Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types

Zak Cutner
Imperial College London
London, UK

Nobuko Yoshida
Imperial College London
London, UK

Martin Vassor
Imperial College London
London, UK

Abstract

Rust is a modern systems language focused on performance and reliability. Complementing Rust's promise to provide "fearless concurrency", developers frequently exploit asynchronous message passing. Unfortunately, sending and receiving messages in an arbitrary order to maximise computation-communication overlap (a popular optimisation in message-passing applications) opens up a Pandora's box of subtle concurrency bugs.

To guarantee deadlock-freedom by construction, we present Rumpsteak: a new Rust framework based on *multiparty session types*. Previous session type implementations in Rust are either built upon synchronous and blocking communication and/or are limited to two-party interactions. Crucially, none support the arbitrary ordering of messages for efficiency.

Rumpsteak instead targets asynchronous `async/await` code. Its unique ability is allowing developers to arbitrarily order send/receive messages while preserving deadlock-freedom. For this, Rumpsteak incorporates two recent advanced session type theories: (1) k-multiparty compatibility (k-MC), which *globally* verifies the safety of a set of participants, and (2) asynchronous multiparty session subtyping, which *locally* verifies optimisations in the context of a single participant. Specifically, we propose a novel algorithm for asynchronous subtyping that is both sound and decidable.

We first evaluate the performance and expressiveness of Rumpsteak against three previous Rust implementations. We discover that Rumpsteak is around 1.7–8.6x more efficient and can safely express many more examples by virtue of offering arbitrary ordering of messages. Secondly, we analyse the complexity of our new algorithm and benchmark it against k-MC and a *binary* session subtyping algorithm. We find they are exponentially slower than Rumpsteak's.

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508404

CCS Concepts: • **Software and its engineering** → **Development frameworks and environments**; **Source code generation**; • **Computer systems organization** → Reliability.

Keywords: Rust, Asynchronous Message Passing, Message Reordering, Computation-Communication Overlap, Multiparty Session Types

1 Introduction

Rust is a statically-typed language designed for systems software development. It is rapidly growing in popularity and has been voted "most loved language" over five years of surveys by Stack Overflow [19]. Rust aims to offer the safety of a high-level language without compromising on the performance enjoyed by low-level languages. *Message passing* over *typed channels* is common in concurrent Rust applications, where (low-level) threads or (high-level) actors communicate efficiently and safely by sending messages containing data.

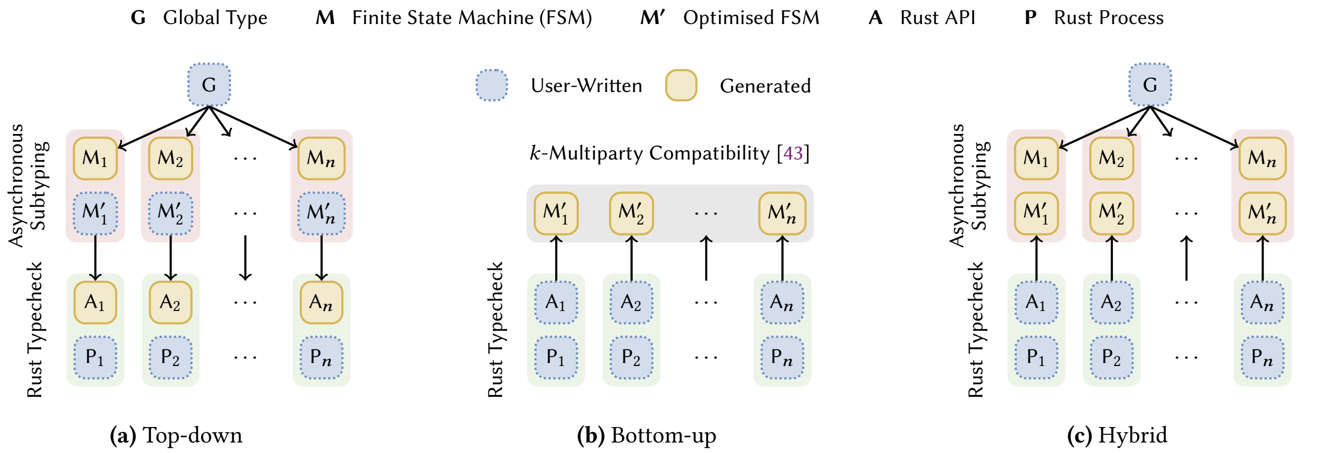
To improve performance by maximising computation-communication overlap [14, 37, 55], developers often wish to arbitrarily change the order of sending and receiving messages—we will present several examples of this technique, which we refer to as *asynchronous message reordering* (AMR). Our challenge is to remedy communication errors such as deadlocks, which can easily occur in message-passing applications, particularly those that leverage AMR.

To achieve this, we introduce Rumpsteak: a framework for efficiently coordinating message-passing processes in Rust using *multiparty session types*. Session types [28, 56] (see [61] for a gentle introduction and [23] for a more exhaustive one) coordinate interactions through *linearly typed channels*, that must be used exactly once, ensuring *protocol compliance* without deadlocks or communication mismatches.

Current state of the art. Since Rust's *affine type system* is particularly well-suited to session types by statically guaranteeing a linear usage of session channels, there are several previous attempts at implementing session types in Rust [12, 34, 38, 41]. However, their current limitations prevent them from guaranteeing all four of *deadlock-freedom*, *multiparty communication*, *asynchronous execution* and AMR.

Our framework. We motivate the importance of each feature and explain how Rumpsteak incorporates this.

Deadlock-freedom. One of the most important properties of concurrent/parallel systems is that their computations



(a) Top-down **(b)** Bottom-up **(c)** Hybrid

Figure 1. Workflow of the Rumpsteak framework (three approaches).

are not blocked. Deadlock-freedom in our context states that the system can always either make progress by exchanging messages or properly terminate.

Multiparty communication. Many previous Rust implementations [12, 34, 38] support only *binary* session types, which is limited to two-party communication. On the other hand, the majority of real-world communication protocols consist of more than two participants. Rumpsteak therefore uses *multiparty* session types (MPST) [29, 30] to ensure deadlock-freedom in protocols with any number of participants (or roles).

Asynchronous execution. Most previous Rust implementations [34, 38, 41] use *synchronous* communication channels. This approach suffers from performance limitations since threads are *blocked* while waiting to receive messages. Rumpsteak instead uses *asynchronous* communication, where lightweight asynchronous tasks share a pool of threads. When one task is blocked, another's work can be scheduled in the meantime to prevent the wasting of computational resources.

Although [12] is also based on asynchronous communication, only Rumpsteak closely integrates with Rust's modern `async`/`await` syntax, allowing asynchronous programs to be written sequentially. To achieve this, asynchronous functions are annotated with `async`, causing them to return *futures*. Developers can `await` calls to these functions, denoting that execution should continue elsewhere until a result is ready.

Asynchronous message reordering. Our main contribution is offering AMR, which no existing work can provide while preserving deadlock-freedom. To motivate this, we introduce a running example of the *double buffering* protocol [33]. Buffering is frequently used in multimedia applications where a continuous stream of data must be sent from a source (e.g. a graphics card) to a sink (e.g. a CPU). To prevent the source from being blocked while the sink is busy, it writes to a buffer, which is later read by the sink.

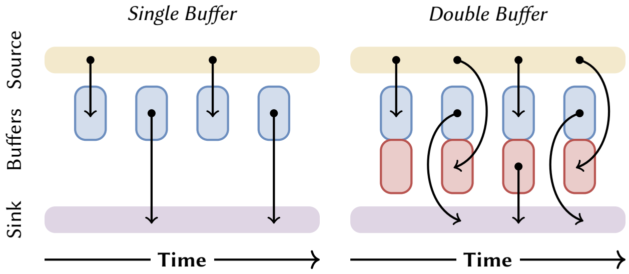


Figure 2. Illustration of the double buffering protocol.

We illustrate this effect in Fig. 2. With a single buffer, both the source and sink are constantly blocked as we cannot read and write simultaneously. However, adding a second buffer allows the source and sink to operate on different buffers at once so that (in the best case) they are never blocked and throughput can increase twofold.

As we will see in § 2, the double buffering protocol takes advantage of AMR so it *cannot* be expressed with standard MPST theory [29]; communication with the source and sink are overlapped so both buffers can be accessed at once. The goal of this paper is to provide a method for ensuring that protocols optimised in this way preserve deadlock-freedom.

Rumpsteak framework. We achieve these features while allowing three different approaches, summarised in Fig. 1.

In the **top-down approach**, the developer writes a global type, which describes the entire protocol (see § 2 for an example). We project this onto each participant to obtain a local finite state machine (FSM), which describes the protocol from that participant's perspective. Next, the developer uses AMR to propose optimised FSMs for each participant. We *locally* verify that each optimised FSM is compatible with the projected one using an asynchronous subtyping algorithm. From each optimised FSM, we generate an API so that the developer can write a Rust process implementation. Our

API uses Rust's type checker to ensure that these processes conform to the protocol, and are therefore deadlock-free.

In the **bottom-up approach**, the developer manually writes an API and process implementation for each participant. We derive the optimised FSMs directly from these APIs and ensure that they are safe using *k-multiparty compatibility* (k-MC) [43]. k-MC globally verifies that multiple FSMs are compatible with each other without using a global type.

Finally, we propose a **hybrid approach** where the projected FSMs are generated from a global type on one side (as in the top-down approach). On the other side, the developer writes both the APIs and process implementations directly, and we derive the optimised FSMs (as in the bottom-up approach). We then *locally* verify that the optimised FSMs are asynchronous subtypes of the projected FSMs.

To summarise, our subtyping algorithm used in the top-down/hybrid approaches *locally* verifies the correctness of optimisations. This makes it far more scalable than k-MC as we will see in **§ 4**. These approaches are also more intuitive to the developer since they use *safety by construction*—it is much easier to write a global type than to determine why a k-MC analysis has failed for a complex protocol.

Contribution and outline. In **§ 2**, we give an overview of the design and implementation of Rumpsteak. In **§ 3**, we present our new *asynchronous subtyping algorithm* used to check that an optimised FSM is correct. We prove our new algorithm is *sound* (Theorem 7) against the precise MPST asynchronous subtyping theory [25], *terminates* (Theorem 6) and analyse its complexity (Theorem 9). In **§ 4**, we compare the performance and expressiveness of our framework with existing work [7, 12, 38, 41, 43]. We show **(1)** Rumpsteak's runtime is faster and can express many more asynchronous protocols than other Rust implementations [12, 38, 41]; and **(2)** our subtyping algorithm is more efficient than existing algorithms [7, 43], confirming our complexity analysis. **§ 5** discusses related work and concludes. The **supplementary materials** available in the full version [17] contain further examples and proofs of the theorems. We include our source code and benchmarks in a **public GitHub repository** [3].

2 Design and Implementation

This section presents Rumpsteak and explains its three workflows: a *top-down approach* using asynchronous subtyping, a *bottom-up approach* using k-multiparty compatibility (k-MC) [43] and a *hybrid approach* combining the two.

2.1 Top-Down Approach

All three approaches result in the same final application, so we use the example of the top-down workflow (Fig. 1a) to give a general overview of Rumpsteak's implementation.

