛ \leftarrow release yₛ ; Qₐₜ :: (zₛ : Cₛ)}

3. \((T, Tₐₜ; R) \quad \Psi ; \vdash Pₐₜ :: (yₛ : Aₛ) \) \quad \Psi ; \vdash yₛ \leftarrow \text{accept xₛ ; Pₐₜ :: (xₛ : Aₛ)}

4. \((T, Tₐₜ; R) \quad \Psi ; \vdash Pₐₜ :: (xₛ : Aₛ) \) \quad \Psi ; \vdash xₛ \leftarrow \text{detach yₛ ; Pₐₜ :: (yₛ : Aₛ)}
Due to its foundation in intuitionistic linear logic, \textsc{sill}_S’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 $\Psi;\Delta \vdash P::(x_i : A_i)$ and $\Psi \vdash P::(x_i : A_i)$ read as “process $P$ offers a session of type $A$ along channel $x$ using sessions offered along channels in $\Psi$ (and $\Delta$).” The typing contexts $\Psi$ and $\Delta$ provide the typing of shared and linear channels, resp. Whereas $\Psi$ is a structural context, $\Delta$ 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 \texttt{accept} and \texttt{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 \textsc{sill}_R – A stepping stone from \textsc{sill}_S to Ferrite

In Section 2, we reviewed \textsc{sill}_S and its typing judgment. Our goal with Ferrite is to faithfully and compositionally encode \textsc{sill}_S typing derivations in Rust. However, when viewed under the lens of a general purpose programming language, most readers will find \textsc{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 \textsc{sill}_S. Moreover, to realize such an embedding we must be able to account for both \textsc{sill}_S’s linear session discipline (i.e. the linear context $\Delta$) and shared session discipline (i.e. the structural context $\Psi$) 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 \textsc{sill}_R as a formal stepping stone between \textsc{sill}_S and our embedding, Ferrite. \textsc{sill}_R is, in its essence, a pragmatic extension of \textsc{sill}_S with Rust (type and term) constructs, allowing us to intersperse Rust code with the communication primitives of \textsc{sill}_S. In \textsc{sill}_R we use the judgment $\Gamma;\Delta \vdash expr :: A$, denoting that expression $expr$ has session type $A$, using the sessions tracked by $\Gamma$ and $\Delta$.

This judgment differs from that of \textsc{sill}_S in its context region $\Gamma$ and term $expr$, with the latter permitting arbitrary Rust expressions in addition to \textsc{sill}_S primitives. Whereas \textsc{sill}_S’s structural context $\Psi$ exclusively tracks shared channels, \textsc{sill}_R’s $\Gamma$ tracks both shared sessions (subject to weakening and contraction) and plain Rust (affine) variables. A shared channel type in both \textsc{sill}_R and \textsc{sill}_S is always of the form $\uparrow^\gamma A$, so there is no confusion among the affine and shared contents of $\Gamma$. 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 \texttt{Clone} trait.
The basic idea underlying this encoding can be schematically described as follows:

\[ \text{Table 2 Overview of SILL}_R \text{ types and terms and their encoding in Ferrite. Note that SILL}_R \text{ uses } \tau \triangleleft A_i \text{ and } \tau \triangleright A_i \text{ for shared channel output and input, resp., and } \epsilon \text{ for termination.} \]

