# Session Types for Rust

Thomas Bracht Laumann Jespersen Philip Munksgaard Ken Friis Larsen

Department of Computer Science, University of Copenhagen, Denmark

ntl316@alumni.ku.dk pmunksgaard@gmail.com kflarsen@diku.dk

## Abstract

We present a library for specifying session types implemented in Rust, and discuss practical use cases through examples and demonstrate how session types may be used in a large-scale application. Specifically we adapt parts of the ad-hoc communication patterns in the Servo browser engine to use session typed channels. Session types provide a protocol abstraction, expanding on traditional typed communication channels, to ensure that communication takes place according to a specified protocol. Thus, the library allows us to provide compile-time guarantees of adherence to a specific protocol without incurring significant run-time penalties.

***Categories and Subject Descriptors*** D.1.3 [*Software Programming Techniques*]: Concurrent programming; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

***Keywords*** Session types, concurrency, generic types, Rust

## 1. Introduction

Communication in concurrent applications is often implemented by message-passing, in which two processes can exchange information via a channel. Typically channels are uni-directional and reciprocal interaction is implemented by means of two unidirectional channels. In large concurrent applications the communication schemes often implement an implicit ad hoc protocol, and it is the responsibility of the programmer to ensure that the protocol is obeyed, as message-passing libraries usually do not offer any mechanisms for specifying protocols.

Session types allow the specification of protocols as types and its associated type discipline ensures that only compatible session type protocols are typeable. Session types require a linear usage of the channels and this is a challenge in the majority of programming languages where aliasing is allowed.

We implement session types in the programming language Rust which provides affine types and argue that affine types are sufficient for preventing protocol violations.

The appeal of session types is that they provide a static guarantee of protocol safety and in our implementation there are no run-time checks to ensure protocol safety, so we argue that the performance overhead of using session types is small.

## 2. Background

The section gives an introduction to Rust and session types. We describe some of the features of Rust that are important to our implementation of session types in Section 2.1, and describe the session typing system in Section 2.2.

### 2.1 Rust

Rust is a systems programming language, initially developed independently by Graydon Hoare before being adopted by Mozilla and developed on a larger scale. The main purpose of Rust is to enable programmers to easily write fast and efficient programs that are both memory and thread safe.

Because it is aimed at being a systems programming language, Rust is designed to offer a high level of control over low-level details like memory allocation, so the programmer can reason about how the program will behave at run-time. In particular, this means that there is no garbage collection. To achieve safety in the absence of garbage collection, Rust employs compile-time static analysis techniques and semantics to determine exactly when an allocated value goes out of scope such that it can be freed. Thus, the Rust compiler decides, at compile-time, where to place destructors and drop statements, freeing the programmer from that responsibility, while guaranteeing memory safety.

Rust also incorporates features typically associated with functional languages of the ML family, such as pattern matching, algebraic data types, strong static typing, higher-order functions and closures. Syntactically, Rust is close to C, employing brace-delineated blocks for structure and semi-colons to terminate statements. For an in-depth introduction to Rust, we refer to the online book *The Rust Programming Language*[1].

The development of Rust is community-driven and publicly available on GitHub[2]. At the time of writing, there is no official language specification, but there is a reference document that informally describes the language constructs, the memory model and other parts of the language [5]. Work has been done to provide a formalization of Rust, with formal semantics and soundness proofs for a collection of core operations [6]. The Rust compiler, `rustc`, serves as a reference implementation.

***Move Semantics, Borrows and Affine Types*** Rust brings some new and innovative concepts into mainstream programming languages, the most significant is probably its statically guaranteed memory safety (while still retaining control over low-level details), which is achieved through the *borrow system* and by using *move semantics*. This section serves as an introduction to the borrow system, move semantics, and how Rust supports affine types through these concepts, all of which are essential to the implementation of our session type library.

[1] `https://doc.rust-lang.org/nightly/book/`

[2] `http://github.com/rust-lang/rust`

```
struct Foo;

fn bar() {
    let x = Foo;      // initialize x
    let y = x;        // consume x
}
```

**Figure 1.** Moving values in Rust

By default, using a value in Rust *consumes* the value, which means that it cannot be used again at a later point. Consider the example in Figure 1. Assigning `x` to `y` consumes `x` and makes future references to `x` illegal. For instance, if we tried to assign `x` to another value `z` or use `x` in a function after we had already assigned `x` to `y`, that would result in a compile time error. This behavior is called *move semantics*, and we also say that `x` is *moved* into `y`. For example, a call like `some_fun(x)` would move `x` into `some_fun` and make it unavailable in the calling function. In addition to move semantics, Rust also supports *copy semantics* where you make a copy of a value before using it, which mimic traditional behavior from languages such as C. Copy semantics is the default for primitive types and can be opted into for compound data types.

With these semantics, Rust can determine when to safely deallocate resources: Each value has a unique owner, a scope, which can change through function calls, assignments, etc. If a value has not been consumed by the end of its owning scope, Rust can safely deallocate it. At the end of the function `bar()` in Figure 1, there are no more references to `y`, so the Rust compiler automatically inserts deallocation calls.

To support multiple references to the same value, Rust uses a concept called *borrowing*. A reference, sometimes called a borrow, can be either mutable or immutable. An immutable reference to a value `x` is denoted `&x` and a mutable reference is denoted `&mut x`. See Figure 2 for an example that uses both mutable and immutable references. To support the move semantics in Rust, all references carry information about the scope in which it can be used, called its *lifetime*. In the simplest case, a reference's lifetime is the scope in which it is declared, but it can also be tied to other references' lifetimes. Using lifetimes, Rust is able to statically guarantee that references never outlive their referent. Additionally, the borrow system ensures that data races cannot happen: A mutable reference, through which we can mutate the original data, excludes all other references, both mutable and immutable, and any reference at all prohibits mutation of the referent. This way, mutation can only happen with exclusive access and data races are avoided.

As a small aside, sometimes it is necessary to be able to express safe programs that cannot be statically verified and Rust provides an escape hatch via the keyword `unsafe`. The keyword `unsafe` is used to mark functions, blocks of code, traits and method implementations as unsafe, telling Rust to set aside some of its usual safety checks. This lets you handle, among other things, raw pointers, perform foreign function calls to C code, manually handle memory allocation and deallocation. When a piece of code has been marked unsafe it is the programmer's responsibility to ensure that Rust's safety guarantees are restored at the end of the code.