Two-party protocol. Before going into the details of the top-down approach, we first give an informal insight into how the protocol is represented at each stage, shown in Fig. 3. For this, we use the example of a simple streaming protocol, later introduced in **§ 4**.

A global session type describes the protocol from a global perspective and includes all participants in the protocol. The global type for the streaming protocol (shown below) is read as follows: participant t first sends *ready* to participant s. Next, s replies to t with two possible messages: either *value* or *stop*. In the latter case, the protocol terminates (type end). In the former case, the protocol continues with **x**, which is a type variable that is bound by μ**x**., i.e. the protocol starts over.

$$\mathrm{G}_{\mathrm{ST}} = \mu \mathbf{x}.\mathtt{t} \to \mathtt{s} : \{\mathit{ready}.\mathtt{s} \to \mathtt{t} : \{\mathit{value}.\mathbf{x}, \quad \mathit{stop}.\mathsf{end}\}\}$$

In Rumpsteak, developers express global types syntactically with Scribble [54, 62]: a widely used and target-agnostic language for describing multiparty protocols. We show the corresponding Scribble description for the streaming protocol in Fig. 3a.

A local finite state machine describes the protocol from the perspective of a single participant. It shows the send and receive actions of that participant, independent of what other participants may be doing in the meantime.

In Fig. 3b, we show the local FSMs for each participant in the streaming protocol, where ! and ? denote *send* and *receive* respectively in session type syntax [61]. For example, the FSM for the source first receives *ready* from the sink, then chooses to either send *value* and start over or simply send *stop* and finish.

The Rust API is an encoding of a local FSM as Rust code. This code then uses Rust's type checker to confirm that a process implementation, also written in Rust, conforms to the FSM. For example, the encodings for the source and sink are shown in Fig. 3c.

Multiparty protocol. In the remainder of this section, we use a more complex example (the double buffering protocol from **§ 1**), which allows us to illustrate how Rumpsteak can verify multiparty protocols. We carefully go through each step of the top-down approach, closely following Fig. 1a.

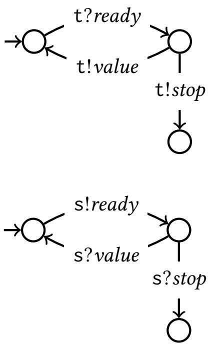
To guarantee safety in our double buffering example using MPST, the developer defines a global type G_{DB} for the protocol. This protocol has three participants: a source s, a kernel k (which controls both of the buffers) and a sink t. We also show the corresponding Scribble description for the double buffering protocol in Listing 1.

$$\begin{aligned}\mathrm{G}_{\mathrm{DB}} = \mu \mathbf{x}.\mathtt{k} \to \mathtt{s} : \{&\mathit{ready}.\mathtt{s} \to \mathtt{k} : \{\\ &\quad \mathit{value}.\mathtt{t} \to \mathtt{k} : \{\mathit{ready}.\mathtt{k} \to \mathtt{t} : \{\mathit{value}.\mathbf{x}\}\}\}\}\end{aligned}$$

Projection. Once we have created a global type, we use a projection algorithm to derive the local FSM for each participant. In Rumpsteak, we perform projection using νScr [2]: a new lightweight and extensible Scribble toolchain implemented in OCaml.

```
global protocol Ring(role s,
    role t) {
  rec loop {
    ready() from t to s;
    choice at s {
      value() from s to t;
      continue loop;
    } or {
      stop() from s to t;
    }
  }
}
```

(a) Scribble protocol description



(b) FSMs for the source and sink

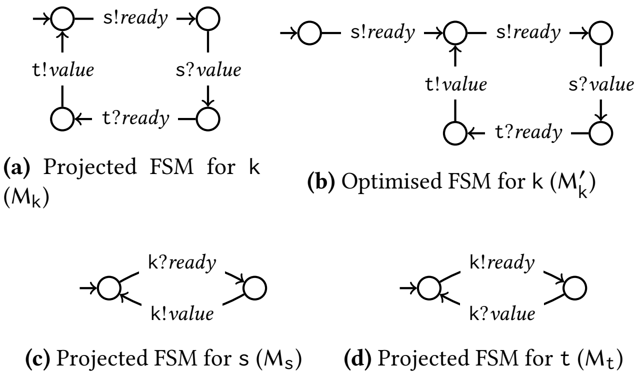
```
type Source = Receive<T, Ready,
    Select<T, SourceChoice>>;

enum SourceChoice {
  Value(Value, Source),
  Stop(Stop, End) }

type Sink = Send<S, Ready,
    Branch<S, SinkChoice>>;

enum SinkChoice {
  Value(Value, Sink),
  Stop(Stop, End) }
```

(c) Rust API code (simplified excerpt)

Figure 3. Different session type representations used within Rumpsteak.

Listing 1. Scribble representation of the double buffering protocol.

```
global protocol DoubleBuffering(role s,
    role k, role t) {
  rec loop {
    ready() from k to s;
    value() from s to k;
    ready() from t to k;
    value() from k to t;
    continue loop;
  }
}
```



(a) Projected FSM for k (M_k)

(b) Optimised FSM for k (M'_k)

(c) Projected FSM for s (M_s)

(d) Projected FSM for t (M_t)

Figure 4. FSMs for roles k, t and s of the double buffering protocol.

We show the projected local FSM for each of the participants in Fig. 4. For instance, the kernel, whose projected type M_k is shown in Fig. 4a, **(1)** informs the source that it is ready to receive; **(2)** receives a value from the source; **(3)** waits for the sink to become ready; **(4)** sends a value to the sink; and **(5)** repeats from step 1.

Asynchronous message reordering. Unfortunately, global types cannot represent overlapping communication. Therefore, the projection M_k cannot achieve the simultaneous interactions we saw in Fig. 2. In practice, developers use AMR to produce an optimised FSM for the kernel M'_k (shown in Fig. 4b). Here, the kernel initially sends two *ready* messages to the source. This allows the source to write to the second buffer while the sink is busy reading from the first. Note that the asynchronous queue is effectively acting as the second buffer—the second message from the source waits in the queue while the kernel is processing the first.

Crucially, Rumpsteak verifies that M'_k is safe to use in the place of M_k without causing deadlocks or other communication errors. Otherwise, we would be throwing away the safety benefits that come with MPST. To achieve this, we must determine that M'_k is an *asynchronous subtype* of M_k, which we perform using our algorithm presented in **§ 3**.

API design. Once we arrive at an optimised FSM M'_i for each participant, our challenge is to create a Rust API A_i, which uses Rust's type checker to ensure that a developer-written process P_i conforms to M'_i (a relevant introduction to the Rust programming language can be found in Jung's thesis [36, Chapter 8]). In the top-down approach, this API is automatically generated from an optimised FSM.

To illustrate, we show the API for the kernel (A_k) in Listing 2, which checks conformance to M'_k. To ensure that our API remains readable by developers and to eliminate extensive boilerplate code, we make use of Rust's procedural macros [57]. By decorating types with #[...], these macros perform additional compile-time code generation.

Roles (Participants). Each role is represented as a struct, which stores its communication channels with other roles. The struct for the kernel (ls. 1 to 6) contains channels to and from s and t. Developers can in fact use any custom channel that implements Rust's standard Sink or Stream interfaces [15] for asynchronous sends and receives respectively. This approach minimises the expensive creation of channels

Listing 2. RUMPSTEAK API for the kernel.

```
1  #[derive(Role)]
2  #[message(Label)]
3  struct K(
4    #[route(S)] Channel,
5    #[route(T)] Channel
6  );
7
8  #[derive(Message)]
9  enum Label {
10   Ready(Ready),
11   Value(Value),
12 }
13
14 struct Ready;
15 struct Value(i32);
16
17 #[session]
18 type Kernel = Send<S, Ready, KernelLoop>;
19
20 #[session]
21 struct KernelLoop(Send<S, Ready, Receive<S, Value,
22     Receive<T, Ready, Send<T, Value, Self>>>>);
```

Listing 3. Process implementation for the kernel.

```
1  async fn kernel(role: &mut K) -> Result<()> {
2    try_session(role, |s: Kernel<'_, _>| async {
3      let mut t = s.send(Ready).await?;
4      loop {
5        let s = t.into_session().send(Ready).await?;
6        let (Value(value), s) = s.receive().await?;
7        let (Ready, s) = s.receive().await?;
8        t = s.send(Value(value)).await?;
9      }
10   }).await
11 }
```

in cases where only bounded or unidirectional channels are required.

Our #[derive(Role)] macro generates methods for programmatically retrieving these channels from the struct. Moreover, an optional #[derive(Roles)] macro can be applied on a struct containing every role to also generate code for automatically instantiating all roles at once. This will create the necessary combination of channels and assign them to the correct structs in order to reduce human error.

Sessions. Following the approach of MULTICRUSTY [41], we build a set of *generic primitives* to construct a simple API. For instance, the Send primitive (l. 21) takes a role, label and continuation as generic parameters. In contrast to the standard approach of creating a type for every state [31], these primitives reduce the number of types required and avoid arbitrary naming of these ‘state types’. For brevity, our API elides two additional parameters used to store channels at runtime, which are reinserted with the #[session] macro.

Labels. Internally, RUMPSTEAK sends a Label enum (l. 9) over reusable channels to communicate with other participants. Each label is represented as a type (ls. 14 to 15) and our #[derive(Message)] macro generates methods for converting to and from the Label enum.

Choice. The kernel's API does not contain choice but we show an example of this below. Choice is represented as an enum where each branch contains the label sent/received and a continuation. This design allows processes to pattern match on external choices to determine which label was received. Methods allowing the enum to be used with Branch (an *external* choice, l. 3) or Select (an *internal* choice) are also generated with the #[session] macro. Even so, the enum is still necessary since Rust's lack of variadic generics means choice cannot be easily implemented as its own primitive.

```
1 #[session]
2 enum Choice {
3   Continue(Continue, Branch<A, Self>),
4   Stop(Stop, End),
5 }
```

Process implementation. Using the API A_k, the developer writes an implementation for the process P_k; we show an example in Listing 3. We discuss how our API uses Rust's type system to check that P_k conforms to the protocol.

Linearity. The linear usage of channels is checked by Rust's *affine type system*, preventing channels from being used multiple times. When a primitive is executed, it consumes itself, preventing reuse, and returns its continuation.

Developers are prevented from constructing primitives directly using visibility modifiers. They must instead use try_session (l. 2). This function accepts a reference to the role being implemented and a closure (or function pointer) to its process implementation. The closure takes the input session type as an argument and returns the terminal type End. If a session is discarded, thereby breaking linearity, then the developer will have no End to return and Rust's compiler will complain that the closure does not satisfy its type.

Infinite recursion. At first glance, it seems impossible to implement processes with infinitely recursive types using this approach. Nevertheless, for types without an End primitive, such as Kernel, we can use an infinite loop (l. 4) to get around this problem. Conveniently, infinite loops are assigned !, Rust's never (or bottom) type, which can be implicitly cast to any other type. This allows the type of the loop to be coerced to End, enabling the closure to pass the type checker as before.