<table>
<thead>
<tr>
<th>Type</th>
<th>SILL(_R)</th>
<th>Terms (SILL(_S))</th>
<th>client</th>
</tr>
</thead>
<tbody>
<tr>
<td>InternalChoice&lt;Row&gt;</td>
<td>{l_i : A_i}</td>
<td>offer (l_i; K)</td>
<td>case (a {l_i : K})</td>
</tr>
<tr>
<td>ExternalChoice&lt;Row&gt;</td>
<td>{l_i : A_i}</td>
<td>offer_choice({l_i : K_i})</td>
<td>choose (a; l_i; K)</td>
</tr>
<tr>
<td>SendChannel(&lt;A,B&gt;)</td>
<td>(A_i \otimes B_i)</td>
<td>send_channel_from a; K</td>
<td>(a \leftarrow \text{receive}_\text{channel})_from f a; K</td>
</tr>
<tr>
<td>ReceiveChannel(&lt;A,B&gt;)</td>
<td>(A_i \rightarrow B_i)</td>
<td>a \leftarrow \text{receive}_\text{channel}; K</td>
<td>(\text{send}<em>\text{channel}</em>\text{to}_f a; K)</td>
</tr>
<tr>
<td>SendValue(&lt;T,A&gt;)</td>
<td>(\tau \triangleleft A_i)</td>
<td>send_value x; K</td>
<td>(x \leftarrow \text{receive}<em>\text{value}</em>\text{from}_a x; K)</td>
</tr>
<tr>
<td>ReceiveValue(&lt;T,A&gt;)</td>
<td>(\tau \triangleright A_i)</td>
<td>x \leftarrow \text{receive}_\text{value}; K</td>
<td>(\text{send}<em>\text{value}</em>\text{to}_a x; K)</td>
</tr>
<tr>
<td>End</td>
<td>(\epsilon)</td>
<td>terminate</td>
<td>wait a; K</td>
</tr>
<tr>
<td>SharedToLinear&lt;A&gt;</td>
<td>(\downarrow_l A_i)</td>
<td>detach_shared_session; K (_i)</td>
<td>release_shared_session (a; K_i)</td>
</tr>
<tr>
<td>LinearToShared&lt;A&gt;</td>
<td>(\uparrow_l A_i)</td>
<td>accept_shared_session; K (_i)</td>
<td>(a \leftarrow \text{acquire}<em>\text{shared}</em>\text{session}_s; K_i)</td>
</tr>
</tbody>
</table>

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 (\(A_i \otimes A_i\) and \(A_i \rightarrow A_i\)) 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:

\[ \text{T}_{\text{cmp}} \quad \Gamma; \Delta \vdash K : A \quad \Gamma; x : \tau; \Delta, a : A \vdash K : B \]

\[ \text{T}_{\text{send}} \quad \Gamma, x : \tau; \Delta \vdash \text{send}_\text{value} x; K : \tau \triangleleft A \quad \Gamma; \Delta, a : \tau \triangleright A \vdash x \leftarrow \text{receive}_\text{value}_\text{from}_a ; K : B \]

Rule T\(_{\text{cmp}}\) indicates that the value bound to variable \(x\) of type \(\tau\) will be sent, after which the continuation \(K\) will execute, offering type \(A\). Dually, rule T\(_{\text{send}}\) states that using such a provider bound to \(a\) will bind \(x\) of type \(\tau\) 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:

\[ \Gamma; \Delta_2 \vdash \text{cont} : A_2 \quad \text{fn expr<...>} \]

\[ \Gamma; \Delta_1 \vdash \text{expr}; \text{cont} : A_1 \quad (\text{cont: PartialSession}<\text{C}_2, A_2>) \]

\[ \rightarrow \text{PartialSession}<\text{C}_1, A_1> \]

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 \(\Gamma; \Delta \vdash \text{expr} : A\) as a value of Rust type PartialSession\(<\text{C}, A>\), where \(\text{c}\) encodes the linear context \(\Delta\) and \(\text{a}\) 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\(<\text{C}_2, A_2>\) and returns a PartialSession\(<\text{C}_1, A_1>\), 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\(_B\)), 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 \(\Delta\), the representation of the shared and affine context region \(\Gamma\) is achieved through Rust’s normal
Table 3: Judgmental embedding of SILL\(_R\) in Ferrite.

