Rusty Variation
Deadlock-free Sessions with Failure in Rust

Wen Kokke
LFCS, University of Edinburgh, 10 Crichton St, EH8 9AB, Edinburgh, United Kingdom
wen.kokke@ed.ac.uk

Rusty Variation (RV) is a library for session-typed communication in Rust which offers strong compile-time correctness guarantees. Programs written using RV are guaranteed to respect a specified protocol, and are guaranteed to be free from deadlocks and races.

1 Introduction

In concurrent programming, processes commonly exchange information by sending messages on shared channels. However, this communication often does not follow a specified protocol, and if it does, the correctness of the implementation with respect to that protocol is rarely checked.

Session types are a substructural typing discipline, introduced by Honda [8], which describe communication protocols as types, capturing both the types and the order of messages. Session type systems, then, ensure that programs correctly implement specified protocols.

Rust is an immensely popular systems programming language.[1] It aims to combine efficiency with abstraction and thread and memory safety. However, out of the box, it does not offer a way to specify protocols and verify communication.

In this paper, we present Rusty Variation[2] (RV), a library for session-typed communication in Rust which allows users to specify and check the adherence to communication protocols, in addition to offering strong correctness guarantees, such as freedom from deadlocks and unwanted races, while accounting for the possibility of failure. It is these strong guarantees of correctness, together with our account of failure, which sets us apart from previous work [9]. We assume some familiarity with Rust. For an introduction to Rust, we refer the reader to *The Rust Programming Language* [3].

We base our work on Exceptional GV by Fowler *et al*. [5, EGV]. Exceptional GV is a descendant of Good Variation [14, GV], which adds support for exceptions and the cancellation of sessions. GV is a λ-calculus with operations for forking threads and sending and receiving values. Crucially, EGV has an affine type system, and permits the explicit cancellation of sessions. EGV guarantees session fidelity, global progress, freedom from deadlocks and livelocks, confluence, and termination. That is to say, there are no locks or race conditions, and all programs eventually halt. For a discussion of EGV and its metatheory, we refer the reader to the work by Fowler *et al*. [5].

Rusty Variation is an implementation of Exceptional GV (EGV) in Rust. We claim that RV is a faithful implementation of EGV, preserving much of EGV's metatheory, meaning RV guarantees session fidelity, global progress, confluence, and freedom from deadlocks and unwanted races. However, we depart from EGV in two ways: First, EGV relies upon exceptions and handlers. Rust, however, rejects exceptions in favour of monadic errors, *i.e*., wrapping possibly failing results in error types such as

[1] https://insights.stackoverflow.com/survey/2019/
[2] https://github.com/wenkokke/rusty-variation/

To appear in EPTCS.

`Option<T>` and `Result<T,E>`. Second, the core language of EGV does not include general recursion. Rust does. In the presence of non-termination, we cannot guarantee livelock freedom, or—well—termination.

RV is currently implemented on top of the channels provided by the `crossbeam_channel` crate, but it can be implemented using any library which offers one-shot channels and selection.

The paper proceeds as follows. In section 2, we describe the programming interface offered by Rusty Variation, and provide several example programs. In section 3, we compare Rusty Variation to the related work by Laumann *et al*. [9]. Finally, in section 4, we describe future work on verifying the correctness guarantees offered by RV.

2 Rusty Variation

2.1 Types, sessions, and duality

There are three basic types which we use to describe session protocols—or *session types*: `Send`, `Recv`, and `End`. We refer to values of these types as *session endpoints* or, when referring to both endpoints, *sessions*. Session endpoints wrap primitive channels—see section 2.6. Any type built using these primitive types is a valid session type:

```
pub struct Send<T, S: Session> { ... }
pub struct Recv<T, S: Session> { ... }
pub struct End { ... }
```

Session types are public, but their member functions are not. Therefore, users cannot construct their own misbehaving sessions. The continuation of a session must itself be a session. This is enforced by the `Session` constraint. In addition to this "kinding" of session types, the `Session` trait also requires any session type to implement duality and session generation:

```
pub trait Session {
  type Dual: Session<Dual=Self>;
  fn new() -> (Self, Self::Dual);
}
```

Duality works as expected: `Send<T, S>` is dual to `Recv<T, S::Dual>`—and vice versa—and `End` is self-dual. The constraint `Session<Dual=Self>` enforces that the dual of a session type is a session type, and that `S::Dual::Dual` is equal to `S` (involutivity).