Channel reuse. We allow roles to be reused across sessions since the channels they contain are usually expensive to create. Crucially, to prevent communication mismatches between different sessions, we must ensure that the same role is not used in multiple sessions at once. Therefore, try_session takes an *exclusive* reference to the role, causing Rust's borrow checker to enforce this requirement.

2.2 Bottom-Up Approach

In the top-down approach, we generate a Rust API A_i from an optimised FSM M'_i. In the bottom-up approach (Fig. 1b), we do the reverse: we ***serialise*** each API A_i to obtain an FSM M'_i. Next, we use k-MC on the set of FSMs $\mathsf{M}'_{1\dots n}$. If they are indeed compatible, then the processes $\mathsf{P}_{1\dots n}$, which implement their respective APIs, are free from deadlocks.

k-MC takes the optimised FSMs of all participants and verifies deadlock freedom. In contrast, asynchronous subtyping checks the optimisation of a single participant's FSM in isolation. Therefore, k-MC can be seen as a global analysis of the protocol and asynchronous subtyping as a local analysis of a single participant.

To perform the serialisation of an API to an FSM we provide a Rust function `serialize<S>() -> Fsm` (this is a simplified version). It takes a session type API (such as `Kernel` from **§ 2.1**) as a generic type parameter `S` and returns its corresponding FSM. This FSM can be printed in a variety of formats and passed into the k-MC tool for verification.

2.3 Hybrid Approach

Developers may naturally prefer the bottom-up approach since code generation, as used in the top-down approach, can be quite opaque and difficult to understand or debug. Nevertheless, the top-down approach has the advantage of using asynchronous subtyping rather than k-MC—analysing the local types for all participants in the protocol at once is challenging to do scalably (see **§ 4**).

Moreover, when a k-MC analysis fails, it can be difficult to determine how a developer should update a complex protocol to make it free from deadlocks. Safety by construction, as used in the top-down approach, is easier to work with since verification is done locally on each participant.

Therefore, we also propose a third, hybrid approach (Fig. 1c). In this workflow, a global type G is provided by the developer and projected to obtain the FSMs $\mathsf{M}_{1\dots n}$ as before. Rather than the developer proposing the optimised FSMs $\mathsf{M}'_{1\dots n}$ directly, they instead simply write the APIs $\mathsf{A}_{1\dots n}$ (as in the bottom-up approach). These are serialised to $\mathsf{M}'_{1\dots n}$ which can (as in the top-down approach) be checked for safety against $\mathsf{M}_{1\dots n}$ using asynchronous subtyping. In essence, the hybrid approach uses the same theory as the top-down approach, but presents a more developer-friendly interface that uses serialisation rather than code generation.

3 A Sound Asynchronous Multiparty Session Subtyping Algorithm

This section proposes an algorithm for asynchronous multiparty subtyping (**§ 3.2**), shows its soundness (Theorem 7), termination (Theorem 6) and complexity (Theorem 9), and sketches its implementation.

We begin by formally defining global and local session types. In the remainder of this section, we will omit sorts S from the syntax to simplify the presentation.

Definition 1 (Global and local types)**.**

$$\begin{array}{rcl} S & ::= & \mathtt{i32} \mid \mathtt{u32} \mid \mathtt{i64} \mid \mathtt{u64} \mid \dots \\ \mathsf{G} & ::= & \mathsf{end} \mid \mathsf{p} \to \mathsf{q} : \{\ell_i(S_i).\mathsf{G}_i\}_{i\in I} \mid \mu\mathbf{t}.\mathsf{G} \mid \mathbf{t} \\ \mathsf{T} & ::= & \mathsf{end} \mid \oplus_{i\in I}\mathsf{p}!\ell_i(S_i).\mathsf{T}_i \mid \&_{i\in I}\mathsf{p}?\ell_i(S_i).\mathsf{T}_i \mid \mu\mathbf{t}.\mathsf{T} \mid \mathbf{t} \end{array}$$

$\oplus_{i\in I}\mathsf{p}!\ell_i(S_i).\mathsf{T}_i$ and $\&_{i\in I}\mathsf{p}?\ell_i(S_i).\mathsf{T}_i$ represent *internal* and *external* choices respectively where ! and ? denote send and receive respectively, and all ℓ_i are pairwise distinct.

Defining sound message reordering is non-trivial since it may introduce deadlocks, as shown in the example below.

Example 2 (Correct/incorrect AMR)**.** Consider the following local types

$$\mathsf{T}_Q = \mathsf{p}?\ell_1.\mathsf{p}!\ell_2.\mathsf{end} \quad \mathsf{T}_P = \mathsf{q}!\ell_1.\mathsf{q}?\ell_2.\mathsf{end}$$

which are given by projecting a global type

$$\mathsf{p} \to \mathsf{q} : \{\ell_1.\mathsf{q} \to \mathsf{p} : \{\ell_2.\mathsf{end}\}\}$$

Reordering the actions of q to become $\mathsf{T}'_Q = \mathsf{p}!\ell_2.\mathsf{p}?\ell_1.\mathsf{end}$, first sending before receiving, retains deadlock-freedom as messages are stored in queues and their reception can be delayed. However, if instead we reorder p's interactions to become $\mathsf{T}'_P = \mathsf{q}?\ell_2.\mathsf{q}!\ell_1.\mathsf{end}$, we arrive at a deadlock since both processes are simultaneously expecting to receive a message that has not yet been sent.

3.1 Precise Asynchronous Multiparty Session Subtyping

Multiparty session subtyping formulates *refinement* for communication protocols with more than two communicating processes. A process implementing a session type T can be safely used whenever a process implementing one of its supertypes T' is expected. This applies in any context, ensuring that no deadlocks or other communication errors will be introduced. Replacing the implementation of T' with that of the subtype T may allow for more *optimised* communication patterns as we have seen. Ghilezan et al. [25] present the *precise* asynchronous subtyping relation $\leq$ for multiparty session processes. Precision is characterised by both *soundness* (safe process replacement is guaranteed) and *completeness* (any extension of the relation is unsound).

Crucially, asynchronous subtyping supports the optimisation of communications. Under certain conditions, the subtype can anticipate some input/output actions occurring in the supertype, performing them *earlier* than prescribed to achieve the most flexible and precise subtyping. Such reorderings can take two forms:

R1. anticipating an input from participant p before a finite number of inputs that are not from p; or

R2. anticipating an output to participant p before a finite number of inputs (from any participant), and also before other outputs that are not to p.

To denote such reorderings, two kinds of finite sequences of inputs/outputs are defined by [25], where $\mathsf{p} \neq \mathsf{q}$. $\mathcal{A}^{(\mathsf{p})}$ is a sequence containing receives from participants apart from p; $\mathcal{B}^{(\mathsf{p})}$ is a sequence containing receives from *any* participant ($\mathsf{r} = \mathsf{p}$ is allowed) and sends to participants apart from p.

$$\mathcal{A}^{(\mathsf{p})} ::= \mathsf{q}?\ell \mid \mathsf{q}?\ell.\mathcal{A}^{(\mathsf{p})} \quad \mathcal{B}^{(\mathsf{p})} ::= \ \mathsf{r}?\ell \mid \mathsf{q}!\ell \mid \mathsf{r}?\ell.\mathcal{B}^{(\mathsf{p})} \mid \mathsf{q}!\ell.\mathcal{B}^{(\mathsf{p})}$$

The *tree refinement relation* $\lesssim$ is defined coinductively on infinite session type trees that contain only single-inputs (SI) and single-outputs (SO). We leave the formal definition of $\lesssim$ in [25] and shall explain its essence with our sound algorithm. Based on $\lesssim$, the subtyping relation $\leq$ for all types (including internal and external choice) is given as

$$\frac{\forall U \in [\![T]\!]_{\mathtt{so}} \ \ \forall V' \in [\![T']\!]_{\mathtt{si}} \ \ \exists W \in [\![U]\!]_{\mathtt{si}} \ \ \exists W' \in [\![V']\!]_{\mathtt{so}} \ \ W \lesssim W'}{\mathsf{T} \leq \mathsf{T}'}$$

where $[\![T]\!]_{\mathtt{so}}$ (resp. $[\![T]\!]_{\mathtt{si}}$) is the minimal set of trees containing only single *outputs* (resp. *inputs*) of the session type tree T. Using existential quantifiers for $[\![U]\!]_{\mathtt{si}}$ and $[\![V']\!]_{\mathtt{so}}$ allows external choices to be added and internal choices to be removed (see the full version [17] for examples of $\leq$).

3.2 Our Algorithm

The precise asynchronous subtyping in [25] is *undecidable*—even when limited to two participants, as proven in [42]. This subsection introduces our *practical* algorithm which is *sound* and *terminates*, through the use of a bound on recursion.

Prefixes. We first define reduction rules for finite single-input and single-output (SISO) session type *prefixes.*

Definition 3 (Prefix reduction)**.** Let us define the syntax of *prefixes* as $\pi, \rho ::= \epsilon \mid \mathsf{p}!\ell(S) \mid \mathsf{p}?\ell(S) \mid \pi_1.\pi_2$ where "." denotes the concatenation operator. We define the reduction $\pi \to \pi'$ as the smallest relation that ensures

$$\begin{array}{rrcl}
[\to\text{I}] & \langle \mathsf{p}?\ell.\pi \llbracket \mathsf{p}?\ell.\pi' \rangle & \to & \langle \pi \llbracket \pi' \rangle \\
[\to\text{O}] & \langle \mathsf{p}!\ell.\pi \llbracket \mathsf{p}!\ell.\pi' \rangle & \to & \langle \pi \llbracket \pi' \rangle \\
[\to\mathcal{A}] & \langle \mathsf{p}?\ell.\pi \llbracket \mathcal{A}^{(\mathsf{p})}.\mathsf{p}?\ell.\pi' \rangle & \to & \langle \pi \llbracket \mathcal{A}^{(\mathsf{p})}.\pi' \rangle \\
[\to\mathcal{B}] & \langle \mathsf{p}!\ell.\pi \llbracket \mathcal{B}^{(\mathsf{p})}.\mathsf{p}!\ell.\pi' \rangle & \to & \langle \pi \llbracket \mathcal{B}^{(\mathsf{p})}.\pi' \rangle
\end{array}$$

[→I] and [→O] erase the top input and output prefixes respectively; [→$\mathcal{A}$] formalises **R1** from **§ 3.1** (permuting the input $\mathsf{p}?\ell$) while [→$\mathcal{B}$] represents **R2** (permuting the output $\mathsf{p}!\ell$).