<table>
<thead>
<tr>
<th>SILL(_R)</th>
<th>Ferrite</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma; \cdot \vdash A)</td>
<td>Session(\langle A \rangle)</td>
<td>Typing judgment for top-level session (i.e. closed program).</td>
</tr>
<tr>
<td>(\Gamma; \Delta \vdash A)</td>
<td>PartialSession(\langle C, A \rangle)</td>
<td>Typing judgment for partial session.</td>
</tr>
<tr>
<td>(\Delta)</td>
<td>C: Context</td>
<td>Linear context; explicitly encoded.</td>
</tr>
<tr>
<td>(\Gamma)</td>
<td>-</td>
<td>Shared / Affine context; delegated to Rust.</td>
</tr>
<tr>
<td>(A)</td>
<td>A: Protocol</td>
<td>Session type.</td>
</tr>
</tbody>
</table>

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\(\langle A \rangle\), 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\(\langle C, A \rangle\) 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-kindred types such as \(\text{Type} \rightarrow \text{Type}\), we use defunctionalization [36, 46] by accepting any Rust type \(F\) implementing the trait RecApp with a given associated type \(F::\text{Applied}\), as shown below. Section 5.1 provides further details.

```rust
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 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:

```rust
trait SharedRecApp<X> { type Applied; }
trait SharedProtocol { ... }
struct SharedToLinear<F> { ... } struct SharedChannel<S: SharedProtocol> { ... }
struct LinearToShared<F: SharedRecApp<SharedToLinear<SharedChannel<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 \([A_0, A_1, \ldots, A_{N-1}]\) to annotate a list of session types of arbitrary length. The notation \(\text{HList}![A_0, A_1, \ldots, A_{N-1}]\) denotes a heterogeneous list of \(N\) session types, with \(A_i\) being the session type at the \(i\)-th position of the list. The \(\text{HList}!\) macro acts as syntactic sugar for the heterogeneous list, which in its raw form is encoded as \((A_0, (A_1, (\ldots, (A_{N-1}, ()))))\). Ferrite uses the Rust tuple constructor \((,)\) for \(\text{HCons}\), and unit \((\_)\) for \(\text{HNil}\). 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 \((\text{Z})\) and successor \((\text{S}<N>\)\), 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:

\[
\Gamma; \Delta; l_n : B_2 \vdash K :: A_2 \\
\Gamma; \Delta; l_n : B_1 \vdash \text{expr} \; l_n; K :: A_1
\]

\[
\text{fn expr<...> ( 1: N, cont: PartialSession<C1, A2> )}
\rightarrow \text{PartialSession<C2, A1>}
\]

\[
\text{where N: ContextLens<C1, B1, B2, Target=C2>}
\]

The index \(N\) amounts to the type of the variable 1 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:

\[
\Gamma; \Delta \vdash K :: A_n \\
\Gamma; \Delta \vdash \text{offer_case} \; l_n; K :: \oplus\{\ldots, l_n : A_n, \ldots\}
\]

\[
\text{fn offer_case<N, Row, C, A> ( 1: N, cont: PartialSession<C, A> )}
\rightarrow \text{PartialSession<C, InternalChoice<Row>}}
\]

\[
\text{where N: Prism<Row, Elem=A>, \ldots}
\]

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 \text{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].

\[
\Gamma, a : \tau; \Delta \vdash K :: A \quad (T \triangleright_R)
\]

\[
\Gamma; \Delta \vdash a \leftarrow \text{receive}_\text{value}; K :: \tau \triangleright A
\]

The SILL \(_R\) right rule \(T \triangleright_R\) types expression \(a \leftarrow \text{receive}_\text{value}\); \(K\) with session type \(\tau \triangleright A\) and the continuation \(K\) with session type \(A\), where \(a\) is now in scope with type \(\tau\). Following the schema hinted in Section 3.2, Ferrite encodes this rule as the function \(\text{receive}_\text{value}\), parameterized by a value type \(\tau \rightarrow \text{PartialSession}\(C, A\)) and an offered session type \(A\).

\[
\text{fn receive_value}_T, C: \text{Context}, A: \text{Protocol}\{\text{cont}: \text{impl FnOnce}(T) \rightarrow \text{PartialSession}\(C, A\)}\}
\rightarrow \text{PartialSession}\(C, \text{ReceiveValue}_T, A\)
\]

The function yields a value of type \(\text{PartialSession}\(C, \text{ReceiveValue}_T, A\)\), i.e. the conclusion of the rule, given an (affine) closure of type \(T \rightarrow \text{PartialSession}\(C, A\)\), encoding the premise of the rule. Notably, Ferrite uses plain Rust binding (through function types) to encode the contents of \(\Gamma\), 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 \(\text{FnOnce}\) trait, ensuring that it can only be called once.