The `new` function generates a new session-typed channel and returns two *dual* channel endpoints. Unlike the rest of the functions discussed in this section, `new` is *not* part of the public interface—see section 2.7.

Using these types, we can define the types of servers which offer squaring or negation as follows, where `i32` is the type of 32-bit integers:

```
type SqrSrv = Recv<i32, Send<i32, End>>;
type NegSrv = Recv<i32, Send<i32, End>>;
```

The client types are obtained by duality.

2.2 Send, receive, close, and fork

The `send` and `recv` functions send and receive values in sessions. Sending is non-blocking, and never fails. Sending a value on a cancelled channel is equivalent to cancelling that value—see section 2.3. Receiving is blocking, and fails if the channel is cancelled.

```
pub fn send<T, S: Session>(x: T, s: Send<T, S>) -> S
pub fn recv<T, S: Session>(s: Recv<T, S) -> Option<(T, S)>
```

The `close` function closes a *completed* session, *i.e.*, of type `End`. Closing is *synchronous*, meaning that `close` blocks until the dual end point is also closed. Closing a cancelled channel fails.

```
pub fn close(s: End) -> Option<()>
```

The `fork` function creates a new session, and spawns a new thread. The new thread runs the first argument of `fork`—a function—which receives one endpoint, and `fork` returns the dual endpoint. It has the following type—where `FnOnce(S) -> Option<()>` is the type of functions from `S` to `Option<()>` which can be invoked only once:

```
pub fn fork<S: Session>(p: P) -> S::Dual
where
  P: FnOnce(S) -> Option<()>,
```

The function passed to `fork` is allowed to fail, hence the `Option`-type. If it fails, the child thread panics silently, and all open sessions in the child thread are cancelled.

Now that we have covered the basics, let us consider an example program. The following program forks off a child process which sends a "ping", then waits to receive that ping and returns:

```
let s = fork(move |s: Send<(), End>| {
  let s = send((), s);
  close(s)
});
let ((), s) = recv(s)?;
close(s)
```

The `let x = M?; N` construct is Rust's "monadic bind" notation for programs which may return errors. If `recv(s)` succeeds, the `?`-operator unpacks the `Option`. If `recv(s)` fails, the `?`-operator short-circuits, skips the rest of the statements, and returns the error.

2.3 Linearity and cancellation

Session types often require linearity—the protocol for a negation server in section 2.2 specifies that the server receives *one* integer, and then sends *one* integer back. For such a server to, *e.g.*, send back multiple integers would be, at best, confusing, and at worst, cause an error. The following two erroneous programs violate the linearity condition. The program on the left reuses the channel s, and the program on the right drops it without completing the session:

```
let s = fork(move |s: Send<(), End>| {
  let s1 = send((), s);
  close(s1)?;
  let s2 = send((), s); // reuse s
  close(s2)
});
let ((), s) = recv(s)?;
close(s)
```

```
let s = fork(move |s: Send<(), End>| {

  // unintentionally
  // left blank.

});
let ((), s) = recv(s)?;
close(s)
```

Rust is an affine language, meaning that any value can be used *at most once*. Hence, our first example doesn't compile: Rust complains about the second usage of `s`, which attempts to use `s` after transferring ownership of the value to `send`.

The second program is trickier, as Rust's type system doesn't help us here. Arguably, however, that is for the best. The idea of *linear* session types ignores the reality that sessions may be dropped at any time, for any number of reasons: a process may panic, dropping all sessions in scope, or—when communicating over the network—the connection might fail. In section 1, we mentioned that EGV has an affine type system, and permits the explicit cancellation of sessions. Following EGV, Rusty Variation also permits the cancellation of sessions. However, instead of having cancellation be an explicit primitive, we implement it implicitly for any dropped value, using Rust's destructors[3].