Because of its move semantics, we can say that Rust has an affine type system. By wrapping essential data in non-copying data structures and ensuring that all methods and functions which interact with said data structures are consuming, that data type becomes affine: We can guarantee that values of that type are used *at most once*. That Rust has affine types is essential for our implementation of session types. However, before we introduce our library we provide a short introduction to session types.

```
struct Foo(u8);

fn print_foo(&Foo(n): &Foo) {
    println!("Foo({})", n);
}

fn add1(x: &mut Foo) {
    x.0 += 1;
}

let mut x = Foo(42);

{                         // --+
    let r = &mut x;       //   | lifetime
    add1(r);              //   | of 'r'
    add1(r);              //   |
}                         // --+

print_foo(&x);            // outputs "Foo(44)"
```

**Figure 2.** Using references in Rust

### 2.2 Session Types

Session types were introduced in [2] and provided a new method for structuring sequences of reciprocal interactions in a type-safe manner. Concurrency primitives based on message-passing have been studied greatly and implemented in a host of programming languages, but typically each interaction is considered distinct and unrelated to other interactions. The session type theory introduced a basic structuring concept called a *session*. A session has an associated (implicit) channel through which all interactions take place and the interactions via a session are modelled by a type—called a *session type*. The typing system ensures that two processes only communicate via a session if their session types are compatible. Session types allow the programmer to specify complex communication protocols without worrying about run-time errors caused by violations of the protocol.

As a concept session types are suitable for embedding in other programming languages, but were initially developed in its own language $\mathcal{L}$ to better showcase the novel features of the system in a minimal language. Since their introduction, session types have been studied widely and re-formulated in other formal frameworks, in particular the $\pi$-calculus [8], and have been implemented in a variety of programming languages, both as libraries and language extensions [1, 7].

***The Honda-Vasconcelos-Kubo Session Typing System*** To illustrate the session typing concepts, we will work with an example protocol that specifies the interaction between a client (CLIENT) and an ATM:[3]

- The CLIENT communicates his/her ID to the ATM
- The ATM then answers either `ok` or `err`
  - In the first case, the CLIENT then proceeds to request either a `deposit` or `withdraw`
    - For a `deposit` the CLIENT first sends an amount, then the ATM responds with the updated balance
    - For a `withdraw` the CLIENT sends the amount to withdraw, and the ATM responds with either `ok` or `err` to indicate whether or not the transaction was successful

[3] This example is originally from [3] and has been adapted and used by several authors.

$$
\begin{aligned}
S &::= \texttt{nat} \mid \texttt{bool} \mid \langle \alpha, \overline{\alpha} \rangle \mid \cdots \\
\alpha &::= ?[\tilde{S}]; \alpha \mid ?[\alpha]; \beta \mid \&\{\ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n\} \mid \varepsilon \mid \bot \\
&\quad \mid ![\tilde{S}]; \alpha \mid ![\alpha]; \beta \mid \oplus\{\ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n\} \mid t \mid \mu t.\alpha
\end{aligned}
$$

**Figure 3.** The syntax of types in $\mathcal{L}$.

- If the ATM answers `err`, then the session terminates.

From the point of view of the ATM, we can describe this protocol by the session type:

$$
\begin{aligned}
\text{ATM} &= ?[\texttt{id}]; \oplus\{\texttt{ok} : \text{ATM}', \texttt{err} : \varepsilon\} \\
\text{ATM}' &= \&\{\ \texttt{deposit} : ?[\texttt{u64}]; ![\texttt{u64}]; \varepsilon, \\
&\qquad\ \texttt{withdraw} : ?[\texttt{u64}]; \oplus\{\texttt{ok} : \varepsilon, \texttt{err} : \varepsilon\} \\
&\qquad \}
\end{aligned}
$$

The interactions "send" and "receive" are denoted by ! and ? respectively and for each occurence, there is an associated type—the type of the value being communicated.

The $\oplus\{\ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n\}$ construct is called *branch selection*, and indicates that the process selects one of the branches $\ell_i$. This choice is then communicated to the other party, and the process continues with the protocol $\alpha_i$. For example in the ATM, after receiving the client's `id` the ATM selects either the `ok` or the `err` branch and communicates the respective label to the client.

The opposite of branch selection is *label branching* and is denoted by $\&\{\ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n\}$. Here the process obtains the selected branch $\ell_i$ from the other party and continues with the selected protocol $\alpha_i$ (this is also sometimes called passive selection). The ATM$'$ type offers the branches `deposit` and `withdraw`, and waits for the client to communicate the selected branch.

A session type terminates when $\varepsilon$ is reached and no further communication in that session is possible. In the ATM protocol, all of the branches are terminated by $\varepsilon$ and only allows one of `deposit` or `withdraw` before terminating (we will amend this shortly). Note that there are two ways to terminate the `withdraw` branch in ATM$'$. Its final $\oplus\{\texttt{ok} : \varepsilon, \texttt{err} : \varepsilon\}$ may seem redundant, but it demonstrates how branches can be used to convey information by themselves: The client knows that if the `err` branch is taken, the request was unsuccessful.

An important aspect of session types is the sequencing of actions built into the types. All interactions, except for $\varepsilon$, cannot stand on their own, but have some notion of a subsequent action. For example, the syntax for "send" is $![\tilde{S}]; \alpha$, where $\alpha$ is some session type. Once the process has performed the "send" action, that particular interaction cannot be performed again, as the type of the session becomes $\alpha$.

The full syntax of session types is shown in Figure 3.

***Recursion*** As mentioned earlier, the ATM session type only allows one of `deposit` or `withdraw` before terminating. Often we may be interested in repeating a protocol, and we can use recursion to achieve this. Recursion is denoted by $\mu t.\alpha$, in which occurrences of $t$ in $\alpha$ are substituted for $\alpha$ in the usual way.