The type \(\text{PartialSession}\) is a core construct of Ferrite that enables the judgmental embedding of SILL \(_R\). A Rust value of type \(\text{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 \(\Gamma; \Delta \vdash \text{expr} :: A\). The type parameters \(C\) and \(A\) are constrained to implement the traits \(\text{Context}\) and \(\text{Protocol}\) – two other Ferrite constructs representing a linear context and linear session type, resp.:

\[
\text{trait Context} \{ \ldots \} \quad \text{trait Protocol} \{ \ldots \}
\]

\[
\text{struct PartialSession}\(C: \text{Context}, A: \text{Protocol}\) \{ \ldots \}
\]

For each SILL \(_R\) session type, Ferrite defines a corresponding Rust struct that implements the trait \(\text{Protocol}\), yielding the listing shown in Table 2. Implementations for \(\epsilon\) (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 \(\text{ReceiveValue}_T, A\), the constraint to implement \(\text{Protocol}\) is propagated to the inner session type, requiring \(A\) to also implement \(\text{Protocol}\):

\[
\text{struct End} \{ \ldots \} \quad \text{struct ReceiveValue}_T, A \{ \ldots \}
\]

\[
\text{impl Protocol}\text{ for End} \{ \ldots \} \quad \text{impl}\(A: \text{Protocol}\)\text{ for ReceiveValue}_T, A \{ \ldots \}
\]

Thus, while Ferrite delegates the handling of the shared/structural context \(\Gamma\) to Rust, the encoding of the linear context \(\Delta\) 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 \(\text{HList}![A_0, A_1, \ldots, A_{N-1}]\), with all its type
elements \(A_i\), implementing \(\text{Protocol}\). Internally, the \texttt{HList} macro desugars the type-level list into a nested tuple \((A_0, (A_1, \ldots, (A_{N-1}, ()))))\). The unit type \((\text{}\)) is used as the empty list \((\texttt{HNil})\) and the tuple constructor \((,\)\) is used as the \texttt{HCons} constructor. The implementation for \(\text{Context}\) is defined inductively as follows:

\[
\text{impl Context for } () \{ \ldots \} \quad \text{impl A: Protocol, C: Context > Context for } (A, C) \{ \ldots \}
\]

To represent a closed program, i.e., a program without free channel variables, we define a type alias \(\text{Session}\langle A\rangle\) for \(\text{PartialSession}\langle C, A\rangle\), with \(C\) restricted to the empty context:

\[
\text{type Session}\langle A\rangle = \text{PartialSession}\langle\text{}, A\rangle;
\]

A complete session type program in Ferrite is thus of type \(\text{Session}\langle A\rangle\) and amounts to the \texttt{SILL} typing derivation proving that the program adheres to the defined protocol. Below we show a “hello world”-style program in Ferrite:

\[
\text{let hello_provider = receive_value(|name| \{ println!("Hello, {name}"), terminate() \})};
\]

The Ferrite program \text{hello_provider} has an inferred Rust type \(\text{Session}\langle\text{ReceiveValue}\langle\text{String, End}\rangle\rangle\). It offers the type \(\text{ReceiveValue}\langle\text{String, End}\rangle\) by first receiving a string value using \texttt{receive_value}, binding it to \textit{name} in the continuation closure. Upon receiving the name string, it prints out the name with a “Hello” greeting, and terminates using \texttt{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 \texttt{ContextLens} trait, which is implemented using type-level natural numbers.