When an initialised value in Rust goes out of scope, Rust calls its destructor, the `drop` method. This method can be customised by implementing the `Drop` trait. For instance, if we use the implementation of `Drop` shown on the left, running the code on the right would print "End dropped!" twice, once from the child thread, and once from the main thread.

```
impl Drop for End {
  fn drop(&mut self) {
    println!("End dropped!")
  }
}
```

```
let _s = fork(move |_s: End| {
  // intentionally left blank.
});
// intentionally left blank.
```

To mimic the cancellation behaviour in EGV, we count the number of references to a session, starting at *two* when a session is created using `new`. Each time a session's destructor is called, we decrement this counter. If the counter is *one*, the session is marked as *disconnected*. Sending on a disconnected session drops the value being sent. Receiving on or closing disconnected sessions returns an error, *i.e.*, `None`. This ensures that no process is left waiting for a message that will never come. If the counter reaches *zero*, the memory for the session is deallocated.

As it turns out, this is exactly the behaviour for the channels from `crossbeam_channel`, which we use in our implementation, and when a session is dropped, the channel it wraps is dropped recusively. Hence, we inherit the correct behaviour of dropping sessions from `crossbeam_channel`.

However, there is one major downside to replacing explicit cancellation with implicit cancellation: it is no longer possible to tell whether the programmer wanted to cancel a session or simply forgot to complete the session. We do two things to remedy this. First, the definitions of `Send`, `Recv`, and `End` are annotated with `#[must_use]`. This causes Rust to emit a warning whenever a session is dropped. Second, we provide an explicit `cancel` function, though its implementation is simply to drop its argument:

```
pub fn cancel<T>(_x: T) -> () { /* intentionally left blank. */ }
```

The following two programs illustrate our solution. The program on the left forgets to use `s`, and Rust emits a warning, complaining that `s` isn't used. The program on the right passes the session to `cancel`, explicitly marking the drop, and hence compiles without any warnings. Both programs have the same semantics, though: the program forks off a process which is *expected* to send a "ping". Instead the forked process cancels the session—be it implicitly or explicitly. Hence, the call to `recv` fails, the `?`-operator short-circuits, and the whole program returns `None`:

```
let s = fork(move |s: Send<(), End>| {
  // shouldn't we be using s?
  Some(())
});
let (z, s) = recv(s)?;
close(s)
```

```
let s = fork(move |s: Send<(), End>| {
  cancel(s);
  Some(())
});
let (z, s) = recv(s)?;
close(s)
```

[3] `https://doc.rust-lang.org/1.35.0/book/ch15-03-drop.html`

2.4 Branching with offer! and choose!

The `choose_left`, `choose_right`, and `offer_either` functions let processes offer or make a *binary* choice. These are *derived* constructs [8, 2]. They are implemented using only the functions in our library—`new`, `send`, `recv`, `close`, and `cancel`—and the sum type `Either`. They send or receive a value of the type `Either<S1, S2>`, where `S1` and `S2` are the possible continuations of the session.

```
type Choose<S1, S2> = Send<Either<S1, S2>, End>;
type Offer<S1, S2> = Recv<Either<S1, S2>, End>;

pub fn choose_left<S1: Session, S2: Session>(s: Choose<S1, S2>) -> S1
pub fn choose_right<S1: Session, S2: Session>(s: Choose<S1, S2>) -> S2
pub fn offer_either<S1: Session, S2: Session, P1, P2, R>(
       s: Offer<S1, S2>, p1: P1, p2: P2) -> Option<R>
where
  P1: FnOnce(S1) -> Option<R>,
  P2: FnOnce(S2) -> Option<R>,
```

There is a subtlety in their implementation. The `choose_left` function creates a new session of type `S1`. It wraps one endpoint in `Either::Left`, sends it over the original session `s`, and returns the other endpoint. However, this leaves it with the continuation of the original session, of type `End`. Closing this endpoint synchronously, using `close`, may fail. However, `choose_left` *sends*, and for consistency with `send`, we would like it to always succeed. Therefore, we opt to *cancel* the continuation instead:

```
pub fn choose_left<S1: Session, S2: Session>(s: Choose<S1, S2>) -> S1 {
    let (here, there) = S1::new();
    let s = send(Either::Left(there), s);
    cancel(s);
    here
}
```

Cancelling the endpoint is harmless—as long as the other processes is prepared, and uses the corresponding function `offer_either`, this acts as an *asynchronous* alternative to close. Pairing `choose_left` with a manually implemented version of `offer_either` which attempts to *close* the continuation instead of cancelling it results in a runtime error.