Example 4 (Prefix reduction)**.** In Example 2, We can consider both T_Q and T_P as prefixes since they already contain no choice. The (safe) reordering T'_Q can be achieved using [→$\mathcal{B}$], where $\mathcal{B}^{(\mathsf{p})} = \mathsf{p}?\ell_1$:

$$\langle \mathsf{p}!\ell_2.\mathsf{p}?\ell_1.\mathsf{end} \llbracket \mathsf{p}?\ell_1.\mathsf{p}!\ell_2.\mathsf{end} \rangle \to \langle \mathsf{p}?\ell_1.\mathsf{end} \llbracket \mathsf{p}?\ell_1.\mathsf{end} \rangle$$

On the other hand, the (unsafe) reordering T'_P cannot be achieved with [→$\mathcal{A}$], since $\mathcal{A}^{(\mathsf{q})} = \mathsf{q}!\ell_1$ violates the definition of $\mathcal{A}^{(\mathsf{q})}$.

$$\frac{\epsilon; \varnothing \vdash_k \langle \epsilon, \mathsf{T}, n \rangle \leq \langle \epsilon, \mathsf{T}', n \rangle \quad k, n \in \mathbb{N}}{\mathsf{T} \leq \mathsf{T}'} \text{[INIT]}$$

$$\frac{}{\rho; \Sigma \vdash_k \langle \epsilon, \mathsf{end}, n \rangle \leq \langle \epsilon, \mathsf{end}, n' \rangle} \text{[END]}$$

$$\frac{\Sigma\,[\langle \pi \llbracket \mathsf{T} \rangle \leq \langle \pi' \llbracket \mathsf{T}' \rangle] = \rho \quad \mathtt{act}(\rho') \supseteq \mathtt{act}(\pi')}{\rho.\rho'; \Sigma \vdash_k \langle \pi, \mathsf{T}, n \rangle \leq \langle \pi', \mathsf{T}', n' \rangle} \text{[ASM]}$$

$$\frac{\langle \pi_1 \llbracket \pi'_1 \rangle \to \langle \pi_2 \llbracket \pi'_2 \rangle \quad \rho; \Sigma \vdash_k \langle \pi_2, \mathsf{T}, n \rangle \leq \langle \pi'_2, \mathsf{T}', n' \rangle}{\rho; \Sigma \vdash_k \langle \pi_1, \mathsf{T}, n \rangle \leq \langle \pi'_1, \mathsf{T}', n' \rangle} \text{[SUB]}$$

$$\frac{\forall i \in I.\ \forall j \in J.\ \rho.\mathsf{p}!\ell_i; \Sigma \vdash_k \langle \pi.\mathsf{p}!\ell_i, \mathsf{T}_i, n \rangle \leq \langle \pi'.\mathsf{q}?\ell_j, \mathsf{T}'_j, n' \rangle}{\rho; \Sigma \vdash_k \langle \pi, \oplus_{i \in I}\mathsf{p}!\ell_i.\mathsf{T}_i, n \rangle \leq \langle \pi', \&_{j \in J}\mathsf{q}?\ell_j.\mathsf{T}'_j, n' \rangle} \text{[OI]}$$

$$\frac{\forall i \in I.\ \exists j \in J.\ \rho.\mathsf{p}!\ell_i; \Sigma \vdash_k \langle \pi.\mathsf{p}!\ell_i, \mathsf{T}_i, n \rangle \leq \langle \pi'.\mathsf{q}!\ell_j, \mathsf{T}'_j, n' \rangle}{\rho; \Sigma \vdash_k \langle \pi, \oplus_{i \in I}\mathsf{p}!\ell_i.\mathsf{T}_i, n \rangle \leq \langle \pi', \oplus_{j \in J}\mathsf{q}!\ell_j.\mathsf{T}'_j, n' \rangle} \text{[OO]}$$

$$\frac{\forall j \in J.\ \exists i \in I.\ \rho.\mathsf{p}?\ell_i; \Sigma \vdash_k \langle \pi.\mathsf{p}?\ell_i, \mathsf{T}_i, n \rangle \leq \langle \pi'.\mathsf{q}?\ell_j, \mathsf{T}'_j, n' \rangle}{\rho; \Sigma \vdash_k \langle \pi, \&_{i \in I}\mathsf{p}?\ell_i.\mathsf{T}_i, n \rangle \leq \langle \pi', \&_{j \in J}\mathsf{q}?\ell_j.\mathsf{T}'_j, n' \rangle} \text{[II]}$$

$$\frac{\exists i \in I.\ \exists j \in J.\ \rho.\mathsf{p}?\ell_i; \Sigma \vdash_k \langle \pi.\mathsf{p}?\ell_i, \mathsf{T}_i, n \rangle \leq \langle \pi'.\mathsf{q}!\ell_j, \mathsf{T}'_j, n' \rangle}{\rho; \Sigma \vdash_k \langle \pi, \&_{i \in I}\mathsf{p}?\ell_i.\mathsf{T}_i, n \rangle \leq \langle \pi', \oplus_{j \in J}\mathsf{q}!\ell_j.\mathsf{T}'_j, n' \rangle} \text{[IO]}$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma\,[\langle \pi \llbracket \mu\mathbf{t}.\mathsf{T} \rangle \leq \langle \pi' \llbracket \mathsf{T}' \rangle \mapsto \rho] \\ \rho; \Sigma' \vdash_k \langle \pi, \mathsf{T}[\mu\mathbf{t}.\mathsf{T}/\mathbf{t}], n-1 \rangle \leq \langle \pi', \mathsf{T}', n' \rangle \qquad n > 0\end{array}}{\rho; \Sigma \vdash_k \langle \pi, \mu\mathbf{t}.\mathsf{T}, n \rangle \leq \langle \pi', \mathsf{T}', n' \rangle} [\mu\text{L}]$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma\,[\langle \pi \llbracket \mathsf{T} \rangle \leq \langle \pi' \llbracket \mu\mathbf{t}.\mathsf{T}' \rangle \mapsto \rho] \\ \rho; \Sigma' \vdash_k \langle \pi, \mathsf{T}, n \rangle \leq \langle \pi', \mathsf{T}'[\mu\mathbf{t}.\mathsf{T}'/\mathbf{t}], n'-1 \rangle \qquad n' > 0\end{array}}{\rho; \Sigma \vdash_k \langle \pi, \mathsf{T}, n \rangle \leq \langle \pi', \mu\mathbf{t}.\mathsf{T}', n' \rangle} [\mu\text{R}]$$

$$\frac{\begin{array}{c}\rho; \Sigma \vdash_{k-1} \langle \pi, \mathsf{T}, n \rangle \leq \langle \pi', \mathsf{T}', n' \rangle \\ \rho; \Sigma \vdash_{k-1} \langle \pi', \mathsf{T}', n' \rangle \leq \langle \pi'', \mathsf{T}'', n'' \rangle \quad k > 0\end{array}}{\rho; \Sigma \vdash_k \langle \pi, \mathsf{T}, n \rangle \leq \langle \pi'', \mathsf{T}'', n'' \rangle} \text{[TRA]}$$

Figure 5. Asynchronous subtyping algorithm rules.

Algorithm. We define the rules for our asynchronous subtyping algorithm in Fig. 5, following the style of [22]. Our rules use the function $\mathtt{act}(W)$, the set of input and output actions of π such that $\mathtt{act}(\epsilon) = \varnothing$, $\mathtt{act}(\mathsf{p}?\ell.\pi) = \{\mathsf{p}?\} \cup \mathtt{act}(\pi)$ and $\mathtt{act}(\mathsf{p}!\ell.\pi) = \{\mathsf{p}!\} \cup \mathtt{act}(\pi)$.

Our algorithm operates on triples of $\langle \pi, \mathsf{T}, n \rangle$, where π is a session prefix and n is a bound on the number of recursions to unroll. We keep track of ρ, a prefix containing all actions in the subtype seen so far, and Σ, a set of subtyping assumptions. Each assumption in Σ is associated with the value of ρ as it was at the time of the assumption. An additional bound k is included to limit applications of the [TRA] rule. We use our algorithm to check whether T is a subtype of T' by beginning with the [INIT] rule. If a proof derivation can be found then we conclude that $\mathsf{T} \leq \mathsf{T}'$. If not, then either $\mathsf{T} \not\leq \mathsf{T}'$ or $\mathsf{T} \leq \mathsf{T}'$ but this cannot be shown by our algorithm since $\leq$ is undecidable (hence our algorithm cannot be complete).

Our algorithm works as follows: **(1)** If both prefixes π and π' are empty and $\mathsf{T} = \mathsf{T}' = \mathsf{end}$, then we have nothing left to check and we terminate with success ([end]). **(2)** If $\langle \pi \mathrel{[\!]} \mathsf{T}\rangle \leq \langle \pi' \mathrel{[\!]} \mathsf{T}'\rangle$ is already in our set of assumptions, then we perform a check on π' to ensure that no actions have been forgotten by the subtype (see the full version [17] for more detail). If this check passes, then we terminate with success ([asm]). **(3)** We attempt to reduce the pair of prefixes $\langle \pi \mathrel{[\!]} \pi'\rangle$. If we can, then the algorithm repeats from **(1)** ([sub]). **(4)** If not, we try to pop one action from the start of both T and T' and push them to the end of π and π' respectively. If this is possible, we repeat from **(1)** ([oi,oo,ii,io]). Note that the quantifiers permit subtyping for internal and external choices. For example, [oo] says T is a subtype of T' if it has a subset of T''s internal choices (defined with $\forall i \in I.\exists j \in J$). **(5)** Otherwise, we attempt to unroll recursion in T (resp. T'), decrement the bound n (resp. n') by one and repeat from **(1)**. If the bound is already zero, we instead terminate ([μl,μr]).

Double buffering example. We show the execution of our algorithm to check the optimised type from § 2 for the kernel in the double buffering protocol (i.e. $\mathsf{T} \leq \mathsf{T}'$).