To illustrate the use of recursion we modify the ATM$'$ session type in the following way:

$$
\begin{aligned}
\text{ATM}' &= \mu t.\&\{\ \texttt{deposit} : ?[\texttt{u64}]; ![\texttt{u64}]; t, \\
&\qquad\ \texttt{withdraw} : ?[\texttt{u64}]; \oplus\{\texttt{ok} : t, \texttt{err} : t\}, \\
&\qquad\ \texttt{quit} : \varepsilon \\
&\qquad \}
\end{aligned}
$$

where we have added $\mu t$ before the label branching, and replaced all occurrences of $\varepsilon$ with $t$, to recurse at that point and repeat the protocol from $\mu t$. We then introduce another branch `quit`, to allow the client to terminate the session. (In the subsequent sections, we use this version of the ATM session type.)

$$
\begin{aligned}
\overline{![\tilde{S}]; \alpha} &= ?[\tilde{S}]; \overline{\alpha} & \overline{\oplus\{l_i : \alpha_i\}_{i \in I}} &= \&\{l_i : \overline{\alpha_i}\}_{i \in I} \\
\overline{?[\tilde{S}]; \alpha} &= ![\tilde{S}]; \overline{\alpha} & \overline{\&\{l_i : \alpha_i\}_{i \in I}} &= \oplus\{l_i : \overline{\alpha_i}\}_{i \in I} \\
\overline{![\alpha]; \beta} &= ?[\alpha]; \overline{\beta} & \overline{\varepsilon} &= \varepsilon \\
\overline{?[\alpha]; \beta} &= ![\alpha]; \overline{\beta} & \overline{\mu t.\alpha} &= \mu t.\overline{\alpha}
\end{aligned}
$$

**Figure 4.** The co-type (or dual) of a type $\alpha$ is denoted $\overline{\alpha}$.

Recursive types are assumed to be contractive, meaning that types do not contain a subexpression of the form $\mu t.\mu t_1.\cdots\mu t_n.t$, and the typing system takes an equi-recursive view of types, considering $\mu t.\alpha$ and its expansion $\alpha[\mu t.\alpha/t]$ equivalent.

***Duality*** An important concept in session types is the concept of *duality*. It is a formulation of the idea that if a process speaks a particular protocol, then its correspondant must be prepared to understand it. For example, if a process wishes to send a value and has the session type $![\tilde{S}]; \alpha$, the opposing process must be prepared to receive a value of same type. Its type must then be $?[\tilde{S}]; \overline{\alpha}$, where $\overline{\alpha}$ denotes the dual of $\alpha$.

Given a session type $\alpha$ we can find its dual by exchanging ! and ?, and $\oplus$ and $\&$. Formally, the dual is defined inductively on a session type as shown in Figure 4.

The dual of our ATM type—the session type for the client—is then found to be:

$$
\begin{aligned}
\overline{\text{ATM}} &= ![\texttt{id}]; \&\{\texttt{ok} : \overline{\text{ATM}'}, \texttt{err} : \varepsilon\} \\
\overline{\text{ATM}'} &= \mu t.\oplus\{\ \texttt{deposit} : ![\texttt{u64}]; ?[\texttt{u64}]; t, \\
&\qquad\ \texttt{withdraw} : ![\texttt{u64}]; \&\{\texttt{ok} : t, \texttt{err} : t\}, \\
&\qquad\ \texttt{quit} : \varepsilon \\
&\qquad \}
\end{aligned}
$$

Duality is the basic requirement for communication safety. Two communicating processes are considered *compatible* if their session types are dual, and only when they are dual is the program well-typed.

***Session Delegation*** One feature of the session typing system we have not yet mentioned is *session delegation* which allows a process to hand over its session to another process. After delegating a session, the original owner of the session can no longer interact with it. Session delegation originally received special treatment to prevent aliasing of sessions. Session types have subsequently been formulated in the context of a functional language with linear types, and it has been shown that this is sufficient to prevent aliasing.

In Section 3.4 we show how to leverage Rust's affine type system to implement session interaction, and directly avoid aliasing. As a result, we do not have to treat session delegation with any extra care, so we will not discuss it further.

## 3. Session Types in Rust

This section describes our implementation of session types in Rust. Our implementation largely mirrors the interface of the Haskell implementation described in [1], but instead of passing channels in a `Session` monad, we provide a `Chan` type that has an associated protocol.

### 3.1 Type Constructs

Each construct in the original session types formulation has a corresponding struct representation in our library. The empty protocol $\varepsilon$ is represented by the unit struct `Eps`. The "send" construct is represented by the struct

```
struct Send<A, P>(PhantomData<(A, P)>)
```

where `A` is the type of the value to send, and `P` is the protocol to execute after `Send` has been executed. The `PhantomData` field is a marker indicating that the type parameters `A` and `P` are *phantom types*. Unused type parameters (phantom types) are not permitted by the Rust compiler and must be marked explicitly using `PhantomData`. The "recv" construct is analogous.

The branching constructs & and ⊕, "offer" and "choose", are also represented as individual structs. The `Offer` struct is declared as:

```
struct Offer<P, Q>(PhantomData<(P, Q)>)
```

and reflects the (passive) choice between protocols `P` and `Q`. The `Choose` struct is declared similarly. Contrary to the original session types definition our implementation only provides binary branching, like the Haskell implementation in [1]. We have considered alternative ways to implement labelled branching, but so far we have not discovered a satisfactory solution.

Recursion is denoted by the `Rec` and `Var` structs together with Peano number structs `Z` and `S<N>`. Contrary to the other session type structs (except for `Eps`), `Rec` does not have a notion of a "next" action. `Rec` is defined as:

```
struct Rec<P>(PhantomData<P>)
```

and contains a single protocol `P`. In Section 3.4 we explain how the recursion structs are used.

***The ATM*** As an example of how to declare session types using our Rust representation, we show how the ATM protocol from Section 2.2 can be expressed:

```
type Atm = Recv<Id, Choose<Rec<AtmInner>,
                           Eps>>;
type AtmInner = Offer<AtmDeposit,
                Offer<AtmWithdraw,
                      Quit>>>;
```

where:

```
type Id          = String;
type Quit        = Eps;
type AtmDeposit  = Recv<u64, Send<u64,
                                  Var<Z>>>;
type AtmWithdraw = Recv<u64, Choose<Var<Z>,
                                    Var<Z>>>;
```

There are a few things to note about this transcription: Because our branching construct is binary, we chain `Choose` and `Offer` constructs to provide several branches. We use type aliases to name the different branches of `AtmInner`, which is useful for readability. Note that for recursion, `Var<Z>` plays the role of $t$ and refers to the innermost occurrence of `Rec`.

### 3.2 Duality

We implement duality by using Rust's trait system. Traits in Rust are equivalent to interfaces in Java or, perhaps more closely, type classes in Haskell, and are used to constrain generic type parameters in functions. A trait defines a set of method signatures (optionally with default implementations) and associated types, that can be implemented for different types in special `impl` blocks. Traits with no methods are typically called "marker traits" and notable examples in the standard library are the `Send` and `Sync` traits.