The `choose_left`, `choose_right`, and `offer_either` functions suffice to encode choice, they are not useful in practice—a protocol with a four-way choice would require three nested `Offer` constructs, and several calls to `offer_either`. Code written in this style rapidly becomes unreadable. For this reason, we also define two macros, `choose!` and `offer!`, which generalise binary choice using `Either` to a choice on *any* **`enum`**, as long as each case wraps a session. This allows processes to offer and make choices between many *labelled* branches, and allows us to define the type for a calculator server as:

```
enum CalcOp { Sqr(SqrSrv), Neg(NegSrv) }
type CalcSrv = Recv<CalcOp, End>;
type CalcCli = CalcSrv::Dual;
```

Using the `choose!` and `offer!` macros, we can implement a calculator server and client. The `main` function below forks off a calculator server, which offers the choice between squaring and negation. The client then selects squaring, sends the number four, and prints the result:

```
fn server(s: CalcSrv) -> Option<()> {
  offer!(s, {
    CalcOp::Sqr(s) => {
      let (x, s) = recv(s)?;
      let s = send(x * x, s);
      close(s)
    },
    CalcOp::Neg(s) => {
      let (x, s) = recv(s)?;
      let s = send(-x, s);
      close(s)
    },
  })
}
```

```
fn client(s: CalcCli) -> Option<i32> {
  let s = choose!(CalcOp::Sqr, s);
  let s = send(4, s);
  let (z, s) = recv(s)?;
  close(s)?;
  Some(z)
}

// main() prints Some(16)
fn main() {
  let s = fork(server);
  let z = client(s);
  println!("{}", z);
}
```

2.5 Homogeneous Selection

Selection is a mechanism which allows processes to block on a list of actions, waiting until the first of them fires. Writing programs which use selection can be tricky and error prone, even using libraries specifically built to help with selection. For instance, the following is an excerpt from the documentation of `crossbeam_channel::Select`[4], a construct which implements a selection mechanism:

> Once a list of operations has been built with `Select`, there are two different ways of proceeding:
>
> - Select an operation with `try_select`, `select`, or `select_timeout`. If successful, the returned selected operation has already begun and *must* be completed. If we don't complete it, a panic will occur.
> - Wait for an operation to become ready with `try_ready`, `ready`, or `ready_timeout`. If successful, we may attempt to execute the operation, but are not obliged to. In fact, it's possible for another thread to make the operation not ready just before we try executing it, so it's wise to use a retry loop. However, note that these methods might return with success spuriously, so it's a good idea to always double check if the operation is really ready.

None of this advice is enforced by Rust—instead, this must be checked by the programmer—and bugs in the use of selection can be hard to find, especially if the approach using `ready` is used.

RV offers an alternative, the `select_mut` function. This function operates on a vector of *receiving* endpoints, returning the result of the first receive operation which completes, and removing the corresponding session from the vector. We restrict ourselves to the receiving endpoints, as RV's `send` operation is non-blocking, hence selecting on sending endpoints doesn't make any sense.

```
pub fn select_mut<T, S: Session>(rs: &mut Vec<Recv<T, S>>) -> Option<(T, S)>
```

The `select_mut` function is implemented on top of `crossbeam_channel::Select`, in a way which respects the rules listed above. In addition, RV offers an immutable variant of `select_mut`, named `select`, which returns the remainder of the endpoints along the with result of the selection. Using `select_mut`, we can write the following program, which forks off ten threads, each sending an index between 1 and

[4] `https://docs.rs/crossbeam-channel/0.3.8/crossbeam_channel/struct.Select.html`

10, and then repeatedly selects on the receiving endpoints, receiving and printing an index, printing the numbers 1 to 10 in a random order:

```
let mut rs = Vec::new();
for i in 0..10 {
  let s = fork(move |s: Send<i32, End>| {
    let s = send(i, s);
    close(s)
  });
  rs.push(s);
}
```

```
loop {
  if rs.is_empty() {
    break Ok(());
  }
  let (i, r) = select_mut(&mut rs)?;
  println!("{}", i);
  close(r)
}
```

The `select_mut` and `select` functions are homogeneous: they only select over vectors of endpoints with *the same type*. This rules out heterogeneous uses of selection, where the operations are of different types. Selection is not a part of EGV, and hence our strong correctness guarantees do not extend to selection.