$$\mathsf{T} = \mathsf{s}!\mathit{ready}.\mathsf{T}' \quad \mathsf{T}' = \mu\mathbf{x}.\mathsf{s}!\mathit{ready}.\mathsf{s}?\mathit{copy}.\mathsf{t}?\mathit{ready}.\mathsf{t}!\mathit{copy}.\mathbf{x}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{act}(\pi_1.\pi_2.\pi_3.\pi_4) \supseteq \mathsf{act}(\epsilon)}{(\div)\quad \rho_5;\Sigma_3 \vdash \langle \epsilon, \mathsf{T}', 1\rangle \leq \langle \epsilon, \mathsf{T}'', 0\rangle}\,[\textsc{asm}]}{(\star)\quad \rho_5;\Sigma_3 \vdash \langle \pi_4, \mathsf{T}', 1\rangle \leq \langle \pi_4, \mathsf{T}'', 0\rangle}\,[\textsc{sub}]}{(\ddagger)\quad \rho_5;\Sigma_3 \vdash \langle \pi_3.\pi_4, \mathsf{T}', 1\rangle \leq \langle \pi_3.\pi_4, \mathsf{T}'', 0\rangle}\,[\textsc{sub}]}{(\dagger)\quad \rho_5;\Sigma_3 \vdash \langle \pi_2.\pi_3.\pi_4, \mathsf{T}', 1\rangle \leq \langle \pi_2.\pi_3.\pi_4, \mathsf{T}'', 0\rangle}\,[\textsc{sub}]}{\rho_5;\Sigma_3 \vdash \langle \pi_1.\pi_2.\pi_3.\pi_4, \mathsf{T}', 1\rangle \leq \langle \pi_2.\pi_3.\pi_4.\pi_1, \mathsf{T}'', 0\rangle}\,[\textsc{sub}]}{\rho_4;\Sigma_3 \vdash \langle \pi_1.\pi_2.\pi_3, \mathsf{t}!\mathit{copy}.\mathsf{T}', 1\rangle \leq \langle \pi_2.\pi_3.\pi_4, \mathsf{s}!\mathit{ready}.\mathsf{T}'', 0\rangle}\,[\textsc{oo}]}{\rho_4;\Sigma_2 \vdash \langle \pi_1.\pi_2.\pi_3, \mathsf{t}!\mathit{copy}.\mathsf{T}', 1\rangle \leq \langle \pi_2.\pi_3.\pi_4, \mathsf{T}', 1\rangle}\,[\mu\textsc{r}]}{\rho_3;\Sigma_2 \vdash \langle \pi_1.\pi_2, \mathsf{t}?\mathit{ready}.\mathsf{t}!\mathit{copy}.\mathsf{T}', 1\rangle \leq \langle \pi_2.\pi_3, \mathsf{t}!\mathit{copy}.\mathsf{T}', 1\rangle}\,[\textsc{io}]}{\rho_2;\Sigma_2 \vdash \langle \pi_1, \mathsf{T}'', 1\rangle \leq \langle \pi_2, \mathsf{t}?\mathit{ready}.\mathsf{t}!\mathit{copy}.\mathsf{T}', 1\rangle}\,[\textsc{ii}]}{\rho_1;\Sigma_2 \vdash \langle \epsilon, \mathsf{s}!\mathit{ready}.\mathsf{T}'', 1\rangle \leq \langle \epsilon, \mathsf{T}'', 1\rangle}\,[\textsc{oi}]}{(*)\quad \rho_1;\Sigma_1 \vdash \langle \epsilon, \mathsf{T}', 2\rangle \leq \langle \epsilon, \mathsf{T}'', 1\rangle}\,[\mu\textsc{l}]}{\rho_1;\Sigma_1 \vdash \langle \pi_1, \mathsf{T}', 2\rangle \leq \langle \pi_1, \mathsf{T}'', 1\rangle}\,[\textsc{sub}]}{\epsilon;\Sigma_1 \vdash \langle \epsilon, \mathsf{T}, 2\rangle \leq \langle \epsilon, \mathsf{s}!\mathit{ready}.\mathsf{T}'', 1\rangle}\,[\textsc{oo}]}{\epsilon;\varnothing \vdash \langle \epsilon, \mathsf{T}, 2\rangle \leq \langle \epsilon, \mathsf{T}', 2\rangle}\,[\mu\textsc{r}]}{\mathsf{T} \leq \mathsf{T}'}\,[\textsc{init}]$$

$$\mathsf{T}'' = \mathsf{s}?\mathit{copy}.\mathsf{t}?\mathit{ready}.\mathsf{t}!\mathit{copy}.\mathsf{T}'$$

$$\pi_1 = \mathsf{s}!\mathit{ready} \quad \pi_2 = \mathsf{s}?\mathit{copy} \quad \pi_3 = \mathsf{t}?\mathit{ready} \quad \pi_4 = \mathsf{t}!\mathit{copy}$$

$$\rho_1 = \pi_1 \quad \rho_2 = \rho_1.\pi_1 \quad \rho_3 = \rho_2.\pi_2 \quad \rho_4 = \rho_3.\pi_3 \quad \rho_5 = \rho_4.\pi_4$$

$$\Sigma_1 = \left[\langle \epsilon \mathrel{[\!]} \mathsf{T}\rangle \leq \langle \epsilon \mathrel{[\!]} \mathsf{T}'\rangle \mapsto \epsilon\right]$$

$$\Sigma_2 = \Sigma_1 \left[\langle \epsilon \mathrel{[\!]} \mathsf{T}'\rangle \leq \langle \epsilon \mathrel{[\!]} \mathsf{T}''\rangle \mapsto \mathsf{s}!\mathit{ready}\right]$$

$$\Sigma_3 = \Sigma_2 \left[\langle \pi_1.\pi_2.\pi_3 \mathrel{[\!]} \mathsf{t}!\mathit{copy}.\mathsf{T}'\rangle \leq \langle \pi_2.\pi_3.\pi_4 \mathrel{[\!]} \mathsf{T}'\rangle \mapsto \rho_4\right]$$

$$(\dagger) = \langle \pi_1.\pi_2.\pi_3.\pi_4 \mathrel{[\!]} \pi_2.\pi_3.\pi_4.\pi_1\rangle \rightarrow \langle \pi_2.\pi_3.\pi_4 \mathrel{[\!]} \pi_2.\pi_3.\pi_4\rangle\ [\rightarrow\mathcal{A}]$$

$$(\ddagger) = \langle \pi_2.\pi_3.\pi_4 \mathrel{[\!]} \pi_2.\pi_3.\pi_4\rangle \rightarrow \langle \pi_3.\pi_4 \mathrel{[\!]} \pi_3.\pi_4\rangle\ [\rightarrow\textsc{i}]$$

$$(\star) = \langle \pi_3.\pi_4 \mathrel{[\!]} \pi_3.\pi_4\rangle \rightarrow \langle \pi_4 \mathrel{[\!]} \pi_4\rangle\ [\rightarrow\textsc{i}]$$

$$(*) = \langle \pi_1 \mathrel{[\!]} \pi_1\rangle \rightarrow \langle \epsilon \mathrel{[\!]} \epsilon\rangle\ [\rightarrow\textsc{o}] \quad (\div) = \langle \pi_4 \mathrel{[\!]} \pi_4\rangle \rightarrow \langle \epsilon \mathrel{[\!]} \epsilon\rangle\ [\rightarrow\textsc{o}]$$

See also the full version [17] for the ring and alternating bit protocols, which include internal and external choices.

Properties. We prove the correctness and complexity of our algorithm. See the full version [17] for the proofs. We define the size of prefixes as $|\epsilon| = 0$ and $|\mathsf{p}?\ell.\pi| = |\mathsf{p}!\ell.\pi| = 1 + |\pi|$.

Lemma 5. *Given finite prefixes π and π', $\langle \pi \mathrel{[\!]} \pi'\rangle$ can be reduced only a finite number of times.*

Theorem 6 (Termination)**.** *Our subtyping algorithm always eventually terminates.*

Theorem 7 (Soundness)**.** *Our subtyping algorithm is sound.*

Lemma 8. *Given finite prefixes π and π', the time complexity of reducing $\langle \pi \mathrel{[\!]} \pi'\rangle$ is $O(\min(|\pi|, |\pi'|))$.*

Theorem 9 (Complexity)**.** *Consider T and T' as (possibly infinite) trees $\mathcal{T}(\mathsf{T})$ and $\mathcal{T}(\mathsf{T}')$ with* asymptotic branching factors *b and b' respectively [21, 39]. Our algorithm has time complexity $O(n \min(b, b')^n)$ and space complexity $O(n \min(b, b'))$ in the worst case to determine if $\mathsf{T} \leq \mathsf{T}'$ with bound n.*

Algorithm implementation. In practice, we make some minor alterations to the algorithm when implementing it in Rumpsteak. As outlined in § 2, Rumpsteak acts on FSMs rather than local types so we modify our bounds-checking accordingly. We also represent prefixes as lazily-removable lists for greater memory efficiency—this additionally allows a slightly simplified termination condition in the case of [asm]. Finally, we provide some opportunities for the algorithm to "short circuit" in cases where we can tell early that a subtype is not valid. See the full version [17] for more details.

4 Evaluation

In this section, we evaluate how Rumpsteak performs with respect to existing tools. First, in § 4.1, we evaluate the runtime performance of programs written in Rumpsteak versus the same benchmarks implemented using other Rust session type tools. In the spirit of Rust's emphasis on efficiency, this runtime performance is of particular significance for developers. Secondly, in § 4.2, we evaluate how Rumpsteak's verification of message reordering (our subtyping algorithm) scales compared to existing verification tools. Although not a runtime cost, subtyping is known to be a computationally challenging problem that often scales poorly.

4.1 Session-Based Rust Implementations

We compare Rumpsteak's runtime against three other session type implementations in Rust: **(1)** Sesh [38], a synchronous implementation of binary session types; **(2)** Ferrite [12], an implementation of shared binary session types [5] supporting asynchronous execution; and **(3)** MultiCrusty [41], a synchronous MPST implementation based on Sesh.

We perform a series of benchmarks shown in Fig. 6. We execute these using a 16-core AMD Opteron™ 6200 Series

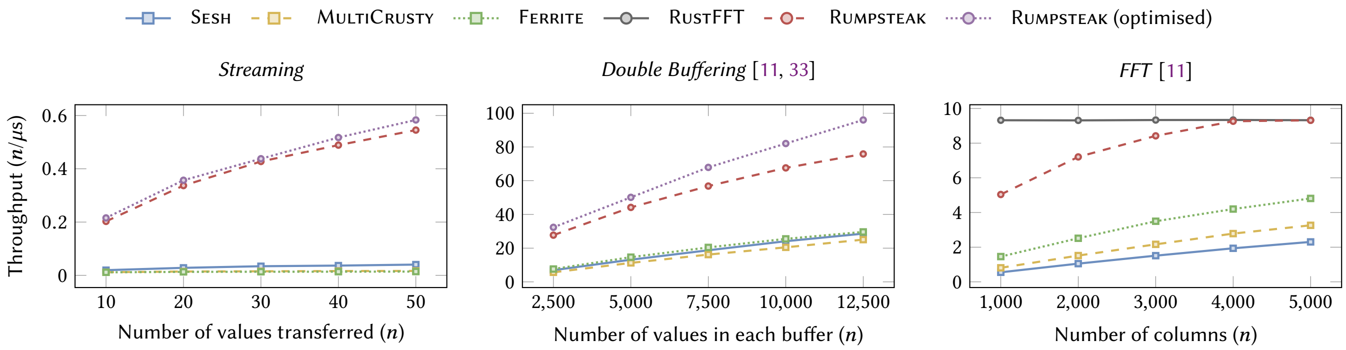


Figure 6. Benchmarking Rumpsteak's runtime performance against previous work in Rust (raw data in the full version [17]).

CPU @ 2.6GHz with hyperthreading, 128GB of RAM, Ubuntu 18.04.5 LTS and Rust Nightly 2021-07-06. We use version 0.3.5 of the Criterion.rs library [27] to perform microbenchmarking and a *multi-threaded* asynchronous runtime from version 1.11.0 of the Tokio library [59].

Streaming. This protocol has two participants, a source s and a sink t, with a combination of recursion and choice. As stated previously, its global type G_{ST} is given as

$$G_{ST} = \mu \mathbf{x}.\mathtt{t} \to \mathtt{s} : \{\mathit{ready}.\mathtt{s} \to \mathtt{t} : \{\mathit{value}.\mathbf{x}, \quad \mathit{stop}.\mathtt{end}\}\}$$

We benchmark the protocol by varying the number of *value*s that are sent before the exchange *stop*s. We use AMR to create an optimised version for Rumpsteak: if the source knows it will send at least n *value*s before *stop*ping, then these messages can be *unrolled* and sent all at once—only receiving *ready* from the sink after sending all n *value*s. For benchmarking, we unroll the first 5 *value*s in the optimised version.

Our results show Rumpsteak reaches around 14.5x the throughput of other implementations. At first, it is limited by channel creation overheads, the cost of which becomes less significant as more messages are sent. The throughput levels off as message passing overheads become the bottleneck. The optimised version eases this overhead, as the source is less frequently blocked on receiving *ready* from the sink—this effect could be increased by unrolling more messages.

