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

ABSTRACT

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.

ACM Reference Format:
Wen Kokke. 2018. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM & MOTIVATION

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, it the correctness of the implementation with respect to that protocol is rarely checked.

Session types are a substructural typing discipline, introduced by Honda [4], which describe communication protocols as *channel types*, capturing both the type 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], 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 races. We assume some familiarity with Rust. For an introduction to Rust, we refer the reader to *The Rust Programming Language*.[3]

2 BACKGROUND & RELATED WORK

Jespersen, Munksgaard, and Larsen [5] implement session types in Rust, based on the session-typed functional language LAST by Gay and Vasconcelos [3]. Their implementation guarantees that programs correctly implement specified communication protocols with some caveats. LAST depends upon linear types, meaning that sessions have to be used exactly once. Rust only offers affine types, 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.* In some cases, the compiler can be made to issue a warning at compile-time when a channel is dropped, for instance through the `#[must_use]` annotation. However, in general, this is not something the compiler can guarantee. Furthermore, LAST ignores the possibility of communication failures, such as network errors, which cannot be detected at compile-time, yet can cause a channel to be dropped.

We embrace the affine, and base our work on Exceptional GV by Fowler et al. [2, EGV]. Exceptional GV is a descendant of Good Variation [8, GV], which itself is a descendant of LAST. Crucially, EGV has an affine type system, and allows for the explicit cancellation of sessions. Furthermore, EGV offers a stronger metatheory than LAST. In addition to session fidelity, EGV also guarantees 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. [2].

3 CONTRIBUTION: RUSTY VARIATION

Our contribution, Rusty Variation, is an implementation of EGV in Rust. We claim that it is a faithful implementation, which preserves much of EGVs metatheory. However, we depart from it in two ways:

First, EGV relies upon exceptions and handlers. Rust, however, has rejected exceptions in favour of monadic errors i.e., wrapping possibly failing results in error types such as `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 session fidelity, livelock freedom, or—well—termination. The work by Jespersen, Munksgaard, and Larsen [5] also suffers from this shortcoming. We can demonstrate this by writing the program which loops forever, and then sends a message. The Rust type checker does not see a problem with this, but obviously the message will never be sent. However, we claim that if the program terminates, it respected the communication protocol.

3.1 Sessions and duality

Any type built using the primitive types below is a valid session:

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

Session *types* are public, but their members are not. Therefore, user 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 implements 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 is a session, and that `S::Dual::Dual` is equal to `S` (involutivity).

The `new` function generates a new session 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 3.6.

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

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

The client types are obtained by duality.

*Graduate student (9727050), supervised by Philip Wadler and Ian Stark.
[1] https://insights.stackoverflow.com/survey/2018/
[2] https://github.com/wenkokke/rusty-variation
[3] https://doc.rust-lang.org/nightly/book/

3.2 Send, receive, and fork!

The `send` and `recv` functions send and receive values in sessions. They have the following types:

```
pub fn send<T, S: Session>(x: T, s: Send<T, S>)
       -> Result<S, Box<Error>>;
pub fn recv<T, S: Session>(s: Recv<T, S)
       -> Result<(T, S), Box<Error>>;
```

The `fork!` macro creates a new session, and spawns a new thread. The new thread runs the first argument of `fork!`—a code block—which receives one endpoint, and `fork!` returns the dual endpoint. While `fork!` is implemented as a macro, it can be considered as a function with the following type:

```
pub fn fork!<S: Session>(p: P) -> S::Dual
      where
        P: FnOnce(S) -> Result<(), Box<Error>>;
```

Using these primitives, we can write the following program. This program forks off a process which sends a "ping" along the session, then waits to receive that ping and returns:

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

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

3.3 Cancellation

The `cancel` function drops a value and cancels all sessions *and* their duals referenced in that value. This cancellation propagates: any process attempting to write or write to a cancelled channel will return fail and cancel all *their* open channels. This ensures that no process is left waiting for a message that will never come.

```
pub fn cancel<T>(x: T) -> Result<(), Box<Error>>;
```

Using `cancel`, we can implement the following program. This program forks off a process which it *expects* to send a "ping". However, instead the forked process cancels the session. Therefore, the call to `recv` fails, the `?`-operator short-circuits, and the whole process returns an error:

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

The cancellation mechanism is implemented through destructors. In fact, in the current implementation it is inherited from the implementation of channels in `std::sync::mpsc`.

The `Send` and `Recv` structures are annotated with `#[must_use]`, to encourage the user to complete the protocol or cancel it explicitly. However, should a session be dropped, this is interpreted as *implicit* cancellation. This ensures deadlock freedom, even in Rust's affine type system.