We declare a trait `HasDual`:

```
trait HasDual {
    type Dual;
}
```

The `HasDual` trait does not require any methods to be implemented, instead it requires implementors to specify an associated type, `Dual`. For types that are their own dual, such as `Eps`, the implementation is straight-forward:

```
impl HasDual for Eps {
    type Dual = Eps;
}
```

For a type like `Send`, the dual implementation is defined inductively. For `Send<A, P>` the following protocol `P` must also implement `HasDual`, which can be expressed by a trait bound on `P`. There is no bound on the type `A` of the transmitted value, but the types must be the same. The declaration is as follows:

```
impl<A, P: HasDual> HasDual for Send<A, P> {
    type Dual = Recv<A, P::Dual>;
}
```

A mirrored implementation is required for `Recv`.

For the branching constructs, `Offer` and `Choose` we require two subsequent protocols, call them `P` and `Q`, that also implement `HasDual`:

```
impl<P: HasDual, Q: HasDual> HasDual
for Choose<P, Q> {
    type Dual = Offer<P::Dual, Q::Dual>;
}
```

The implementation for `Offer` is analoguous.

Finally, `Rec` is its own dual and is defined as follows:

```
impl<P: HasDual> HasDual for Rec<P> {
    type Dual = Rec<P::Dual>;
}
```

Rust's type system is powerful enough to infer the correct types in many cases and, as we shall see, once a declared protocol is associated with a channel the user rarely has to specify any subsequent protocol types. We can even obtain the dual of a declared protocol without having to manually exchange `Send` and `Recv`, and `Offer` and `Choose`. For example, we can obtain the dual of `Atm` as:

```
type Client = <Atm as HasDual>::Dual;
```

### 3.3 Session-typed Channels

So far we have only described session types as a DSL embedded in Rust, and have not discussed how processes interact over a session-typed connection. In [1] the communication channel is passed in a monad, but this approach does not translate well to Rust, because it requires chaining a lot of closures. The Rust compiler currently does not optimize chains of closures well, which can quickly lead to a stack overflow. Secondly, we found the monadic approach with many closures in Rust to be unergonomic. Instead we provide a `Chan` type which is used to facilitate all session-typed communication in an idiomatic fashion.

The `Chan` type is defined as:

```
struct Chan<E, P> (
    Sender<Box<u8>>,
    Receiver<Box<u8>>,
    PhantomData<(E, P)>
)
```

and contains both a `Sender` and `Receiver` to enable bi-directional communication. The value types `Box<u8>` are placeholders for (pointers to) the actual types transmitted. The `P` type parameter is the session type protocol associated with the channel.

To transmit a value of some type `A` on a `Chan`, we use the `write_chan()` function:

```
fn session_channel<P: HasDual>()
        -> (Chan<(), P>, Chan<(), P::Dual>) {
    let (tx1, rx1) = channel();
    let (tx2, rx2) = channel();
    let c1 = Chan(tx1, rx2, PhantomData);
    let c2 = Chan(tx2, rx1, PhantomData);
    (c1, c2)
}
```

**Figure 5.** Obtaining a session-typed channel

```
    unsafe fn write_chan<A, E, P>
        (&Chan(ref tx, _, _): &Chan<E, P>, x: A)
        where A: marker::Send + 'static
   {
        let tx: &Sender<Box<A>> = transmute(tx);
        tx.send(Box::new(x)).unwrap();
    }
```

First, the `Sender` is borrowed from the `Chan` and the reference transmuted to a boxed version of the type we want to transmit. The `Box` type is a uniquely owned, heap-allocated pointer, and we use it to ensure that all transmitted values have the same size. We then box the value `x` (moving it to the heap) and transmit the pointer.

The corresponding `read_chan()` function looks as follows:

```
    unsafe fn read_chan<A, E, P>
        (&Chan(_, ref rx, _): &Chan<E, P>) -> A
        where A: marker::Send + 'static
    {
        let rx: &Receiver<Box<A>> = transmute(rx);
        *rx.recv().unwrap()
    }
```

Like `write_chan()`, it transmutes a reference to the embedded `Receiver` and receives and unwraps the boxed value. Obviously, these operations are highly unsafe to use, so they are not exposed by the library, but only used internally. In the next section we show how they these unsafe methods are used in a safe way.

Channel types are created in pairs, using the `session_channel()` function. This function constructs two `Chan` structs whose inner `Sender`s and `Receiver`s are connected, and whose protocols are dual. Its declaration can be seen in Figure 5. The `PhantomData` constructor infers the correct type from the context, so the type of `c1` is `Chan<(), P>` and the type of `c2` is `Chan<(), P::Dual>`.

The only thing left to explain is the `E` parameter. In short, it is an environment associated with the channel used to implement recursion. We discuss its usage in details in the following section.

### 3.4 Channel Interaction

Channel interactions are implemented as methods on the session channel end-points. Each action is restricted to only work on channels where the next step in the protocol matches the action. As an example, consider the implementation for `send()` in Figure 6. The `send()` function is only implemented for channels of type `Chan<E, Send<A, P>>`. That is, for channels whose next step in the protocols is `Send`. Calling `send()` consumes the channel value and returns a new channel of type `Chan<E, P>` where the `Send` has been removed. Conceptually we are "taking a step" in the protocol. Rust's move semantics ensures that it is only possible to call `send()` at most once for a given `Send` channel.

We use the `unsafe` keyword here because sending untyped values through a channel is clearly dangerous and Rust cannot verify that it is safe. However, in the context of our session types, we can guarantee that doing so will violate neither protocol nor

```
impl<E, P, A: marker::Send> Chan<E, Send<A, P>> {
    fn send(self, x: A) -> Chan<E, P> {
        unsafe {
            write_chan(&self, x);
            transmute(self)
        }
    }
}
```

**Figure 6.** Implementation of `send()`

```
impl<E, P, Q> Chan<E, Choose<P, Q>> {
    fn sel1(self) -> Chan<E, P> {
        unsafe {
            write_chan(&self, true);
            transmute(self)
        }
    }
    fn sel2(self) -> Chan<E, Q> {
        unsafe {
            write_chan(&self, false);
            transmute(self)
        }
    }
}
```

**Figure 7.** Implementation of `sel1()` and `sel2()`

memory safety. The `write_chan()` method performs the actual untyped send operation: It borrows the `Sender` field of the `Chan`, transmutes it to a `Sender<Box<A>>` (to preserve the type) and sends the given value. The corresponding `recv()` method calls `read_chan()` and receives the pointer of type `Box<A>`. It then deconstructs the box (moving the value onto the stack) and returns the value to the caller.