2.6 Implementation

So far we have avoided discussing the implementation whenever possible, and with good reason: RV can be implemented using *any* library which offers one-shot channels and selection. Hence, the important contribution in this paper is not the implementation, but the programming interface offered by RV.

Let us briefly discuss the implementation. We use the asynchronous channels from `crossbeam_channel`, which at the time of writing is the de facto implementation of channels in Rust. Channels are typed, and can only transfer values of a single type. Therefore, we use one-shot channels, and encode sessions following Scalas and Yoshida, and Padovani [13, 12]: Each `send` creates a new session for the continuation, and uses the primitive `send` operation to send the value along with one endpoint of the continuation session. If the primitive `send` operation returns an error, we swallow it, using `unwrap_or`. Each `recv` receives a value together with a channel on which to continue the session. If the primitive `recv` operation returns an error, the `?`-operator short-circuits, and we return the error. The channels in `crossbeam_channel` are highly optimised for one-shot usage.

```
use crossbeam_channel::{Sender, Receiver}; // abbreviated import list

pub struct Send<T, S: Session> { channel: Sender<(T, S::Dual)> }
pub struct Recv<T, S: Session> { channel: Receiver<(T, S)> }

pub fn send<T, S: Session>(x: T, s: Send<T, S>) -> S
{
    let (here, there) = S::new();
    s.channel.send((x, there)).unwrap_or(());
    here
}

pub fn recv<T, S: Session>(s: Recv<T, S>) -> Option<(T, S)>
{
    let (v, s) = s.channel.recv()?;
    Some((v, s))
}
```

Closing is implemented using two asynchronous channels, which we use to simulate a single synchronous channel. Both channels transmit a unit value. Each close operator first sends on one of the channels (for which it has the sender) and then receives (blocking) on the other channel (for which it has the receiver). If the other endpoint is cancelled, both channels are marked as disconnected, and thus any close operation which is still blocking will raise an exception. Otherwise, the first close operation blocks until the second is executed.

```
pub struct End { sender: Sender<()>, receiver: Receiver<()> }

pub fn close(s: End) -> Option<()> {
    s.sender.send(()).unwrap_or(());
    s.receiver.recv()?;
    Some(())
}
```

While the current implementation is efficient, it could likely be improved by replacing `crossbeam_channel` with a minimal implementation of one-shot channels and selection.

The Rust standard library offers `std::sync::mpsc`, an implementation of channels. The first implementation of RV used the channels from this library. However, `std::sync::mpsc` has some fundamental issues with its design[5], and seems to be on its way out. In 2018, the selection mechanism, `mpsc_select!`, was deprecated[6], and there are plans to phase out `std::sync::mpsc` altogether [7].

2.7 What If I Want to Be Evil?

There are three ways in which you can make code written using Rusty Variation lock in safe Rust. The first is by writing an infinite loop, which we discussed earlier.

The biggest issue lies in the fact that Rust's semantics do not guarantee that the destructors are actually run. This does not mean that Rust can *arbitrarily* decide not to run destructors, but that it is possible to write programs which leak memory. The most obvious example of this is `std::mem::forget`[7]:

> Takes ownership and "forgets" about the value *without running its destructor*.
> Any resources the value manages, such as heap memory or a file handle, will linger forever in an unreachable state. However, it does not guarantee that pointers to this memory will remain valid.

Explicitly calling `std::mem::forget` on session endpoints is unlikely to happen by accident, hence we don't consider this a huge shortcoming. However, it is also possible to leak memory by creating reference cycles[8]—in fact, `std::mem::forget` was initially marked as **`unsafe`**, until it was discovered that it could be implemented in safe Rust creating reference cycles. This means that care needs to be taken when combining sessions and cyclic data structures, *e.g.*, storing session endpoints in doubly linked lists.

The second problem arises from the availability of `new`, which creates two dual session endpoints. Code written using only `fork` is guaranteed to be deadlock free [11], but using `new` and `std::thread::spawn`, it becomes easy to write code which deadlocks. However, it is impossible to export a trait without exporting all its members. Arguably, hiding `new` is also undesirable. Using only `fork`, it is impossible to create any cyclic communication structures, including those which are well behaved. Therefore, a programmer may occasionally *need* access to `new`.

