RUSTY VARIATION

(or, Deadlock-free sessions with failure in Rust)

by Wen Kokke

WHY SESSION TYPES IN RUST?

THE PING PROTOCOL

```
fn send_ping(s1: Send<(), End>)
        -> Result<(), Box<Error>> {

    let s2 = send((), s1)?;
    close(s2)



}
```

THE PING PROTOCOL — RE-USE

```
fn send_ping(s1: Send<(), End>)
        -> Result<(), Box<Error>> {

    let s2 = send((), s1)?;
    close(s2)?;

    let s3 = send((), s1)?; // reuse `s1`
    close(s3)

}
```

THE PING PROTOCOL — DROPPING

```
fn send_ping(s1: Send<(), End>)
        -> Result<(), Box<Error>> {


    // this function body
    // unintentionally left blank.


    Ok(())
}
```

THE PING PROTOCOL — A LONG WAIT

```
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)
```

A TALE OF TWO LANGUAGES IN FOUR EXAMPLES

EXCEPTIONAL GV

(by Fowler et al.)

Looks like this:

$$
\begin{aligned}
&\textbf{let } s = \textbf{fork}(\lambda(s : !\mathbf{1}.\,\mathrm{End}).\\
&\qquad \textbf{let } s = \textbf{send}((), s)\\
&\qquad \textbf{close}(s)\\
&)\\
&\textbf{let } ((), s) = \textbf{recv}(s)\\
&\textbf{close}(s)
\end{aligned}
$$

RUSTY VARIATION

(by me)

Looks like this:

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

I KNOW, THE FONTS ARE VERY DIFFERENT

ROADMAP

» talk about Exceptional GV

» talk about Rusty Variation

» what are the differences?

» what are the similarities?

EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$\bullet \left(\begin{array}{l} \mathbf{let}\ s = \mathbf{fork}(\lambda(s : !\mathbf{1}.\mathrm{End}). \\ \quad\quad \mathbf{let}\ s = \mathbf{send}((), s) \\ \quad\quad \mathbf{close}(s) \\ ) \\ \mathbf{let}\ ((), s) = \mathbf{recv}(s) \\ \mathbf{close}(s) \end{array} \right)$$

We mark the main thread with a •
Next we evaluate the fork instruction

EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{ll} & \bullet \begin{pmatrix} \mathbf{let}\ s = a \\ \mathbf{let}\ ((), s) = \mathbf{recv}(s) \\ \mathbf{close}(s) \end{pmatrix} \quad \| \\ & \circ \begin{pmatrix} \mathbf{let}\ s = \mathbf{send}((), b) \\ \mathbf{close}(s) \end{pmatrix} \quad \| \\ & a(\epsilon) \leftrightsquigarrow b(\epsilon) \end{array}\right)$$

This forks off the process and allocates a buffer
Next we evaluate the let binding

EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{ll} & \bullet\begin{pmatrix}\mathbf{let}\ ((),s)=\mathbf{recv}(a)\\ \mathbf{close}(s)\end{pmatrix} \quad \| \\ & \circ\begin{pmatrix}\mathbf{let}\ s=\mathbf{send}((),b)\\ \mathbf{close}(s)\end{pmatrix} \quad \| \\ & a(\epsilon)\!\leftrightsquigarrow\!b(\epsilon) \end{array}\right)$$

The receive instruction blocks on the empty buffer
Next we evaluate the send instruction

EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\begin{pmatrix}\mathbf{let}\ ((),s)=\mathbf{recv}(a)\\ \mathbf{close}(s)\end{pmatrix} & \| \\ \circ\begin{pmatrix}\mathbf{let}\ s=b\\ \mathbf{close}(s)\end{pmatrix} & \| \\ a((),\epsilon)\leftrightsquigarrow b(\epsilon) & \end{array}\right)$$

This moves the value to the buffer
Next we evaluate the let binding

EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\begin{pmatrix}\mathbf{let}\ ((),s) = \mathbf{recv}(a) \\ \mathbf{close}(s)\end{pmatrix} & \| \\ \circ(\mathbf{close}(b)\,) & \| \\ a((),\epsilon) \leftrightsquigarrow b(\epsilon) & \end{array}\right)$$

The close instruction blocks (it is synchronous)
Next we evaluate the receive instruction

EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\begin{pmatrix}\mathbf{let}\ ((),s) = ((),a) \\ \mathbf{close}(s)\end{pmatrix} & \| \\ \circ(\,\mathbf{close}(b)\,) & \| \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \end{array}\right)$$

This moves the value to the main thread
Next we evaluate the let binding

EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet(\mathbf{close}(a)) & \| \\ \circ(\mathbf{close}(b)) & \| \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \end{array}\right)$$

The close instructions are no longer blocked

(The buffer is empty and there is a close instruction waiting on either side)

Next we evaluate the close instructions

EXCEPTIONAL GV

Let's see how our example EGV program executes!

- ()

Fin

RUSTY VARIATION

What about our Rust program?

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

RUSTY VARIATION

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

RUSTY VARIATION

```
let (s, here) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let s = send((), s)?;
        close(s)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let s = here
let ((), s) = recv(s)?;
close(s)
```

RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

SOUNDS FAMILIAR?

LET'S TALK ABOUT ERRORS

EXCEPTIONAL GV

(by Fowler et al.)