The branching constructs implicitly communicate the branch choices. For a channel of type `Chan<E, Choose<P, Q>>` two methods are implemented: `sel1()` and `sel2()`. The first method selects the first branch (of type `P`) and returns a channel of type `Chan<E, P>`. The second selects the branch `Q`. Calling either of these transmits the decision through the channel as a boolean: `true` when selecting the first branch and `false` for the second branch. See Figure 7.

There is only one method for a channel of type `Chan<E, Offer<P, Q>>` called `offer()` (see Figure 8). `offer()` returns a `Branch`, which is akin to Haskell's `Either` type and is defined as:

```
enum Branch<L, R> { Left(L), Right(R) }
```

The caller of `offer()` must subsequently match on the return type to figure out which branch was selected and act accordingly.

The *protocol environment* `E` is used to implement recursion. When a `Rec<P>` appears in a protocol, the user must call `enter()`, which pushes the protocol `P` onto the protocol stack, to "enter" the recursive environment in the protocol `P`. Thus, the environment maintains a stack of the bodies of each enclosing `Rec`. The type transformation is:

$$\texttt{Chan<E, Rec<P>>} \longrightarrow \texttt{Chan<(P, E), P>}$$

and is implemented by the `enter()` method.

To repeat a protocol in the stack, we use de Bruijn indices to specify the index of the protocol. The `Var<N>` struct is provided where `N` is the index in the stack. The indices are defined as Peano numbers with the structs `Z` and `S<N>`. Thus, `Var<Z>` signals that we should recurse into the protocol located at the top of the stack,

```
impl<E, P, Q> Chan<E, Offer<P, Q>> {
    fn offer(self) -> Branch<Chan<E, P>,
                             Chan<E, Q>>
    {
        unsafe {
            let b = read_chan(&self);
            if b {
                Branch::Left(transmute(self))
            } else {
                Branch::Right(transmute(self))
            }
        }
    }
}
```

**Figure 8.** Implementation of `offer()`

and `Var<S<Z>>` indicates that we should pop the first element from the stack (changing the type to `Var<Z>`) and then recurse into the protocol left on top of the stack. The type transformations are as follows:

$$\texttt{Chan<(P, E), Var<Z>>} \longrightarrow \texttt{Chan<(P, E), P>}$$

$$\texttt{Chan<(P, E), Var<S<N>>>} \longrightarrow \texttt{Chan<E, Var<N>>}$$

The `zero()` method implements the first transformation, replacing the protocol `Var<Z>` by the protocol `P` located at the top of the stack. The `succ()` method implements the second transformation, removing the protocol from the top of the stack and decrementing the `Var<S<N>>` counter to `Var<N>`. In both cases, the implementation is only available (i.e., they are typeable) for channel types with a non-empty protocol stack.

The functions `enter()`, `zero()` and `succ()` only tranform the protocol type of the channel, they do not send or receive any values. In other words, their implementations only serve to advance the protocol and their method bodies all have the form:

```
unsafe { transmute(self) }
```

## 4. Example: ATM Implementation

To illustrate how to work with session-typed channels, in this section we present an implementation of an ATM program that uses the `Atm` protocol. We also present client programs that interact with the ATM.

### 4.1 ATM

Our ATM implementation is a function `atm()` that accepts a channel with the `Atm` protocol. When the function receives the channel, the session is already initiated, meaning the client is already connected and ready to interact. The ATM then handles the client's requests in accordance with the protocol. The function is shown in Figure 9.

The provided channel `c` has type `Chan<(), Atm>`, which is the type we defined in Section 3.1. The first interaction is to receive the ID provided by the client. Calling `recv()` consumes the channel, and we obtain a tuple containing the new channel with the `Recv` part removed and the ID. For all channel interactions, it is necessary to recapture the channel value because the method call consumes the channel (it is moved out of the calling scope). This pattern lends itself to chaining method calls such as:

```
c.sel2().close();
```

avoiding the need to rebind the `Chan<(), Eps>` channel only to call `close()` immediately.

To iterate, we declare a mutable channel value with:

```
fn atm(c: Chan<(), Atm>) {
    let (c, id) = c.recv();
    if !approved(id) {
        c.sel2().close();
        return;
    }
    let mut c = c.sel1().enter();
    loop {
        c = match c.offer() {
            Left(c) => {
                let (c, amt) = c.recv();
                // update balance
                c.send(new_balance).zero()
            }
            Right(c) => match c.offer() {
                Left(c) => {
                    let (c, amt) = c.recv();
                    if balance >= amt {
                        // adjust balance
                        c.sel1().zero()
                    } else {
                        c.sel2().zero()
                    }
                }
                Right(c) => {
                    c.close();
                    break;
                }
            }
        }
    }
}
```

**Figure 9.** An ATM function

```
c = offer! {
    c,
    Deposit => {
        // Update balance
    },
    Withdraw => {
        // Update balance if possible
    },
    Quit => {
        // Quit
    }
}
```

**Figure 10.** Using `offer!` for the loop body of `atm`

```
    let mut c = c.sel1().enter();
```

which has the type `Chan<(AtmInner, ()), AtmInner>`. In a loop, we then interact according to the `AtmInner` protocol and reassign `c` whenever we reach `Var<Z>`. We only break out of the loop when the user selects the `Quit` branch.

The interaction with the chain of `Offer`s is a little verbose because we must match for each occurrence of `Offer`. We introduced a small syntax extension, via a macro, to Rust called `offer!` that allows chained `Offer` constructs to be matched more succinctly. The loop body could then look as in Figure 10. The labels in `offer!` only provide a useful label for the programmer, they do not carry any semantic meaning. In particular, it is not possible to

```
fn client_deposit(c: Chan<(), Client>) {
    let c = match c.send(id).offer() {
        Left(c)  => c.enter(),
        Right(c) => { c.close(); return }
    };

    let (c, new_bal) = c.sel1().send(100).recv();
    println!("new balance: {}", new_bal);

    c.zero().sel2().sel2().close();
}
```

**Figure 11.** A client depositing an amount

reorder the branches, as they are closely tied to the structure of the chained `match` statements from Figure 9.

### 4.2 ATM Clients

We also want to show how to implement client programs for the ATM example. We first declare type aliases for the protocol and the inner part of the recursion:

```
type Client      = <Atm as HasDual>::Dual;
type ClientInner = <AtmInner as HasDual>::Dual;
```

Our first client program simply wants to deposit an amount to her account and then exit. The client program is shown in Figure 11.