[5] `https://github.com/rust-lang/rust/pull/42397#issuecomment-315867774`
[6] `https://github.com/rust-lang/rust/issues/27800`
[7] `https://doc.rust-lang.org/1.35.0/std/mem/fn.forget.html`
[8] `https://doc.rust-lang.org/1.35.0/book/ch15-06-reference-cycles.html`

The following two programs illustrate the two ways of constructing deadlocking programs discussed above. The program on the left uses `std::mem::forget` to drop the session endpoint without running its destructors. The program on the right creates two pairs of channels to construct a deadlock:

```
// Deadlock using std::mem::forget
let s = fork(move |s: Send<(), End>| {
  std::mem::forget(s);
  Some(())
});
let ((), s) = recv(s)?;
close(s)
```

```
// Deadlock using new and spawn
let (s1, r1) = Send<Void, End>::new();
let (s2, r2) = Send<Void, End>::new();
std::thread::spawn(move || {
  let (v, r1) = recv(r1)?;
  close(r1)?;
  let s2 = send(v, s2);
  close(s2)
});
let (v, r2) = recv(r2)?;
close(r2)?;
let s1 = send(v, s1);
close(s1)
```

3 Related Work

Laumann *et al*. [9] implement session types in Rust, based on the session-typed functional language LAST by Gay and Vasconcelos [6]. Their implementation guarantees session fidelity, albeit with some caveats. LAST is a linear language, meaning that sessions can neither be duplicated nor dropped. As we discussed in section 2.3, Rust is an affine language, meaning that values—and, particularly relevant here, sessions—can be dropped. The implementation of LAST in Rust, therefore, guarantees session fidelity under the assumption that channels are *never dropped*.

Laumann *et al*. [9] remark that they "cannot prevent a channel value from being dropped prematurely". They offer several solutions, but remark that each of these solutions are either insufficient, premature, or both. They note that the proposal to add linear types to Rust is currently postponed[9], and that while `#[must_use]` can be made to issue a warning if a value is dropped, it is "easy to accidentally and purposely bypass". Finally, they propose a compiler extension, developed in collaboration with Manish Goregaokar, `humpty_dumpty`[10], "which provides a lint that ensures that types annotated with `#[drop_protect]` are used linearly". However, they note that `humpty_dumpty` "is still a work in progress, which by no means guarantees linearity in all cases". No further work has been done on `humpty_dumpty` since 2015. The current version of the library enforces linearity *dynamically*, using "destructor bombs", implemented by Manish Goregaokar[11]. This technique hides a `panic!` in channel destructors. Each invocation of `close` then calls `std::mem::forget`, bypassing the destructor bomb.

We argue that even if Rust were able to enforce linearity, the *idea* of linear session types is flawed (see section 2.3), as sessions may be dropped at any time, for any number of reasons, *e.g*., a process panicking, dropping all sessions in scope.

The following two programs illustrate the difference in handling failure between Rusty Variation and the work by Laumann *et al*. [9]. In both programs, the programmer has forgotten to complete the server, accidentally dropping the session endpoint `_s` or channel `_c`, respectively. The program on the left, written in RV, returns an error. The program on the right, written using the library described by

[9] `https://github.com/rust-lang/rfcs/pull/776`
[10] `https://github.com/Manishearth/humpty_dumpty`
[11] `https://github.com/Munksgaard/session-types/commit/0f25ccb7c3bc9f65fa8eaf538233e8fe344a189a`

Laumann *et al*. [9], segfaults. We have since reported this segfault to Laumann *et al*., and they have fixed it in the most recent version of the library. The program on the right now panics:

```
// Rusty Variation
// main() returns None
fn server(_s: Recv<(), End>) {
  // unintentionally left blank.
}
fn client(s: Send<(), End>) {
  close(send((), s))
}
fn main() -> Option<()> {
  client(fork(server))
}
```

```
// Laumann et al.
// main() causes segfault
fn server(_c: Chan<(), Recv<(), Eps>>) {
  // unintentionally left blank.
}
fn client(c: Chan<(), Send<(), Eps>>) {
  c.send(()).close();
}
fn main() -> () {
  connect(server, client);
}
```

Another feature which distinguishes Rusty Variation from the work by Laumann *et al*. [9] is in how both libraries generalise binary choice to labelled choice. Rusty Variation uses enumerations to provide labelled choice, as described in section 2.4. Laumann *et al*. [9] provide an `offer!` macro, which expands a labelled offer to a series of binary offers, *e.g*., offering a ternary choice expands to two nested invocations of `offer_either`.