\[
\text{#[derive(Copy)] struct Z; \quad \text{#[derive(Copy)] struct SN}(\text{PhantomData<N>});}
\]

\[
\text{trait ContextLens<C: Context, A1: Protocol, A2: Protocol> \{ type Target: Context; \ldots \}}
\]

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

\[
\text{N: ContextLens}\langle\texttt{HList}[\ldots, A_N, \ldots]\rangle, A_N, B, \text{Target} = \texttt{HList}[\ldots, B, \ldots]\rangle
\]

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:

\[
\text{impl A1: Protocol, A2: Protocol, C: Context > ContextLens}\langle\texttt{A1, C}, A1, A2\rangle \text{ for } Z \{ \text{type Target} = (A2, C); \ldots \}
\]

\[
\text{impl A1: Protocol, A2: Protocol, B: Protocol, C: Context, N: ContextLens}\langle\text{C, A1, A2}\rangle \text{ for SN} \{ \text{type Target} = (B, N::Target); \ldots \}
\]

In the inductive case, for any natural number \(N\) implementing the context lens for a context \(\texttt{HList}[A_0, \ldots, A_N, \ldots]\), its successor \(\texttt{SN}\) implements the context lens for \(\texttt{HList}[A_{N-1}, A_0, \ldots, A_N, \ldots]\), with a new element \(A_{N-1}\) appended to the head of the linear context. Using context lenses, we can encode the \texttt{SILL} left rule \(T_3\) shown below, which types sending an ambient value \(x\) to a channel \(a\) in the linear context that expects to receive a value.
\[ \Gamma; \Delta, a : A \vdash K :: B \]
\[ \Gamma, x : \tau; \Delta, a : \tau \vdash \text{send}\_\text{value\_to} a x; K :: B \]

\((T_{\text{BL}})\)

In Ferrite, \(T_{\text{BL}}\) 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 \(C_1\). This requires the \(N\)-th channel to have type \(\text{ReceiveValue}\langle T,A\rangle\). A continuation \(cont\) is then given with the linear context \(C_2\), which has the \(N\)-th channel updated to type \(A\).

\[ \text{fn} \text{send\_value\_to}<N, T, C_1: \text{Context}, C_2: \text{Context}, A: \text{Protocol}, B: \text{Protocol}> \]
\[ ( n: N, x: T, cont: \text{PartialSession}<C_2, B> ) \rightarrow \text{PartialSession}<C_1, B> \]
\[ \text{where} \ N: \text{ContextLens}<C_1, \text{ReceiveValue}<T,A>, A, \text{Target}=C_2 \]

**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 \(-\circ\). 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:

\[ \text{struct} \ \text{Empty}; \quad \text{trait} \ \text{Slot} \{ \ldots \} \]
\[ \text{impl} \ \text{Slot} \ \text{for} \ \text{Empty} \{ \ldots \} \quad \text{impl}<A: \text{Protocol}> \ \text{Slot} \ \text{for} \ A \{ \ldots \} \]

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`.

\[ \Gamma; \Delta \vdash K :: A \]
\[ \Gamma, \Delta, a : \epsilon \vdash \text{wait}\_a; K :: A \]
\[ \Gamma; \vdash \text{terminate}; :: \epsilon \]

\((T_{\text{LR}})\)

Using `Empty`, it is straightforward to implement SILL\(_R\)'s session termination. Rule \(T_{\text{LR}}\) is encoded via a context lens that replaces a channel of session type \(\text{End}\) 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.

\[ \text{fn} \ \text{wait}<C_1: \text{Context}, C_2: \text{Context}, A: \text{Protocol}, N> \]
\[ ( n: N, \text{cont}: \text{PartialSession}<C_2, A> ) \rightarrow \text{PartialSession}<C_1, A> \]
\[ \text{where} \ N: \text{ContextLens}<C_1, \text{End}, \text{Empty}, \text{Target}=C_2 \]