The first action is to send an ID. When the client sends her ID, she must take into account that the ID may not be recognized, which is handled by a call to `offer()`. In case the client is not approved, the only possible action is then to close the channel.

Next, the client selects the `AtmDeposit` branch and sends the amount. After a deposit, the ATM sends the updated balance, which the client then prints. The resulting channel after `recv()` is then:

```
Chan<(ClientInner, ()), Var<Z>>
```

so to repeat the `ClientInner` protocol we call `zero()`.

After printing the updated balance, the client wants to close the connection, which can be done from a channel of type

```
Chan<(ClientInner, ()), ClientInner>
```

by two calls to `sel2()` and then `close()`. Note that the structure of the protocol can result in long chains of method calls to navigate the protocol. This is shown in the last statetement of `client_deposit` (in Figure 11), where the session is closed:

```
c.zero().sel2().sel2().close();
```

Our second client attempts to withdraw an amount from her account. In case the withdraw would result in an overdraft, the client instead deposits another amount. See Figure 12.

### 4.3 Extending the ATM

Suppose we wanted to add another branch to the `Atm` protocol. For example, we could add a branch `AtmBalance` that returns the current balance to the client. We define `AtmBalance` as:

```
type AtmBalance = Send<u64, Var<Z>>;
```

and modify `AtmInner` to include it:

```
type AtmInner = Offer<AtmDeposit,
                Offer<AtmWithdraw,
                Offer<AtmBalance,
                      Quit>>>;
```

Now, our ATM and client implementations need to be modified to take this new branch into account. For the existing clients, the

```
fn client_withdraw(c: Chan<(), Client>) {
    let c = match c.send(id).offer() {
        Left(c)  => c.enter(),
        Right(c) => { c.close(); return }
    };
    let c = match c.sel2().sel1()
                   .send(200).offer() {
        Left(c) => {
            println!("Withdraw OK");
            c.zero()
        }
        Right(c) => {
            println!("Overdraft!");
            let c = c.zero();
            c.sel1().send(50).recv().0.zero()
        }
    };
    c.sel2().sel2().close();
}
```

**Figure 12.** A client withdrawing an amount

```
c = offer! {
    c,
    Deposit => { ... },
    Withdraw => { ... },
    Balance => {
        c.send(balance).zero()
    },
    Quit => { ... }
}
```

**Figure 13.** The body of the `atm()` loop with the added `Balance` branch

modifications are minimal in that they only need to select the correct branch for exiting. The `deposit_client`, for example, would update its last statement to:

```
c.zero().sel2().sel2().sel2().close();
```

to select the `Quit` branch. We have introduced convenience methods to allow skipping through a list of `Choose`. To call `sel2()` three times we could instead use `skip3()`:

```
c.zero().skip3().close();
```

This also demonstrates a limitation in our binary branching constructs: Extending a protocol might require users to take into account parts of the protocol that they are not interested in. We discuss this further in Section 6.4

The `atm()` function must also be updated to handle the new branch. If we use the `offer!` macro (as shown in Figure 10), we can simply add the branch in the right place. See Figure 13. As mentioned earlier, the order of the branches matter, so we must add the `Balance` branch in betweeen `Withdraw` and `Quit`. Session channel interactions must adhere to the structure of the protocol, and even the slightest mistake, like omitting a call to `offer()` results in type errors.

During the course of our work, we have implemented a number of examples from the literature, including an arithmetic server [8], a polygon clipping algorithm [1], and a ticket-ordering system [7]. Our experience so far is, that it is simplest to declare the protocol type upfront and then write the implementation afterwards. This way, the type system aids the user in writing correct programs that does not violate their protocol.

```
pub enum Msg {
    PaintInit(Arc<StackingContext>),
    Paint(Vec<PaintRequest>),
    UnusedBuffer(Vec<Box<LayerBuffer>>),
    PaintPermissionGranted,
    PaintPermissionRevoked,
    Exit(Option<Sender<()>>, PipelineExitType)
}
```

**Figure 14.** The `Msg` type for paint task

## 5. Example: Use in Servo

This section gives a brief introduction to Servo and the concurrency problems we attempt to address with our session types library. We describe the internal communication patterns employed in Servo and how to transform these patterns into session-typed communication.

Servo is a new experimental browser engine developed by Mozilla designed to take advantage of the now ubiquitous multi-core architectures. Existing browser engines showcase excellent sequential performance on web page rendering, but they are not designed to take advantage of multi-core architectures. In Servo, tasks that are traditionally executed sequentially are concurrent, and some of the novel features are concurrent DOM traversal, as well as concurrent layout painting and JavaScript execution.

The Servo project has experienced problems with orchestrating its running tasks, and we sought to address this class of problems using session types. Shutting down in a structured way has proved to be challenging in particular.

In Servo, a task typically listens on one port (a `Receiver`) for incoming messages with the message types defined by an `enum`, the Rust equivalent of tagged unions. Our working example task is the *paint task* from the Servo pipeline. It receives messages of type `Msg` as defined in Figure 14. We found that there are three tasks that send messages to the paint task: The compositor, the constellation and the layout task. We will not discuss these tasks in detail, but refer the reader to [10]. We further found that each variant of the `Msg` enum is only sent by one of these three tasks—except for `Exit`. The division is as follows: The compositor sends `Paint` and `UnusedBuffer`, the constellation sends `PaintPermissionGranted`, `PaintPermissionRevoked` and `Exit` and the layout task sends `PaintInit` and also `Exit`.

We replace the paint task's single channel with three session-typed channels: one for each of the tasks that communicate with it. The variants of the `Msg` can be directly translated to session type "fragments". Variants that contain data are translated to a `Recv` followed by `Var<Z>`. For example, the `PaintInit` variant is translated to:

```
Recv<Arc<StackingContext>, Var<Z>>
```

Variants that do not carry any data, like `PaintPermissionGranted` and `PaintPermissionRevoked` are translated to `Var<Z>`. We defer the discussion of `Exit` for now.

The protocol between the compositor and paint task then looks as follows (from the point of view of the paint task):

```
type Compositor = Rec<
    Offer<Recv<Vec<PaintRequest>, Var<Z>>,
    Offer<Recv<Vec<Box<LayerBuffer>>, Var<Z>>,
          Eps>>>;
```

That is, the compositor has a choice between three branches. The first two each correspond to a variant of the `Msg` type: the first branch corresponds to the `Paint` message, the second to `UnusedBuffer`. We also include a "quit" branch because we want to enforce the principle that all session-typed channels are eventually closed.