The `offer!` macro allows—even requires—the programmer to label their choices, but during macro expansion these labels are simply discarded *without being checked*. This allows us to write the program below, in which an inconsistency between types and documentation on the one hand, and the labelled implementation on the other hand, results in a bug. The example program implements a calculator server, which offers the choice between squaring and negation, in that order.

In the `server` function, the order in which the `NEG`-case and the `SQR`-case are handled doesn't match the order in which they are declared in the types. However, the code compiles, not because the cases are associated correctly through the use of labels, but because the labels aren't checked, and squaring and negation happen to have the same type signature.

When the client selects the first option, they wrongly assume it represents squaring, based on the types. Unfortunately, it represents negation. The `main` program, when run, therefore execute negation, and erroneously returns `-42`:

```
// Laumann et al.
type SqrSrv  = Recv<i32, Send<i32, Eps>>;
type NegSrv  = Recv<i32, Send<i32, Eps>>;
type CalcSrv = Offer<SqrSrv, NegSrv>;

fn server(c: Chan<(), CalcSrv>) {
  offer!{c,
    NEG => {
      let (c, n) = c.recv();
      let c = c.send(-n);
      c.close();
    },
    SQR => {
      let (c, n) = c.recv();
      let c = c.send(n * n);
      c.close();
    }
  }
}
```

```
type SqrCli  = Send<i32, Recv<i32, Eps>>;
type NegCli  = Send<i32, Recv<i32, Eps>>;
type CalcCli = Choose<SqrCli, NegCli>;

fn client(c: Chan<(), CalcCli>) {
  let c = c.sel1(); // select SqrCli
  let c = c.send(42);
  let (c, n) = c.recv();
  println!("{}", n);
  c.close();
}

// main() prints -42
fn main() {
  connect(server, client);
}
```

Lastly, while we haven't discussed recursive session types in this paper, Laumann *et al*. [9] use a rather cumbersome encoding for recursive session types, using type-level de Bruijn indices. Below we show two definitions of the session type for a stream of integers, one in Rusty Variation, and one using the library by Laumann *et al*. [9]:

```
// Rusty Variation
type IntStream = Send<i32, IntStream>;
```

```
// Laumann et al.
type IntStream = Rec<Send<i32, Var<Z>>>;
```

The usage of de Bruijn indices to encode recursive session types is an artefact of the fact that, when Laumann *et al*. [9] implemented their library, Rust didn't support recursive types.

4 Future Work

4.1 Heterogeneous Selection

In section 2.5, we mention that the `select_mut` and `select` functions are homogeneous: they only select over vectors of endpoints with *the same type*. Heterogeneous selection is an important operation for channel-based communication. *Typing* heterogeneous selection, and by extension, heterogeneous vectors is notoriously cumbersome without dependent types. There are several avenues for future research:

The `frunk` crate[12] provides an encoding of heterogeneous lists in Rust. We could use this encoding to define a heterogeneous variant of selection.

Rust has limited support for existential types[13], which could be used to quantify over the types of the values received and the session continuations, and select on a homogeneous vector of the (purely fictional) type `Vec<∃T.∃S.Recv<T,S>>`.

[12] `https://beachape.com/frunk/frunk/index.html`
[13] `https://github.com/rust-lang/rfcs/blob/master/text/2071-impl-trait-existential-types.md`

4.2 Testing & Verification

In section 2, we claim that RV preserves the metatheory of EGV, and while a full discussion of the testing and formal verification efforts we have undertaken to ensure the correctness of RV is out of the scope of this paper, it would be remiss of us not substantiate this claim.

Firstly, RV is unit tested. We test various use-cases from EGV, from basic operations and their interactions with cancellation, to delegation and recursive session types. This test suite has caught several errors in the implementation. However, unit tests are error prone, and it is hard (if not impossible) to obtain a set of tests which cover all use-cases of a library. Unfortunately, tools such as RustBelt [10] and Rust Distilled [15] are not yet mature enough to allow us to *prove* the correctness of RV.