Sesh and MultiCrusty have much lower throughputs since they use more expensive synchronous communication. These implementations also create a new channel for each interaction. This causes their throughput to stay constant as there are fewer one-time costs to spread than in Rumpsteak. Interestingly, Ferrite performs similarly to Sesh and MultiCrusty, despite employing asynchronous execution like Rumpsteak. This is because its design requires that the source and sink are implemented using *recursion* rather than *iteration*. Ferrite also requires particularly strong safety requirements at compile time. Therefore, the sink's output buffer must be guarded with a mutex instead of being accessed directly.

Double Buffering. We benchmark our running example by performing only two iterations. This allows for both of the kernel's buffers to be filled and, importantly, for the protocol to terminate. Sesh and Ferrite do not support MPST so their implementations use binary session types between pairs of participants. This approach does *not* provide the same safety as MPST and so we cannot verify deadlock-freedom as we can in MultiCrusty and Rumpsteak (see Table 1).

We parameterise the size of the buffers and measure the throughput. Performance is similar to that of the stream benchmark; Rumpsteak's throughput reaches around 3.2x that of the others. Moreover, we confirm the intuition described in **§ 1**. Increasing the size of the buffers does generally improve throughput. However, it remains limited when only a single buffer can be used at once. Fig. 6 shows that optimising the kernel as described in **§ 2** increases the throughput since the source and sink operate on different buffers at once.

FFT. The fast Fourier transform (FFT) is an algorithm for computing the discrete Fourier transform of a vector. We take $n \times 8$ *matrices*, where the FFT is computed on each column to produce a new, transformed matrix, and implement the Cooley-Tukey FFT algorithm which divides the problem into—in our case—eight. Each of these problems can be solved independently by different processes, which communicate with one another using message passing. The exchanges between participants are described in [11, Fig. 7].

We have chosen to use FFT as a benchmark as it is a standard problem with numerous implementations, and we can therefore compare implementations based on MPST with actual high-optimised implementations.

We implement a concurrent version of the algorithm in Sesh, Ferrite, MultiCrusty and Rumpsteak, which uses eight processes. Each process works on an array of n inputs, representing one column. Arithmetic operations are performed in a pairwise fashion on two of these arrays.

We use the same approach as in the double buffering protocol to write the binary implementations for Sesh and Ferrite. Interestingly, to represent this as a sequence of binary

Table 1. Expressiveness of Rumpsteak compared to previous work.

Protocol	n	C	R	IR	AMR	Sesh	Ferrite	MultiCrusty	Rumpsteak	k-MC	SoundBinary
Two Adder [2]	2	✔	✔			✔	✔	✔	✔	✔	✔
Three Adder	3					✗	✗	✔	✔	✔	✘
Streaming	2	✔	✔			✔	✔	✔	✔	✔	✔
Optimised Streaming	2	✔	✔		✔	✘	✘	✘	✔	✔	✔
Ring [11]	3		✔	✔		✗	✗	✔	✔	✔	✘
Optimised Ring [11]	3		✔	✔	✔	✗	✗	✗	✔	✔	✘
Ring With Choice [11]	3	✔	✔	✔		✗	✗	✔	✔	✔	✘
Optimised Ring With Choice [11]	3	✔	✔	✔	✔	✗	✗	✗	✔	✔	✘
Double Buffering [11]	3		✔	✔		✗	✗	✔	✔	✔	✘
Optimised Double Buffering [11, 33]	3		✔	✔	✔	✘	✘	✘	✔	✔	✘
Alternating Bit [1, 43]	2	✔	✔	✔		✗	✗	✗	✔	✔	✔
Elevator [6, 43]	3	✔	✔	✔	✔	✘	✘	✘	✔	✔	✘
FFT [11]	8					✗	✗	✔	✔	✔	✘
Optimised FFT [11]	8				✔	✘	✘	✘	✔	✔	✘
Authentication [48]	3	✔				✗	✗	✔	✔	✔	✘
Client-Server Log [41]	3	✔	✔	✔		✗	✗	✔	✔	✔	✘
Hospital [7]	2	✔	✔	✔	✔	✘	✘	✘	✗	✘	✔

n Number of participants **C** Choice **R** Recursion **IR** Infinite recursion **AMR** Asynchronous message reordering
✔ Expressible ✗ Expressible using endpoint types (but without deadlock-freedom guarantee) ✘ Not expressible

sessions, we require additional synchronisation of all participants at each stage of the protocol. While we can perform AMR to Rumpsteak's version, in practice we found that this does not have much effect since the protocol is already heavily synchronised so cannot progress any faster.

We benchmark the protocol by varying the number of columns in the matrix. We also compare these implementations with the most downloaded open-source Rust FFT implementation, RustFFT [60]. RustFFT does not use concurrency and computes the FFT of a matrix by iteratively performing the Cooley-Tukey algorithm on each column.

Ferrite performs better than before since its FFT implementation does not suffer from the limitations explained previously. Like Rumpsteak, it benefits from the use of asynchronous execution, although the additional synchronisation from representing the problem as a set of binary sessions causes Rumpsteak's throughput to remain around 1.9x greater.

Most excitingly, Rumpsteak achieves RustFFT's throughput for large matrices. RustFFT's implementation is highly tuned for low-level efficiency and does not incur any overheads associated with message passing. Moreover, asynchronous executors are generally designed for higher-level tasks such as networking; MPI-based communication [44] would more likely be used to parallelise this problem in practice. Therefore, we think it impressive that even a basic parallel implementation using Rumpsteak can reach the same performance as a state-of-the-art sequential implementation.

Expressiveness. Table 1 discusses the expressiveness of Rumpsteak compared with Sesh, Ferrite and MultiCrusty. Since Sesh and Ferrite support only binary session types, they are unable to guarantee deadlock-freedom in protocols with more than two participants. MultiCrusty has greater expressiveness than Sesh and Ferrite since it implements MPST but, unlike Rumpsteak, still cannot ensure deadlock-freedom for protocols optimised using AMR. In addition, many optimisations, like the ones we have benchmarked, break duality between pairs of participants so are not expressible at all by Sesh, Ferrite and MultiCrusty. Meanwhile, our powerful API and new subtyping algorithm allow Rumpsteak to express many examples using AMR.

4.2 Verifying Asynchronous Message Reordering

We perform a second set of benchmarks, shown in Fig. 7, to evaluate Rumpsteak's asynchronous subtyping algorithm. We compare it against **(1)** SoundBinary [7], a sound subtyping algorithm defined for *binary* session types only; and **(2)** k-MC [43], an algorithm for directly checking the compatibility of a set of FSMs without the need for a global type. Compatibility is checked up to a bound k on the size of each process' asynchronous queue. We note that neither SoundBinary nor k-MC provide a runtime framework like Rumpsteak does, they are used only to verify the AMR.

We benchmark with the same machine as before. SoundBinary and k-MC are written in Haskell so we must run each tool's binary rather than simply timing Rust functions. We, therefore, provide a command-line for Rumpsteak's subtyping algorithm so it is comparable with these other tools. Rumpsteak's binary is compiled with Rust 1.54.0 and we use Hyperfine [50] to compare the execution time for each tool.

Streaming (from § 4.1). We vary n, the number of *values* we unroll. Using SoundBinary and Rumpsteak, we check

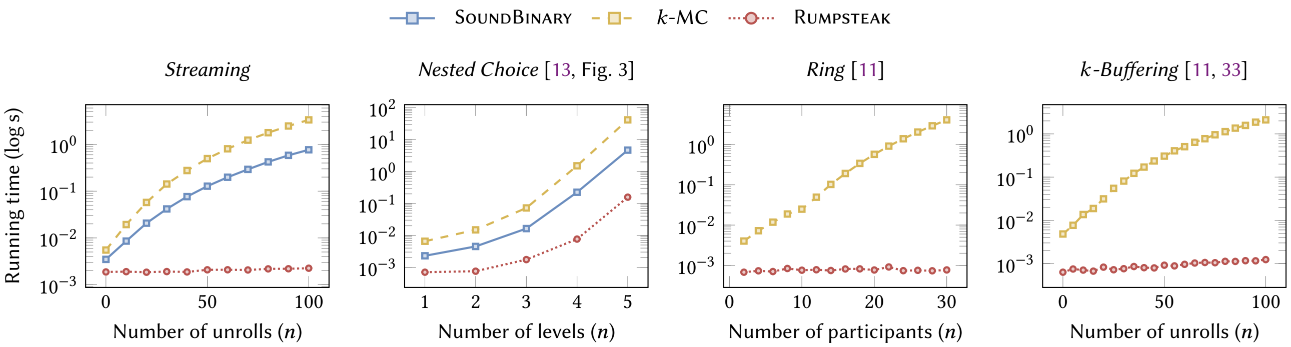


Figure 7. Benchmarking Rumpsteak's subtyping performance against previous work (raw data in the full version [17]).

that the optimised source is a subtype of its projected version and, using k-MC, we check that the optimised source is compatible with the sink. Our results show that Rumpsteak scales significantly better than both SoundBinary and k-MC. While all three implementations can verify up to 100 unrolls in under a second, the execution time taken by SoundBinary and k-MC increases exponentially while Rumpsteak's remains mostly flat. This is consistent with our complexity analysis in Theorem 9, since unrolling more *value*s does not increase the branching factor of the subtype.

Nested choice. We next consider a protocol from Chen et al. [13, Fig. 3] containing nested choice. We perform this nesting up to a parameterised number of levels n to increase the complexity. Specifically, we check that $\mathsf{T}_n \leq \mathsf{T}'_n$ where

$$\mathsf{T}_0 = \mathsf{T}'_0 = \mathsf{end}$$
$$\mathsf{T}_{n+1} = !m.(?r.\mathsf{T}_n \,\&\, ?s.\mathsf{T}_n \,\&\, ?u.\mathsf{T}_n) \oplus !p.(?r.\mathsf{T}_n \,\&\, ?s.\mathsf{T}_n)$$
$$\mathsf{T}'_{n+1} = ?r.(!m.\mathsf{T}'_n \oplus !p.\mathsf{T}'_n \oplus !q.\mathsf{T}'_n) \,\&\, ?s.(!m.\mathsf{T}'_n \oplus !p.\mathsf{T}'_n)$$

We find that Rumpsteak again performs more scalably than SoundBinary and k-MC. Here, Rumpsteak's improved efficiency is significant—for five levels, k-MC takes around 40s while Rumpsteak requires only a fraction of a second. Nonetheless, we observe that Rumpsteak also exhibits exponential performance, in contrast to our stream benchmark. This is because our algorithm bounds recursion but not choice, causing an explosion in the paths to visit as the number of nested choices increases. This is also explained by our complexity analysis in Theorem 9 since a greater number of nested choices increases the depth to which we must explore.

Ring [11]. We consider a ring protocol with a parameterised number of participants. Each participant (aside from the first, which initiates the protocol) receives a value from its preceding neighbour then sends a value to its succeeding neighbour. Providing that this receive is not dependent on the send, we can use AMR to send before receiving.