The shutdown sequence in Servo is elaborate and has roughly two modes: Either the entire application is shutting down or a single pipeline is being shut down. A "pipeline" in Servo comprises a script task, a layout task and a paint task; a new pipeline is initiated for each page load.

When shutting down a single pipeline, the paint task must collect all outstanding buffers from the compositor before exiting, that is, it waits for `UnusedBuffer` messages and does not exit before all buffers have been returned. The paint task signals on a special port that it has shut down to allow the constellation to wait for it. In a complete shutdown, all outstanding buffers are leaked. This is assumed safe because the memory allocated by the application will be collected by the underlying operating system.

Handling the shutdown sequence with session-typed channels makes this pattern explicit. The channel between the constellation and the paint task has the following branch (from the point of view of the constellation):

```
Choose<Eps, Recv<(), Eps>>
```

In a complete shutdown, the constellation chooses the first branch, closing the connection outright. In a pipeline-only shutdown the second branch is chosen allowing the constellation to wait for acknowlegdment `()` (the unit type in Rust) that the paint task has shut down.

The constellation protocol then looks as follows (again, from the point of view of the paint task):

```
type PaintPermissionGranted = Var<Z>;
type PaintPermissionRevoked = Var<Z>;
type Exit = Choose<Eps, Send<(), Eps>>;

type Constellation = Rec<
    Offer<PaintPermissionGranted,
    Offer<PaintPermissionRevoked,
          Exit>>>;
```

Where we use type aliases to make it more apparent to the casual reader what each branch is.

One of the challenges in Servo is to prevent tasks from dropping connections. We aim to run protocols to completion because it minimizes the risk of crashes. This forced us to rewrite parts of the compositor because it would drop all its paint task connections during a complete shutdown. We changed it to have the compositor close all its connections before exiting. Another challenge we faced in Servo was that the paint task channel (the `Sender`) could be cloned multiple times to be shared with the compositor, and we had to rewrite parts of Servo's pipeline handing to ensure that existing connections between a paint task and the compositor were reused.

In summary, we have managed to translate some of Servo's internal communication patterns to use session types, making the inherent protocols more explicit and systematic. The work is available on GitHub[4].

## 6. Discussion

### 6.1 Performance

Protocol safety is checked at compile time, thus the only *direct* overhead is in branches (transmission of an additional $\log(n)$ boolean values for $n$ different protocols) because of the additional boxing during send and receive, and from small wrapper functions like `send()`, which could be inlined by the compiler. The overhead from boxing could be removed by writing our own specialized

[4] `https://github.com/laumann/servo/tree/session-types`

implementation of untyped channels, while the extra transmission of boolean values is harder to fix without a different approach to branching.

However, like with all static type disciplines, the point of the type system is to disallow certain programs, and some of these programs could be correct and well performing. Thus, it might be that the library imposes an *indirect* overhead.

### 6.2 Safety

We are trying to achieve the static guarantee that two processes are *compatible* "*in the sense that an interaction between the two will not terminate prematurely because of a mismatch in the expectations of one of the partners*" [9].

We will not give a formal proof for our claims, partly because Rust has no formal specification, but we will argue for our claims in terms of Rust's informal semantics. We focus on our library-provided methods and functions, treating `session-types` as an embedded DSL for dealing with sessions. Later we discuss what effect certain Rust functions, like `drop()`, has on our claims.

It is important to note that we are arguing from *safe* Rust code. Rust, as a systems programming language, gives you the foot-gun, but it is clearly labelled `unsafe`. In safe Rust we are guaranteed freedom from dangling pointers, invalid pointer dereferencing, double frees and other similar problems. In unsafe Rust such errors can still be created, and we have already seen the `transmute()` function for type coercion. The only restriction on transmuting a value from one type to another is that the sizes must match. But it allows you to, for example, re-interpret an array of four bytes as a 32-bit number:

```
let a: [u8; 4] = [0u8, 0u8, 0u8, 0u8];
let n: u32 = unsafe {
    transmute::<[u8; 4], u32>(a)
};
```

All our protocols are zero-sized phantom types, and we cannot prevent the user from transmuting an obtained `Chan` value to specify some other (potentially incompatible) protocol. But this cannot be done without marking the operation as `unsafe`. For the same reason, the `HasDual` trait is marked `unsafe`: We cannot prevent users from implementing `HasDual` for custom data types, but we *can* indicate that doing so potentially violates protocol safety.

To argue safety, we demonstrate that for any two processes communicating over a session-typed channel with end-points $c_1$ and $c_2$, the associated session types $P_1$ and $P_2$ are compatible (i.e., they are dual).

The only way to obtain a a `Chan` value is through the `session_channel()` function, which returns two `Chan` end-points of a new session-typed channel. The restriction on the type parameter `P` is that it must implement `HasDual`. Thus, upon session initialization, the protocols $P_1$ and $P_2$ of the channel end-points are dual.

We sketch an inductive proof on the structure of the session types DSL, arguing that any typeable protocol $P_1$ for a channel $c_1$ has a corresponding, unique dual $P_2$ that is the protocol type of the other channel end-point $c_2$. Our base case is `Eps`:

`Eps` $P_1$ = `Eps`. By definition $P_2$ = `Eps`.

We must then argue that any interaction on a session-typed channel cannot violate protocol safety, in other words, we cannot arrive at a state where the protocols are *not* dual. We argue in terms of the methods that transmit values over the internal channels, which are `send()`, `sel1()` and `sel2()`.

`send()` Assume `P` and `Q` are dual protocol types and `A` is any type. Let $P_1$ = `Send<A, P>`. By definition $P_2$ = `Recv<A, Q>`. The only interaction available on $c_1$ is `send()` which consumes the channel, transmits a provided value of type `A` and yields a new channel $c_1'$ with protocol type `P`. The only operation available on $c_2$ is `recv()` which consumes $c_2$, receives a value of type `A` and returns a tuple containing a new channel $c_2'$ and the received value. The protocol type of $c_2'$ is `Q`.

`sel1()` Assume `P` and `R` are dual, and `Q` and `S` are dual. Let $P_1$ = `Choose<P, Q>`. By definition $P_2$ = `Offer<R, S>`. Calling `sel1()` on $c_1$ transmits the boolean value `true`, consumes $c_1$ and yields $c_1'$ with protocol $P_1'$ = `P`. The only method available on $c_2$ is `offer()`, which receives a boolean value. When `true` is received $c_2$ is consumed, yielding $c_2'$ with protocol $P_2'$ = `R`. $c_2'$ is returned from `offer()` wrapped in an `Ok` value.