In the absence of a mature formal semantics for Rust, we are designing a formal language, RV (or "formal RV"), which matches, as close as possible, the semantics of our Rust library. It is a concurrent lambda calculus with heap-based shared-memory semantics. We plan to prove that formal RV guarantees preservation, progress, termination, confluence and the diamond property, and deadlock and livelock freedom. Furthermore, we plan to show that there exists a translation from EGV to formal RV, based on the monadic reflection of exceptions into the `Option` monad [4] and the translation of channel-based communication to shared memory, which is in operational correspondence with EGV. This would show that formal RV is a faithful implementation of EGV.

Lastly, we plan to ensure the correspondence between formal RV and the implementation of RV, using property-based testing. We plan to generate a series of random formal RV terms, using Neat [1], and check if their behaviour, when run as Rust programs, corresponds to the behaviour of the term in formal RV.

4.3 Extending Exceptional GV

Several features of Rusty Variation are not covered by the formal language EGV. In section 2.5, we mention that selection isn't a part of EGV. In section 3, we mention that recursion isn't part of EGV. If we wish to have any hope of formally verifying RV, along the lines described in section 4.2, we would need to extend EGV to cover these features.

References

[1] Koen Claessen, Jonas Duregård & Michał H. Pałka (2015): *Generating constrained random data with uniform distribution*. *Journal of Functional Programming* 25, doi:10.1017/s0956796815000143. Available at https://doi.org/10.1017/s0956796815000143.

[2] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session types revisited*. *Information and Computation* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002. Available at https://doi.org/10.1016/j.ic.2017.06.002.

[3] The Rust Project Developers (2019): *The Rust Programming Language*, 1.35.0 edition. Available at https://doc.rust-lang.org/1.35.0/book/.

[4] Andrzej Filinski (1994): *Representing monads*. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*, ACM Press, doi:10.1145/174675.178047. Available at https://doi.org/10.1145/174675.178047.

[5] Simon Fowler, Sam Lindley, J. Garrett Morris & Sára Decova (2019): *Exceptional Asynchronous Session Types: Session Types Without Tiers*. *Proc. ACM Program. Lang.* 3(POPL), pp. 28:1–28:29, doi:10.1145/3290341. Available at http://doi.acm.org/10.1145/3290341.

[6] Simon J. Gay & Vasco T. Vasconcelos (2009): *Linear type theory for asynchronous session types*. *Journal of Functional Programming* 20(01), p. 19, doi:10.1017/s0956796809990268. Available at http://dx.doi.org/10.1017/S0956796809990268.

[7] Stjepan Glavina (2019): *Proposal: New channels for Rust's standard library*. Available at https://stjepang.github.io/2019/03/02/new-channels.html.

[8] Kohei Honda (1993): *Types for dyadic interaction*. In: *CONCUR'93*, Springer Nature, pp. 509–523, doi:10.1007/3-540-57208-2_35. Available at http://dx.doi.org/10.1007/3-540-57208-2_35.

[9] Thomas Bracht Laumann Jespersen, Philip Munksgaard & Ken Friis Larsen (2015): *Session types for Rust*. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, ACM, pp. 13–22.

[10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers & Derek Dreyer (2017): *RustBelt: securing the foundations of the rust programming language*. *Proceedings of the ACM on Programming Languages* 2(POPL), pp. 1–34, doi:10.1145/3158154. Available at https://doi.org/10.1145/3158154.

[11] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In: *Programming Languages and Systems*, Springer Berlin Heidelberg, pp. 560–584, doi:10.1007/978-3-662-46669-8_23. Available at https://doi.org/10.1007/978-3-662-46669-8_23.

[12] Luca Padovani (2017): *A simple library implementation of binary sessions*. *Journal of Functional Programming* 27, p. e4, doi:10.1017/S0956796816000289.

[13] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight session programming in Scala*. In: *LIPIcs-Leibniz International Proceedings in Informatics*, 56, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[14] Philip Wadler (2012): *Propositions As Sessions*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, ACM, New York, NY, USA, pp. 273–286, doi:10.1145/2364527.2364568. Available at http://doi.acm.org/10.1145/2364527.2364568.

[15] Aaron Weiss, Daniel Patterson & Amal Ahmed (2018): *Rust Distilled: An Expressive Tower of Languages*. *CoRR* abs/1806.02693. Available at http://arxiv.org/abs/1806.02693.