3.4 Branching with offer! and select!

The `offer_either`, `select_left`, and `select_right` functions let processes offer or make a *binary* choice. They implement this by receiving or sending a value of the sum type `Either<S1, S2>`, where `S1` and `S2` are the possible continuations of the session. For instance, we could write the type of a "calculator" server, offering negation and addition, as `Recv<Either<NegSrv, AddSrv>, End>`.

While these functions suffice to encode choice, we also define two macros, `offer!` and `select!`, which generalise this functionality to *all enums*—as long as each value wraps a session. This allows processes to offer and select choices between many *labelled* branches, and allows us to define the type for our calculator server as:

```
type CalcSrv = Recv<CalcOp, End>;
enum CalcOp {Neg(NegSrv), Add(AddSrv)}
```

Using these macros, we can write the following program. This program forks off a calculator server, which offers the choice between negation and addition, selects addition, sends some numbers, and then returns the result:

```
let s = fork!(move |s: CalcSrv| {
  offer!(s, {
    CalcOp::Neg(s) => {
      let (x, s) = recv(s)?;
      send(-x, s)
    },
    CalcOp::Add(s) => {
      let (x, s) = recv(s)?;
      let (y, s) = recv(s)?;
      send(x + y, s)
    },
  })});
let s = select!(CalcOp::Add, s)?;
let s = send(4, s)?;
let s = send(5, s)?;
let (z, End) = recv(s)?;
Ok(z)
```

3.5 Implementation

We use the asynchronous channels from `std::sync::mpsc`. These are ideal, as they already implement the correct cancellation behaviour. Channels are typed, and can only transfer values of a single type. Therefore, we use one-shot channels, and encode sessions following Dardha, Giachino, and Sangiorgi [1], Padovani [6], and Scalas and Yoshida [7]: each `send` creates a new channel for the continuation of the session, and sends that along with the value, and each `recv` receives a value together with a channel on which to continue the session. Fortunately, the channels in `std::sync::mpsc` are highly optimised for one-shot usage.[4]

3.6 What If I Want to Be Evil?

There are three ways in which you can make code written using Rusty Variation deadlock. The first is by looping, which we discussed earlier. The other two are explicit, and surprisingly, `unsafe` is not one of them:

The biggest issue lies in the fact that Rust's semantics do not guarantee that destructors are actually run. For instance, it is always possible to "forget" a value, using `std::mem::forget`, without running its destructors. The following simple program deadlocks:

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

Second, it is not possible—at the moment—to export a trait without exporting all its members. Therefore, any user can use the `new` function, and doing so, it is trivial to write a program which deadlocks:

```
let (s1, r1) = Send<Void, End>::new();
let r2 = fork!(move |s2: Send<Void, End>| {
  let (v, End) = recv(r1)?;
  send(v, s2)
});
let (v, End) = recv(r2)?;
send(v, s1)
```

[4] https://doc.rust-lang.org/src/std/sync/mpsc/mod.rs.html#147-149

REFERENCES

[1] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. "Session types revisited". In: *Information and Computation* 256 (Oct. 2017), pp. 253–286. DOI: 10.1016/j.ic.2017.06.002. URL: https://doi.org/10.1016/j.ic.2017.06.002.

[2] Simon Fowler et al. "Session Types without Tiers". Draft on webpage at http://simonjf.com/drafts/zap-draft-mar18.pdf. 2018.

[3] Simon J. Gay and Vasco T. Vasconcelos. "Linear type theory for asynchronous session types". In: *Journal of Functional Programming* 20.01 (Dec. 2009), p. 19. DOI: 10.1017/s0956796809990268. URL: http://dx.doi.org/10.1017/S0956796809990268.

[4] Kohei Honda. "Types for dyadic interaction". In: *CONCUR'93*. Springer Nature, 1993, pp. 509–523. DOI: 10.1007/3-540-57208-2_35. URL: http://dx.doi.org/10.1007/3-540-57208-2_35.

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

[6] Luca Padovani. "A simple library implementation of binary sessions". In: *Journal of Functional Programming* 27 (2017), e4. DOI: 10.1017/S0956796816000289.

[7] Alceste Scalas and Nobuko Yoshida. "Lightweight session programming in Scala". In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[8] Philip Wadler. "Propositions As Sessions". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ICFP '12. Copenhagen, Denmark: ACM, 2012, pp. 273–286. ISBN: 978-1-4503-1054-3. DOI: 10.1145/2364527.2364568. URL: http://doi.acm.org/10.1145/2364527.2364568.