`sel2()` The argument for `sel2()` is the same as for `sel1()`, except `false` is transmitted and the resulting protocol types are $P_1'$ = `Q` and $P_2'$ = `S`, and `offer()` returns the new channel $c_2'$ wrapped in an `Err` value.

For completeness we should also show that non-transmitting methods, such as `enter()`, `zero()` and `succ()` cannot introduce incompatible protocols. The proof sketch is similar, but we must also take the protocol stack (the environment) into account.

### 6.3 Affine vs Linear Types

A challenge presented by the use of affine types is that we cannot prevent a channel value from being dropped prematurely. We argue that any interaction that takes place over a session-typed channel is safe under the condition that none of the interacting parties drop their channel prematurely. Linear types prevent this exact problem and there are proposals to extend Rust with linear types, but this work has at the time of writing been postponed.[5] So we have considered alternative ways to enforce linear usage of channel values.

Currently, Rust supports an annotation on types and functions called `#[must_use]`, that signifies that a value of this type must be used. By default, a compiler lint produces warnings on `must_use` types when they are not used. The lint can also be instructed to reject programs that do not use their values. As an example, the `Result` type is a `must_use` type to hint to the programmer that the result of a function that might produce errors should not be ignored (i.e., you should handle error situations). However, `#[must_use]` is not quite sufficient for our purposes, as it is easy to accidentally and purposely bypass.

The Rust compiler exposes an extensive plugin architecture and in collaboration with Manish Goregaokar we have developed a compiler plugin, `humpty_dumpty`, which provides a lint that ensures that types annotated with `#[drop_protect]` are used linearly.[6] These values cannot be dropped except by functions explicitly annotated with `#[allow_drop]`. It is still a work in progress, which by no means guarantees linearity in all cases, but it provides better security than `#[must_use]`.

### 6.4 Binary vs Labelled Branches

Although we can express the same protocols using binary branches as we can with labelled, multi-way branches, binary branches are less flexible and more cumbersome to work with. When using binary branches, the ordering of the branches in an `Offer` matters: Changing the order of the branches in one side of the session type requires making similar changes to the other side as well. In contrast, the branches in a labelled branching construct can be freely moved around because the order does not matter. Additionally, the labels themselves makes the session types easier to read and reason about: Any user seeing a branch labelled `Quit` will have a

[5] `https://github.com/rust-lang/rfcs/pull/776`

[6] `https://github.com/Manishearth/humpty_dumpty`

pretty good idea what the branch is supposed to do. Finally, using binary branches means that you have to nest branches in order to express choices between more than two options. Multi-way branching, which most implementations of labelled branches support, means that you can have an arbitrary number of choices in each branch.

We chose to use binary branches for practical reasons. Although most users would probably find labelled branches superior to binary branches, we have not found a good way to implement them in Rust without resorting to language extensions or the like. Instead, we opted to make working with binary branches more convenient by introducing the `offer!` macro and the `skip` functions.

Some implementations of session types using labelled branching also support a sophisticated kind of subtyping where a choice ($\oplus$) with labels $L = l_1, l_2, \ldots, l_n$ is the dual of an offer ($\&$) with labels $L' = l'_1, l'_2, \ldots, l'_n$ as long as $L$ is a subset of $L'$. This means that you can update the offer with new branches without altering the choice. This is a helpful feature, which makes labelled branches very flexible to work with.

Our library also supports a kind of subtyping. For instance, if a client is using only one part of a binary branch, it can ignore the other part and replace its type with a type parameter. By doing so, we can change the other part of the branch on the server side without changing the client side at all. For example, consider the following functions:

```
fn server(c: Chan<(), Offer<Recv<u8, Eps>,
                            Recv<i8, Eps>>>)
{
    ...
}

fn client<P>(c: Chan<(), Choose<Send<u8, Eps>,
                                P>>)
{
    c.sel1().send(42u8).close();
}
```

Here, the session types in `client` and `server` are dual, even though we have used a type parameter in the session type for `client`. However, we can only do this if `client` never uses the other branch of the `Choose`, as the case is here. Like subtyping in labelled branches, this allows for more flexibility, and makes it easier to write several clients that talk to the same server, for example.

Although binary branches are somewhat limited compared to labelled branches, we feel that we have found a good compromise. The current implementation is simple, easy to understand, and although it is limited to binary branches, it is no less powerful in terms of possible interaction patterns. We will continue to consider ways to further improve our binary branches, or even implement labelled branching, but we are convinced that the current library is easy to use and understand for most users.

## 7. Related Work

The syntactical constructs provided in our session types library are a direct translation of the constructs presented in [1].

The idea of associating the protocol with a channel value was inspired by the SessionJ language that implements session types as an extension to Java [7].

## 8. Conclusion

We have demonstrated that session types can be implemented directly in a language supporting affine types, and argued for its safety. Like linear types, affine types prevent aliasing, but, unlike linear types, fail to promise progress.

We have developed a library that is available on GitHub[7] and shown it is practically usable, through both examples and inclusion in real-world applications.

## Acknowledgments

Thanks to Lars Bergstrom and Manish Goregaokar who have provided lots of feedback on the work presented here.

## References

[1] Riccardo Pucella and Jesse A. Tov. Haskell Session Types with (Almost) No Class. *SIGPLAN Not.*,44(2):25–36, September 2008

[2] Kaku Takeuchi, Kohei Honda and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE '94: Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 398–413, London, UK, 1994. Springer-Verlag.

[3] Kohei Honda, Vasco Thudichum Vasconcelos and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, pages 122–138, London, UK, 1998. Springer-Verlag.

[4] The Rust Programming Language. `http://doc.rust-lang.org/book`

[5] The Rust Reference. `http://doc.rust-lang.org/reference.html`

[6] Eric Reed. Patina: A Formalization of the Rust Programming Language. `ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf`, February 2015.

[7] Raymond Hu, Nobuko Yoshida and Kohei Honda. Session-Based Distributed Programming in Java. In *ECOOP 2008 Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 516-541, Springer Berlin Heidelberg, 2008.

[8] Simon Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In Swierstra, S.Doaitse, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer Berline Heidelberg, 1999.

[9] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):1950, 2010.

[10] Philip Munksgaard and Thomas Bracht Laumann Jespersen. Practical Session Types in Rust. Master's thesis, Department of Computer Science, University of Copenhagen, June 2015.

[7] `http://github.com/Munksgaard/session-types`