We benchmark verifying this optimisation using both k-MC and Rumpsteak. In this case, we cannot use SoundBinary since the protocol is multiparty. While Rumpsteak can verify each participant's subtype individually, k-MC must consider the entire protocol at once. Unsurprisingly, k-MC is significantly less scalable as its running time grows exponentially, whereas Rumpsteak's performance remains mostly constant.

***k*-buffering.** We extend the double buffering protocol to a parameterised number of buffers. We benchmark verifying the optimisation to the kernel discussed previously. Crucially, with a greater number of buffers, we can unroll a greater number of ready messages. We again compare only k-MC and Rumpsteak since the protocol is multiparty. Similarly to other benchmarks, Rumpsteak scales more efficiently than k-MC.

Expressiveness. Table 1 discusses the expressiveness of Rumpsteak's algorithm against SoundBinary and k-MC (note again that SoundBinary and k-MC only verify AMR and do not provide language implementations like Rumpsteak). As we discussed, SoundBinary supports only two parties so cannot express many of the case studies. However, it can express some unbounded binary protocols that Rumpsteak and k-MC cannot, such as the Hospital example [7, **§ 1**][1]. Unsurprisingly, k-MC's expressiveness in the table coincides with Rumpsteak's as k-MC with k=∞ is equivalent to the liveness property induced by asynchronous multiparty subtyping [25, Theorem 7.3]. On the other hand, k-MC can verify a wider syntax of local FSMs than those corresponding to Definition 1.

5 Conclusion and Related Work

There are a vast number of studies on session types, some of which are implemented in programming languages [4] and tools [24]. The top-down approach using Scribble [2, 54, 62] we presented in this paper has been implemented for a number of other programming languages such as Java [31, 32, 40], Go [10], TypeScript [45], Scala [53], MPI-C [49], Erlang [47], Python [18], F# [46], F★ [63] and Actor DSL [26]. Several implementations use an EFSM-based approach to generate

[1] Notice, for the case of Rumpsteak, that we can manually write the endpoints, and the framework checks the conformance to the protocol. However, Rumpsteak cannot verify the *deadlock-freedom* property of the protocol, hence the *amber cross*.

APIs from Scribble for target programming languages such as [10, 31, 32, 45, 46, 53, 63] to ensure correctness by construction, but none are integrated with AMR like Rumpsteak.

Rumpsteak provides three approaches for gaining *efficiency* and ensuring *deadlock-freedom* in message-passing Rust applications: (1) the *top-down* approach for ensuring *correctness/safety by construction* and maximising *asynchrony* with local analysis; (2) the *bottom-up* approach using global analysis on a set of FSMs; and (3) the *hybrid* approach, which combines inference with local analysis. All three approaches are backed by Rumpsteak's API (described in **§ 2**), which uses Rust's affine type system to ensure protocol compliance.

Note that Rumpsteak is the first framework (for any programming language) to enable the bottom-up (by integrating with k-MC [43]) and the hybrid approaches.

Rust session type implementations. We closely compared the performance and expressiveness of Rumpsteak with three existing works in **§ 4**: **(1)** Sesh [38], a synchronous implementation of binary session types; **(2)** Ferrite [12], an implementation of shared binary session types [5] supporting asynchronous execution; and **(3)** MultiCrusty [41], a synchronous MPST implementation based on Sesh.

Of these previous implementations, only MultiCrusty can also support MPST with deadlock-freedom. However, to represent a multiparty session, MultiCrusty requires defining a tuple of binary sessions for each role as well as their order of use. This leads to a more complex and less intuitive API than Rumpsteak's. Our API requires fewer definitions and is also both more expressive and performant (see **§ 4**).

Rumpsteak and Ferrite support asynchronous execution, which is more efficient than the synchronous and blocking communication used by Sesh and MultiCrusty. However, only Rumpsteak supports Rust's idiomatic `async`/`await` syntax. Ferrite instead requires nesting sequential communication, which is more verbose and less efficient. Iteration cannot be easily expressed (see **§ 4**) and it requires stricter compile-time concurrency guarantees than Rumpsteak. We do not compare directly against [34] (another synchronous implementation of binary session types) as this uses an older edition of Rust and has several limitations already discussed in [38, 41].

Rumpsteak can perform AMR, which is not possible in any of these existing implementations. Combined with its straightforward design and use of asynchronous execution, we found that it is therefore much more efficient than all previous work. It has around 14.5x, 3.2x and 1.9x greater throughput than the next fastest implementation in the stream, double buffering and FFT protocols respectively. Rumpsteak can also compete with a popular Rust implementation in the FFT benchmark, achieving the same throughput for large input sizes.

Recent work [20] builds a DSL to offer protocol conformance for Rust, supported by typestates. They do not yet explore combining the DSL with communication channels like Rumpsteak and [5, 34, 38, 41]. If this is achieved, it would be interesting to integrate session type tooling with their DSL and compare this with Rumpsteak.

Verification of asynchronous subtyping. Interest in AMR has grown recently, both in theory and practice. An extension of the Iris framework, Actris [35], formalises a variant of binary asynchronous subtyping, which is in turn implemented in the Coq proof assistant. The asynchronous session subtyping defined in [13, 25] is *precise* but was shown to be undecidable, even for binary sessions [9, 42]. Hence, in general, checking $\mathsf{M}'_i \leq \mathsf{M}_i$ is undecidable. Various limited classes of session types for which $\mathsf{M}'_i \leq \mathsf{M}_i$ is decidable [7, 8, 11, 42] are proposed but they are not applicable to our use cases since **(1)** the relations in [7, 9, 42] are *binary* and the same limitations do not work for multiparty sessions; and **(2)** the relation in [11, Def. 6.1] does not handle subtyping across unrolling recursions, e.g. the relation is inapplicable to the double buffering algorithm [33] (see [11, Remark 8.1]).

Our new algorithm for asynchronous optimisation is *terminating* (see Theorem 6), *sound* (see Theorem 7) and capable of verifying optimisations in a range of classical examples (see **§ 4**). It is also more performant than a global analysis based on k-MC, and an existing two-party sound algorithm [7] (see **§ 4**). This is because our algorithm is implemented efficiently (see **§ 3.2**) and can often execute in linear rather than exponential time (see Theorem 9).

Deadlock detection. There is also recent progress [51, 52] on static deadlock detection in Rust by tracking locks and unlocks in Rust's intermediate representation (MIR) [58]. This targets shared memory, rather than message-passing applications, and does not attempt safety by construction. In future work, we could also analyse the MIR directly in Rumpsteak, replacing the use of our API. However, it is difficult to identify concurrency primitives in the MIR (this would be made harder with `async`/`await`); [51] currently supports only three concurrent data structures from Rust's ecosystem.

Acknowledgments

We thank (in alphabetical order) Julia Gabet, Paul Kelly, Roland Kuhn, Nicolas Lagaillardie, Julien Lange, and Fangyi Zhou, for their valuable proofreading, feedback and advice on early versions of this work. We also thank the PPoPP reviewers for the helpful comments in their reviews.

The work is supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T014709/1, EP/V000462/1, and NCSS/EPSRC VeTSS.

A Artifact

A.1 Content of the artifact

The artifact [16] contains the following files:

```
.
|-- Artifact.md
|-- Artifact.pdf
|-- Dockerfile
|-- gen_fig_6.sh
|-- gen_fig_7.sh
`-- getting_started.sh
```

Note that an internet connection is required to use the artifact.

A.2 Getting started guide (Docker artifact)

In this subsection, we will run the benchmarks used to produce Figs. 6 and 7 of the paper in a Docker container. At the end of this section, you should be have all the plots of those figures.

> Note that the benchmark results may not match those in Figs. 6 and 7 when run in a Docker container or a machine with different specification than used in the paper. See the subsection on *Claims supported or not by the artifact* for more discussion of this.

To run the getting started guide, we assume you are running a Linux machine with Docker and Gnuplot installed, as well as standard Unix tools (`tail`, `awk`, `cut`, etc.).

This *Getting started* subsection is fully automated: extract the archive and run the `getting_started.sh` script. *The script takes a long time to run all the benchmarks. On the author's laptop, it took approximately 2 hours and 15 minutes.* When the script finishes, you should have a few `*.png` plots which correspond to the plots shown in Figs. 6 and 7 of the paper.

```
$ cd /path/to/extracted/artifact/
$ ./getting_started.sh
```

Once run, to clean-up your system, in addition to removing the archive folder, you should remove the docker image (the following command assumes you don't have other docker images/containers on your system):

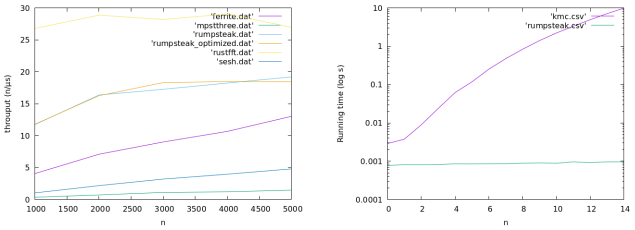
```
$ docker rmi rumpsteak_tool
$ docker system prune
```

A.2.1 Output.

The `getting_started.sh` generates figures similar to the one used in the paper[2].

We show in Fig. 8 the figures `fft.png` and `ring.png` that are the equivalent of Fig. 6 (subfigure FFT) and Fig. 7 (subfigure Ring) in the paper.

A.2.2 Analysing the raw data.



(a) Figure `fig_6_fft.png` generated by the artifact, which corresponds to Example FFT in Fig. 6 in the paper.

(b) Figure `fig_7_ring.png` generated by the artifact, which corresponds to Example Ring in Fig. 7 in the paper.

Figure 8. Two examples of the figures generated. Note that those figure are obtain from a different run of the artifact, in a lower-end computer, which explains the differences with figures presented in the paper.

```
.
|-- double_buffering
|   |-- ...
|-- fft
|   |-- ...
|-- report
|   `-- index.html
`-- stream
    |-- ...
```

Figure 9. The files created by the Criterion benchmark (in `results/criterion/`). A detailed report is available by viewing `index.html` in a browser.

Runtime benchmarks. The results used to generate Fig. 6 are generated with Criterion. Criterion produces the tree of files shown in Fig. 9. The report can be viewed in a web brower.

Subtyping benchmarks. Hyperfine generates a CSV file (located at `results/data/*/*.csv`) for the results of each benchmark. For each row, this file contains

1. `command`: the actual command that was run;
2. `mean`: the mean time to execute the command;
3. `stddev`: the standard deviation of executing the command;
4. `median`: the median time to execute the command;
5. `user`: the mean *user* time to run the command;
6. `system`: the mean *system* time to run the command;
7. `min`: the minimum time to run the command;
8. `max`: the maximum time to run the command; and
9. `parameter_n`: the value of the parameter used in the command

where all times are given in seconds. Since we do not care about execution time spent in the kernel (all relevant computation is done in user space) we take only the timings

[2] They are not exactly the same in the sense that they are not produced by the same tool (Tikz vs. Gnuplot), but the data they use is produced similarly.

in the user column. We provide a script to plot the mean user execution time against the parameter value for each tool.

A.3 Claims supported or not by the artifact

A.3.1 Claims supported by the artifact.

- Results presented in Fig. 7 are fairly stable across different machines, even when run in Docker. One should expect to obtain similar plots on their machine.

A.3.2 Claims partially supported by the artifact.