Looks like this:

$$
\begin{aligned}
&\textbf{let } s = \textbf{fork}(\lambda(s : !\mathbf{1}.\,\mathrm{End}).\\
&\qquad \textbf{cancel}(s)\\
&)\\
&\textbf{let } ((), s) = \textbf{recv}(s)\\
&\textbf{close}(s)
\end{aligned}
$$

RUSTY VARIATION

(by me)

Looks like this:

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

I KNOW, THE FONTS ARE VERY DIFFERENT

EXCEPTIONAL GV

Let's see how EGV handles errors!

$$\bullet \left(\begin{array}{l} \mathbf{let}\ s = \mathbf{fork}(\lambda(s : !\mathbf{1}.\mathrm{End}). \\ \qquad \mathbf{cancel}(s) \\ ) \\ \mathbf{let}\ ((), s) = \mathbf{recv}(s) \\ \mathbf{close}(s) \end{array} \right)$$

We mark the main thread with a •
Next we evaluate the fork instruction

EXCEPTIONAL GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet \begin{pmatrix} \textbf{let } s = a \\ \textbf{let } ((), s) = \textbf{recv}(s) \\ \textbf{close}(s) \end{pmatrix} & \| \\ \circ (\textbf{cancel}(b)) & \| \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \end{array}\right)$$

This forks off the process and allocates a buffer
Next we evaluate the let binding

EXCEPTIONAL GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet \begin{pmatrix} \mathbf{let}\ ((), s) = \mathbf{recv}(a) \\ \mathbf{close}(s) \end{pmatrix} & \| \\ \circ (\mathbf{cancel}(b)) & \| \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \end{array}\right)$$

The receive instruction blocks on the empty buffer
Next we evaluate the cancel instruction

EXCEPTIONAL GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\begin{pmatrix}\mathbf{let}\ ((),s)=\mathbf{recv}(a)\\ \mathbf{close}(s)\end{pmatrix} & \| \\ a(\epsilon)\leftrightsquigarrow b(\epsilon) & \| \\ \lightning a & \end{array}\right)$$

This cancels the session and creates a zapper thread
Next we evaluate the receive instruction

EXCEPTIONAL GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\begin{pmatrix}\mathbf{let}\ ((), s) = \mathbf{raise} \\ \mathbf{close}(s)\end{pmatrix} & \| \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \| \\ \lightning b & \| \\ \lightning a & \end{array}\right)$$

Receiving on a channel raises an exception
if the other endpoint is cancelled

EXCEPTIONAL GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\,\mathbf{halt} & \| \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \| \\ \lightning a & \| \\ \lightning b & \end{array}\right)$$

An uncaught exception turns into halt
Next we garbage collect the buffer

EXCEPTIONAL GV

Let's see how EGV handles errors!

- **halt**

Fin

RUSTY VARIATION

What about the Rust library?

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

RUSTY VARIATION

For that, let's look at how cancel is implemented:

```
fn cancel<T>(x: T) -> () {
    // this function body
    // intentionally left blank.
}
```

Wait, what happened to x?

It went out of scope!

RUSTY VARIATION

What happens when a channel x leaves scope unused?

» destructor is called

» values in buffer are deallocated

» destructors for values in buffer are called

» buffer is marked as DISCONNECTED

» calling recv on DISCONNECTED buffer returns Err

SOUNDS FAMILIAR?

WHAT ARE THE DIFFERENCES?

» explicit cancellation vs. implicit cancellation

(what happens if we forget to complete a session?)

» try/catch vs. error monad

(using the "$\mathbf{try}\, L\, \mathbf{as}\, x\, \mathbf{in}\, N\, \mathbf{otherwise}\, M$" instruction)

» channel vs. shared memory

(process calculus vs. heap-based semantics)

WHAT ARE THE DIFFERENCES?

» simply-typed linear lambda calculus vs. Rust

this means we have:

» no recursion vs. general recursion

» lock freedom vs. deadlock freedom

» etc.

HOW CAN WE GET DEADLOCKS IN RUSTY VARIATION?

» by using mem::forget

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

» by storing channels in manually managed memory and not cleaning up

WHAT ARE THE SIMILARITIES?

» in theory, everything else?

» can we prove it?

> “doesn't Rust have formal semantics?
> I heard so much about RustBelt!
>
> no.
>
> RustBelt formalises elaborated Rust and
> doesn't support many features we depend on.

WHAT ARE THE SIMILARITIES?

» in theory, everything else?

» can we prove it? no.

» can we test it?

```
#[test]
fn ping_works() {
    assert!(|| -> Result<(), Box<Error>> {

        // ...insert example here...

    }().is_ok()); // it actually is!
}
```

WHAT ARE THE SIMILARITIES?

» in theory, everything else?

» can we prove it? no.

» can we test it? yes.

» can we properly test it?

HOW EFFICIENT IS RUSTY VARIATION?

» buffers are either empty or non-empty

» size of buffers is statically known

  (unless you're sending boxed references)

» each buffer only involves a single allocation

» size of session is statically known

  (but buffers are allocated lazily)

» it's really quite efficient y'all

RELATED WORK

session-types

(by Laumann et al.)

» library for session types in Rust

» dibsed the best package name

» embeds LAST[2] in Rust

(a linear language embedded in an affine one)

» forget to complete a session? segfault!

[2] Linear type theory for asynchronous session types, Gay & Vasconcelos, 2010

THE PING PROTOCOL — DROPPING

```
fn send_ping(s1: Send<(), End>)
        -> Result<(), Box<Error>> {


    // this function body
    // unintentionally left blank.


    Ok(())
}
```

THE PING PROTOCOL — DROPPING

```
fn recv_ping(s1: Recv<(), End>)
        -> Result<(), Box<Error>> {


    // this function body
    // unintentionally left blank.


    Ok(())
}
```

CONCLUSIONS

RUSTY VARIATION

» embeds EGV into Rust

» is unit tested

» will be QuickChecked

» is very efficient

» improves session-types