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

\[ \text{trait} \ \text{EmptyContext}: \text{Context} \{ \ldots \} \quad \text{impl} \ \text{EmptyContext} \ \text{for} \ () \{ \ldots \} \quad \text{impl}<C: \text{EmptyContext}> \ \text{EmptyContext} \ \text{for} \ (\text{Empty, } C) \{ \ldots \} \]

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

\[ \text{fn} \ \text{terminate}<C: \text{EmptyContext}>() \rightarrow \text{PartialSession}<C, \text{End}> \]
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 $T \rightarrow_R$ 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 $\Delta$ of the continuation.

\[
\frac{\Gamma; a : A \vdash K :: B}{\Gamma; \Delta \vdash a \leftarrow \text{receive_channel}; K :: A \rightarrow B} \quad (T \rightarrow_R)
\]

To encode $T \rightarrow_R$, an append operation on contexts is defined via the `AppendContext` trait:

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

The `AppendContext` trait is parameterized by a linear context $C$ and an associated type `Appended`. If a linear context $C_1$ implements the trait `AppendContext<C2>`, it means that context $C_2$ can be appended to $C_1$, with $C_3 = C_1::Appended$ 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<[List![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.

```rust
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 $T \rightarrow_R$ is then encoded as follows:

```rust
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, ()), Append=C2>
```

The function `receive_channel` is parameterized by a linear context $C_1$ implementing `AppendContext` to append the session type $A$ to $C_1$. The continuation argument `cont` is a closure that is given a context lens $C::Length$, and returns a `PartialSession` with $C2=C_1::Appended$ as its linear context. The function returns a `PartialSession` with linear context $C_1$, offering session type `ReceiveChannel<A, B>`.

We note that in the type signature of `receive_channel`, the type $C_1::Length$ is not shown to have any `ContextLens` implementation. However when $C_1::Length$ is instantiated to the concrete types $Z, S<X>$, 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:

```rust
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>, 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.

Γ; Δ₁ ⊢ K₁ :: A  Γ; Δ₂, a : A ⊢ K₂ :: B  \( (\text{T-CUT}) \)
Γ; Δ₁, Δ₂ ⊢ a ← cut K₁; K₂ :: B  \( (\text{T-FWD}) \)

In SILLₗᵣ, rule T-CUT allows two session-typed programs to run in parallel, with the channel offered by K₁ added to the linear context of program K₂. 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 SILLₗᵣ:

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

To implement cut in Ferrite, we need a way to split a linear context C = Δ₁, Δ₂ into two sub-contexts C₁ = Δ₁ and C₂ = Δ₂ 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:

```r
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:

```r
(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:

```r
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 X3 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:

```rust
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:

```rust
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:

```rust
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:

```rust
trait RecApp<X> { type Applied; } type AppRec<F, X> = <F as RecApp<X>>::Applied;
```