- Results presented in Fig. 6 are quite dependent on the number of cores provided by the machine and on other programs being run on the machine simultaneously. The results presented in the paper are run on a 16-core AMD Opteron 6200 Series at 2.6GHz with hyperthreading and 128GB of RAM.

  Attempts run on lower-end processors or from within a Docker container may not provide similar results (in particular, the rustfft implementation is highly optimised for sequential execution and outperforms approaches based on message passing on processors with fewer cores). Nonetheless, the performance of Rumpsteak vs. other MPST implementations would remain mostly comparable.

A.3.3 Claims not supported by the artifact.

- NuScr is an external tool not contributed by the authors, and therefore not part of the claims of the paper.

References

[1] [n.d.]. Introduction to Protocol Engineering. http://cs.uccs.edu/~cs522/pe/pe.htm

[2] [n.d.]. *νScr*. https://github.com/nuscr/nuscr

[3] [n.d.]. *Rumpsteak*. https://github.com/zakcutner/rumpsteak

[4] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[5] Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–29. https://doi.org/10.1145/3110281

[6] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. 2018. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In *Computer Aided Verification (LNCS, Vol. 10982)*. Springer, 372–391. https://doi.org/10.1007/978-3-319-96142-2_23

[7] Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. 2021. A Sound Algorithm for Asynchronous Session Subtyping and its Implementation. *Logical Methods in Computer Science* 17 (2021). Issue 1. https://doi.org/10.23638/LMCS-17(1:20)2021 (repository is found at https://github.com/julien-lange/asynchronous-subtyping).

[8] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2017. Undecidability of Asynchronous Session Subtyping. *Information and Computation* 256, C (2017), 300–320. https://doi.org/10.1016/j.ic.2017.07.010

[9] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2018. On the Boundary Between Decidability and Undecidability of Asynchronous Session Subtyping. *Theoretical Computer Science* 722 (2018), 19–51. https://doi.org/10.1016/j.tcs.2018.02.010

[10] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming Using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30. https://doi.org/10.1145/3290342

[11] David Castro-Perez and Nobuko Yoshida. 2020. CAMP: Cost-Aware Multiparty Session Protocols. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428223

[12] Ruofei Chen and Stephanie Balzer. 2021. Ferrite: A Judgmental Embedding of Session Types in Rust. arXiv:2009.13619 (repository is found at https://github.com/ferrite-rs/ferrite).

[13] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *Logical Methods in Computer Science* 13 (2017). Issue 2. https://doi.org/10.23638/LMCS-13(2:12)2017

[14] Jaemin Choi, David F. Richards, and Laxmikant V. Kale. 2020. Achieving Computation-Communication Overlap with Overdecomposition on GPU Systems. In *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 1–10. https://doi.org/10.1109/ESPM251964.2020.00006

[15] Alex Crichton. [n.d.]. *Futures*. https://github.com/rust-lang/futures-rs

[16] Zak Cutner, Nobuko Yoshida, and Martin Vassor. 2021. *Artifact: Deadlock-Free Asynchronous Message Reoerdering in Rust with Multiparty Session Types*. https://doi.org/10.5281/zenodo.5786034

[17] Zak Cutner, Nobuko Yoshida, and Martin Vassor. 2021. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. (repository is found at https://arxiv.org/abs/2112.12693).

[18] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2015. Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. *FMSD* (2015), 1–29.

[19] Ryan Donovan. 2020. *Why the Developers Who Use Rust Love It so Much*. https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/

[20] José Duarte and António Ravara. 2021. Retrofitting Typestates into Rust. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity (SBLP)*. ACM. https://github.com/rustype/typestate-rs/blob/main/paper/sblp21.pdf (to appear).

[21] Stefan Edelkamp and Richard E. Korf. 1998. The Branching Factor of Regular Search Spaces. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI)*. AAAI, 299–304. https://www.aaai.org/Papers/AAAI/1998/AAAI98-042.pdf

[22] Simon Gay and Malcolm Hole. 2005. Subtyping for Session Types in the Pi Calculus. *Acta Informatica* 42, 2–3 (2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

[23] Simon Gay and António Ravara. 2017. Behavioural Types: from Theory to Tools. 1–412. https://doi.org/10.13052/rp-9788793519817

[24] Simon Gay and António Ravara. 2017. *Behavioural Types: from Theory to Tools*. River Publisher. 1–412 pages. https://doi.org/10.13052/rp-9788793519817

[25] Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. 2021. Precise Subtyping for Asynchronous Multiparty Sessions. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434297

[26] Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. 2021. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In *35th European Conference on Object-Oriented Programming (LIPIcs, Vol. 194)*. Schloss Dagstuhl, 10:1–10:30. https://doi.org/10.4230/LIPIcs.ECOOP.2021.10

[27] Brook Heisler. [n.d.]. *Criterion.rs*. https://github.com/bheisler/criterion.rs

[28] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *Programming Languages and Systems (LNCS, Vol. 1381)*. Springer, 122–138. https://doi.org/10.1007/bfb0053567

[29] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 273–284. https://doi.org/10.1145/1328438.1328472

[30] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *Journal of the ACM* 63, 1 (2016), 1–67. https://doi.org/10.1145/2827695

[31] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification through Endpoint API Generation. In *Fundamental Approaches to Software Engineering (LNCS, Vol. 9633)*. Springer, 401–418. https://doi.org/10.1007/978-3-662-49665-7_24

[32] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *Fundamental Approaches to Software Engineering (LNCS, Vol. 10202)*. Springer, 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

[33] Hai Huang, Padmanabhan Pillai, and Kang G. Shin. 2002. Improving Wait-Free Algorithms for Interprocess Communication in Embedded Real-Time Systems. In *2002 USENIX Annual Technical Conference*. USENIX Association. https://www.usenix.org/legacy/events/usenix02/huang.html

[34] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming (WGP)*. ACM, 13–22. https://doi.org/10.1145/2808098.2808100

[35] Jonas Kastberg Hinrichsen. 2021. *Sessions and Separation*. Ph.D. Dissertation. IT University of Copenhagen, Copenhagen. http://itu.dk/people/jkas/papers/thesis.pdf

[36] Ralf Jung. 2020. Understanding and evolving the Rust programming language. https://doi.org/10.22028/D291-31946

[37] Ki-Hwan Kim and Q-Han Park. 2012. Overlapping Computation and Communication of Three-Dimensional FDTD on a GPU Cluster. *Computer Physics Communications* 183, 11 (2012), 2364–2369. https://doi.org/10.1016/j.cpc.2012.06.003

[38] Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science* 304 (2019), 48–60. https://doi.org/10.4204/eptcs.304.4 (repository is found at https://github.com/wenkokke/sesh).

[39] Richard E. Korf. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence* 27, 1 (1985), 97–109. https://doi.org/10.1016/0004-3702(85)90084-0

[40] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking Protocols with Mungo and StMungo: A Session Type Toolchain for Java. *Science of Computer Programming* 155 (2018), 52–75. https://doi.org/10.1016/j.scico.2017.10.006

[41] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2020. Implementing Multiparty Session Types in Rust. In *Coordination Models and Languages (LNCS, Vol. 12134)*. Springer, 127–136. https://doi.org/10.1007/978-3-030-50029-0_8 (repository is found at https://github.com/NicolasLagaillardie/mpst_rust_github).

[42] Julien Lange and Nobuko Yoshida. 2017. On the Undecidability of Asynchronous Session Subtyping. In *Foundations of Software Science and Computation Structures (LNCS, Vol. 10203)*. Springer, 441–457. https://doi.org/10.1007/978-3-662-54458-7_26

[43] Julien Lange and Nobuko Yoshida. 2019. Verifying Asynchronous Interactions via Communicating Session Automata. In *Computer Aided Verification (LNCS, Vol. 11561)*. Springer, 97–117. https://doi.org/10.1007/978-3-030-25540-4_6 (repository is found at https://bitbucket.org/julien-lange/kmc-cav19).

[44] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (version 4.0).

[45] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2021. Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*. ACM, 94–106. https://doi.org/10.1145/3446804.3446854

[46] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A Session Type Provider: Compile-time API Generation of Distributed Protocols with Refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction (CC)*. ACM, 128–138. https://doi.org/10.1145/3178372.3179495

[47] Rumyana Neykova and Nobuko Yoshida. 2017. Let It Recover: Multiparty Protocol-Induced Recovery. In *26th International Conference on Compiler Construction*. ACM, 98–108.

[48] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. 2013. SPY: Local Verification of Global Protocols. In *Runtime Verification (LNCS, Vol. 8174)*. Springer, 358–363. https://doi.org/10.1007/978-3-642-40787-1_25

[49] Nicholas Ng, Jose Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. 2015. Protocols by Default. In *Compiler Construction (LNCS, Vol. 9031)*. Springer, 212–232. https://doi.org/10.1007/978-3-662-46663-6_11

[50] David Peter. [n.d.]. *Hyperfine*. https://github.com/sharkdp/hyperfine

[51] Boqin Qin. 2020. `rust-lock-bug-detector`*: Statically Detect Double-Lock & Conflicting-Lock Bugs on MIR*. https://github.com/BurtonQin/rust-lock-bug-detector

[52] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 763–779. https://doi.org/10.1145/3385412.3386036

[53] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *31st European Conference on Object-Oriented Programming (LIPIcs, Vol. 74)*. Schloss Dagstuhl, 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

[54] Scribble Authors. [n.d.]. *Scribble: Describing Multi Party Protocols*. http://www.scribble.org/

[55] Marc Sergent, Mario Dagrada, Patrick Carribault, Julien Jaeger, Marc Pérache, and Guillaume Papauré. 2018. Efficient Communication/Computation Overlap with MPI+OpenMP Runtimes Collaboration. In *Parallel Processing (LNCS, Vol. 11014)*. Springer, 560–572. https://doi.org/10.1007/978-3-319-96983-1_40

[56] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *Parallel Architectures and Languages Europe (LNCS, Vol. 817)*. Springer, 398–413. https://doi.org/10.1007/3-540-58184-7_118

[57] The Rust Project Developers. [n.d.]. *Procedural Macros*. https://doc.rust-lang.org/reference/procedural-macros.html

[58] The Rust Project Developers. [n.d.]. *The MIR (Mid-level IR)*. https://rustc-dev-guide.rust-lang.org/mir/index.html

[59] Tokio Contributors. [n.d.]. Tokio. https://github.com/tokio-rs/tokio

[60] Allen Welkie and Elliott Mahler. [n.d.]. *RustFFT*. https://github.com/ejmahler/RustFFT

[61] Nobuko Yoshida and Lorenzo Gheri. 2020. A Very Gentle Introduction to Multiparty Session Types. In *Distributed Computing and Internet Technology (LNCS, Vol. 11969)*. Springer, 73–93. https://doi.org/10.1007/978-3-030-36987-3_5

[62] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *Trustworthy Global Computing (LNCS, Vol. 8358)*. Springer, 22–41. https://doi.org/10.1007/978-3-319-05119-2_3

[63] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically Verified Refinements for Multiparty Protocols. *Proceedings of the ACM on Programming Languages* 4 (2020), 1–30. Issue OOPSLA. https://doi.org/10.1145/3428216