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<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`:

```rust
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:
A recursive session type is defined starting with an empty recursion context. Since nested recursive session types allow a \( \text{RecX} \) to be embedded inside another \( \text{RecX} \), we have \( \text{RecX} \) also implement \( \text{RecApp} \), provided it has an empty recursion context. When unfolded from another recursion context \( R \), \( \text{RecX} \) 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 \( \text{RecX} \) 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:

\[
\text{impl} \langle A, R \rangle \text{RecApp}<(A, R)> \text{for} Z \{ \text{type} \text{Applied} = A; \}
\]

\[
\text{impl} \langle A, R, N: \text{RecApp}<R> \rangle \text{RecApp}<(A, R)> \text{for} S<N> \{ \text{type} \text{Applied} = N::\text{Applied}; \}
\]

### 5.2 Shared Session Types

In the previous section we explored a recursive session type \( \text{Counter} \), which is defined using \( \text{Rec} \) and \( Z \). Since \( \text{Counter} \) 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 \textit{shared} session types in Ferrite, enabling \textit{safe} shared communication in the presence of multiple clients.

#### Shared Session Types in Ferrite

As introduced in Section 2, the \( \text{SILL}_S \) (and \( \text{SILL}_R \)) 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 \( \text{Counter} \) type in \( \text{SILL}_R \) is:

\[
\text{SharedCounter} = \uparrow^1 \text{Int} \circ \downarrow^5 \text{SharedCounter}
\]

The linear portion of \( \text{SharedCounter} \) in between \( \uparrow^1 \) (acquire) and \( \downarrow^5 \) (release) amounts to a critical section. When a \( \text{SharedCounter} \) is \textit{acquired}, it offers a linear session type \( \text{Int} \circ \downarrow^5 \text{SharedCounter} \), willing to send an integer value, after which it must be \textit{released} to become available again as a \( \text{SharedCounter} \) to the next client.

The recursive aspect of shared session types in \( \text{SILL}_R \) means that we can reuse the implementation technique that we use for recursive session types. The type \( \text{SharedCounter} \) can be defined in Ferrite as follows:

\[
\text{type} \text{SharedCounter} = \text{LinearToShared}<\text{SendValue}<\text{u64}, \text{Release}>;\]

Compared to linear recursive session types, the main difference is that instead of using \( \text{Rec} \), a shared session type is defined using the \( \text{LinearToShared} \) construct. This corresponds to \( \uparrow^1 \) in \( \text{SILL}_R \), with the inner type \( \text{SendValue}<\text{u64}, \text{Release}> \) corresponding to the linear portion of the shared session type. At the point of recursion, the type \( \text{Release} \) is used in place of \( \downarrow^5 \text{SharedCounter} \). As a result, the type \( \text{LinearToShared}<\text{SendValue}<\text{u64}, \text{Release}> \) is unfolded into \( \text{SendValue}<\text{u64}, \text{SharedToLinear}<\text{LinearToShared}<\text{SendValue}<\text{u64}, \text{Release}> \) after being acquired. Type unfolding is implemented as follows:

\[
\text{trait} \text{SharedRecApp}<X> \{ \text{type} \text{Applied}; \} \text{trait} \text{SharedProtocol} \{ \ldots \}
\text{struct} \text{SharedToLinear}<F> \{ \ldots \} \text{struct} \text{LinearToShared}<F> \{ \ldots \}
\text{impl} <F> \text{Protocol} \text{for} \text{SharedToLinear}<\text{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 ↓S↓L in SILL. 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:

```rust
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 SILL 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.

```rust
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 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:

```rust
type SharedCounter = LinearToShared<SendValue<u64, Release>>;
f32 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.

```rust
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\(_5\) 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 \text{offer\_left}; K : A \oplus B} \quad \text{(T\(_{\oplus 2L_R}\))}
\]

\[
\frac{\Gamma; \Delta \vdash K : B}{\Gamma; \Delta \vdash \text{offer\_right}; K : A \oplus B} \quad \text{(T\(_{\oplus 2R_R}\))}
\]

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

```rust
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:

```racket
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:

```racket
```

The function accepts a natural number N as the first parameter, which acts as the prism for selecting a session type AN out of the row type Row=HList![..., AN, ...]. Through the associated type AN::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 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, T2R:

\[
\begin{align*}
\Gamma; \Delta \vdash K_l :: A \\
\Gamma; \Delta \vdash K_r :: B \\
\Gamma; \Delta \vdash \text{offer\_choice\_2} K_l, K_r :: A\&B (T2R)
\end{align*}
\]

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:

```racket
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>:

eenum 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 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:
enum CanvasMsg { Canvas2d(Canvas2dMsg, CanvasId), Close(CanvasId), ... }
enum Canvas2dMsg { LineTo(Point2D), GetTransform(Sender<Transform2D>), ... }
enum ConstellationCanvasMsg { Create { id_sender: Sender<CanvasId>, size: Size2D } }
struct CanvasPaintThread { canvases: HashMap<CanvasId, CanvasData>, ... }
impl CanvasPaintThread { ... }
fn start() -> (Sender<ConstellationCanvasMsg>, Sender<CanvasMsg>) {
    let (msg_sender, msg_receiver) = channel();
    let (create_sender, create_receiver) = channel();
    thread::spawn(
        move || {
            let canvas_msg_receiver = recv(canvas_msg_receiver) -> { ...
            CanvasMsg::Canvas2d(message, canvas_id) => { ...
            Canvas2dMsg::LineTo(point) => self.canvas(canvas_id).move_to(point),
            Canvas2dMsg::GetTransform(sender) => sender.send(self.canvas(canvas_id).get_transform()).unwrap(), ... }
            CanvasMsg::Close(canvas_id) => canvas_paint_thread.canvases.remove(&canvas_id) }
            recv(create_receiver) -> { ConstellationCanvasMsg::Create { id_sender, size } => {
                let canvas_id = ...;
                self.canvases.insert(canvas_id, CanvasData::new(size, ...));
                id_sender.send(canvas_id); }
            (create_sender, msg_sender) } ...
    });
    (create_sender, msg_sender) }
fn canvas(&mut self, canvas_id: CanvasId) -> &mut CanvasData {
    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:

```rust
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.

```rust
enum CanvasMessage { BeginPath, ClosePath, Fill(FillOrStrokeStyle), ... }
```

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:

```rust
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:

```rust
define_choice! { CanvasOps; Message: ReceiveValue<CanvasMessage, Release>, GetImageData: ReceiveValue<Rect<u64>, Size2D<u64>, IpcBytesSender>, SendValue<ByteBuf, Release>, IsPointInPath: ReceiveValue<(f64, f64, FillRule), SendValue<bool, Release>>, ... }
```
Table 4 MotionMark Benchmark scores in fps (higher is better).

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Servo</th>
<th>Servo/Ferrite</th>
<th>Firefox</th>
<th>Chrome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arcs</td>
<td>12.21 ± 6.75%</td>
<td>11.83 ± 11.49%</td>
<td>52.61 ± 32.88%</td>
<td>46.00 ± 9.00%</td>
</tr>
<tr>
<td>Paths</td>
<td>43.76 ± 10.66%</td>
<td>40.98 ± 18.94%</td>
<td>55.59 ± 28.80%</td>
<td>59.50 ± 14.90%</td>
</tr>
<tr>
<td>Lines</td>
<td>7.48 ± 7.06%</td>
<td>11.47 ± 12.74%</td>
<td>14.35 ± 6.65%</td>
<td>32.43 ± 6.48%</td>
</tr>
<tr>
<td>Bouncing clipped rects</td>
<td>18.45 ± 7.06%</td>
<td>18.23 ± 11.00%</td>
<td>34.82 ± 7.76%</td>
<td>58.07 ± 19.85%</td>
</tr>
<tr>
<td>Bouncing gradient circles</td>
<td>8.02 ± 7.74%</td>
<td>7.72 ± 12.63%</td>
<td>58.79 ± 21.03%</td>
<td>59.77 ± 10.07%</td>
</tr>
<tr>
<td>Bouncing PNG images</td>
<td>7.97 ± 5.91%</td>
<td>6.31 ± 10.26%</td>
<td>24.61 ± 6.35%</td>
<td>59.94 ± 13.04%</td>
</tr>
<tr>
<td>Stroke shapes</td>
<td>10.60 ± 3.95%</td>
<td>10.35 ± 10.96%</td>
<td>51.21 ± 11.25%</td>
<td>59.38 ± 16.87%</td>
</tr>
<tr>
<td>Put/get image data</td>
<td>60.01 ± 3.81%</td>
<td>32.08 ± 10.83%</td>
<td>50.66 ± 20.16%</td>
<td>60.00 ± 5.00%</td>
</tr>
</tbody>
</table>

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-kindred 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 1channels [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 an open question to find suitable approaches that work well in Rust.
References


22:28 Ferrite: A Judgmental Embedding of Session Types in Rust


