Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types

Nicolas Lagaillardie
Department of Computing, Imperial College London, London, SW7 2AZ, United Kingdom

Rumyana Neykova
Department of Computer Science, Brunel University London, London, UB8 3PH, United Kingdom

Nobuko Yoshida
Department of Computing, Imperial College London, London, SW7 2AZ, United Kingdom

Abstract

Communicating systems comprise diverse software components across networks. To ensure their robustness, modern programming languages such as Rust provide both strongly typed channels, whose usage is guaranteed to be *affine* (*at most once*), and *cancellation* operations over *binary* channels. For coordinating components to correctly communicate and synchronize with each other, we use the structuring mechanism from *multiparty session types*, extending it with affine communication channels and implicit/explicit cancellation mechanisms. This new typing discipline, *affine multiparty session types* (AMPST), ensures *cancellation termination* of multiple, independently running components and guarantees that communication will not get stuck due to error or abrupt termination. Guided by AMPST, we implemented an automated generation tool (`MultiCrusty`) of Rust APIs associated with cancellation termination algorithms, by which the Rust compiler auto-detects unsafe programs. Our evaluation shows that `MultiCrusty` provides an efficient mechanism for communication, synchronization and propagation of the notifications of cancellation for arbitrary processes. We have implemented several usecases, including popular application protocols (OAuth, SMTP), and protocols with exception handling patterns (circuit breaker, distributed logging).

2012 ACM Subject Classification Software and its engineering → Software usability; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi

Keywords and phrases Rust language, affine multiparty session types, failures, cancellation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.1

Funding 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 and EP/V000462/1, and NCSS/EPSRC VeTSS.

Acknowledgements We thank the ECOOP reviewers for their insightful comments and suggestions, and (alphabetical order) Zak Cutner, Wen Kokke, Roland Kuhn, Dimitris Mostrous and Martin Vassor for discussions.

1 Introduction

The advantage of message-passing concurrency is well-understood: it allows cheap horizontal scalability at a time when technology providers have to adapt and scale their tools and applications to various devices and platforms. In recent years, the software industry has seen a shift towards languages with native message-passing primitives (e.g., Go, Elixir and Rust). Rust, for example, has been named the most loved programming language in the annual Stack Overflow survey for five consecutive years (2016-20) [47]. It has been used for the implementation of large-scale concurrent applications such as the Firefox browser, and Rust libraries are part of the Windows Runtime and Linux kernel. Rust's rise in popularity is due to its efficiency and memory safety. Rust's dedication to safety, however, does not *yet* extend to communication safety. Message-passing based software is as liable to errors as

ECOOP 2022.
Editors: Editors; Article No. 1; pp. 1:1–1:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

other concurrent programming techniques [49] and communication programming with Rust built-in message-passing abstractions can lead to a plethora of communication errors [27].

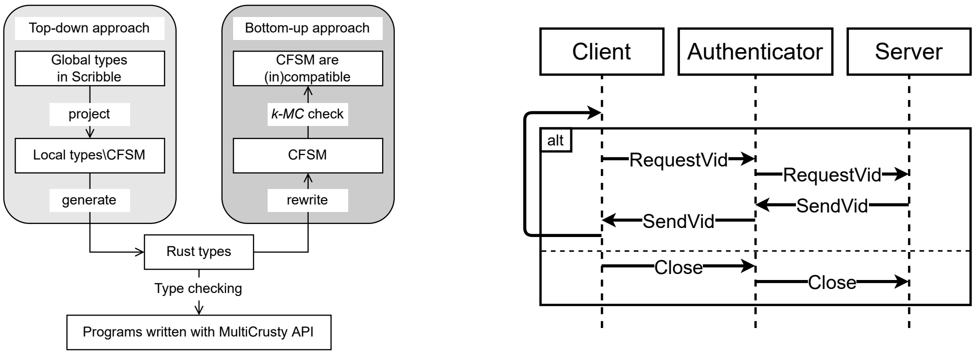
Much academic research has been done to develop rigorous theoretical frameworks for the verification of message-passing programs. One such framework is *multiparty session types* (MPST) [18] – a type-based discipline that ensures concurrent and distributed systems are *safe by design*. It guarantees that processes following a predefined communication protocol (also called a *multiparty session*) are free from communication errors and deadlocks. Rust may seem a particularly appealing language for the practical embedding of session types with its message-passing abstractions and affine type system. The core theory of session types, however, has serious shortcomings as its safety is guaranteed under the assumption that a session should run until its completion without any failure. Adapting MPST in the presence of failure and realising it in Rust are closely intertwined, and raise two major challenges:

Challenge 1: Affine multiparty session types (AMPST). There is an inherent conflict between the affinity of Rust and the linearity of session types. The type system of MPST guarantees a *linear* usage of channels, i.e., communication channels in a session must be used *exactly once*. As noted in [27], in a distributed system, it is a common behaviour that a channel or the whole session can be cancelled *prematurely* – for example, a node can suddenly get disconnected, and the channels associated with that node will be dropped. A naive implementation of MPST cancellation, however, will lead to incorrect error notification propagation, orphan messages, and stuck processes. The current theory of MPST does not capture affinity, hence cannot guarantee deadlock-freedom and liveness between multiple components in a realistic distributed system. Classic multiparty session type systems [18] do not prevent any errors related to session cancellation. An affine multiparty session type system should (1) prevent infinitely cascading errors, and (2) ensure deadlock-freedom and liveness in the presence of session cancellations for arbitrary processes. Although there are a few works on affine session types, they are either binary [36, 13] or modelling a very limited cancellation over a single communication action, and a general cancellation is not supported [16] (see § 6.2, and [30]).

Challenge 2: Realising an affine multiparty channel over binary channels. The extension of binary session types to multiparty is usually not trivial. The theory assumes multiparty channels, while channels, in practice, are binary. To preserve the global order specified by a global protocol, also called the order of interactions, when implementing a multiparty protocol over binary channels, existing works [19, 37, 42, 6] use code generation from a global protocol to local APIs, requiring type-casts *at runtime* on the underlying channels, compromising the type safety of the host type system. Implementing MPST with failure becomes especially challenging given that cancellation messages should be correctly propagated across multiple binary channels.

In this work, we overcome the above two challenges by presenting a new affine multiparty session types framework for Rust (AMPST). We present a shallow embedding of the theory into Rust by developing a library for safe communication, `MultiCrusty`. The library utilises a new communication data structure, *affine meshed channels*, which stores multiple binary channels without compromising type safety. A macro operation for exception handling safely propagates failure across all in-scope channels. We leverage an existing binary session types library, Rust's macros facilities, and optional types to ensure that communication programs written with `MultiCrusty` are *correct-by-construction*.

Our implementation brings three insightful contributions: (1) multiparty communication safety can be realised by the native Rust type system (without external validation tools); (2) top-down and bottom-up approaches can co-exist; (3) Rust's destructor mechanism can be



(a) `MultiCrusty` Workflow (top-down) **(b)** Video streaming service usecase

Figure 1 Programming with multiparty session types

utilised to propagate session cancellation. All other works generate not only the types but also the communication primitives for multiparty channels which are protocol-specific. The crucial idea underpinning the novelty of our implementation is that one can pre-generate the possible communication actions without having the global protocol; and then use the types to limit the set of permitted actions. Without this realisation neither (1), nor (2) is possible.

Paper Summary and Contributions:

§ 2 outlines the gains of programming with *affine meshed channels* by introducing our running example, a Video streaming service, and its Rust implementation using `MultiCrusty`.

§ 3 establishes the metatheory of AMPST. We present a core multiparty session π-calculus with session delegation and recursion, together with new constructs for exception handling, and affine selection and branching (from Rust optional types). The calculus enjoys session-fidelity (Theorem 3.14), deadlock-freedom (Theorem 3.16), liveness (Theorem 3.17), and a novel cancellation termination property (Theorem 3.22).

§ 4 describes the main challenges of embedding AMPST in Rust, and the design and implementation of `MultiCrusty`, a library for safe multiparty communication Rust programming.

§ 5 evaluates the execution and compilation overhead of `MultiCrusty`. Microbenchmarks show negligible overhead when compared with the built-in unsafe Rust channels, provided by `crossbeam-channel`, and up to two-fold runtime improvement to a binary session types library on protocols with high-degree of synchronisation. We have implemented, using `MultiCrusty`, examples from the literature, and application protocols (see [30]).

Additionally, § 6 discusses related works and § 7 concludes. The proofs of our theorems are included in [30]. Our library is available in this public library: `https://github.com/NicolasLagaillardie/mpst_rust_github/`. An ECOOP artifact is also available.

2 Overview: affine multiparty session types (AMPST) in Rust

Framework overview: AMPST in Rust Figure 1a depicts the overall design of `MultiCrusty`. Our design combines the top-down [18] and bottom-up [31] methodologies of multiparty session types in a single framework. Our bottom-up approach is discussed in details in [30].

The top-down approach enables *correctness-by-construction* and requires that a developer specifies a global type (hereafter a global protocol) describing the communication behaviour of the program. We utilise the Scribble toolchain [45] for writing and verifying global protocols. The toolchain projects local types for each role in a protocol. We have augmented the toolchain to further generate those local types into Rust types, i.e., types that stipulate the behaviour of communication channels.

Our Rust API (`MultiCrusty` API) integrates both approaches, as illustrated in Figure 1a. Developers can choose to either (1) write the global protocol and have the Rust types

```
// generates at compile-time communication primitives for 3-party affine meshed channels
gen_mpst!(MeshedChannelsThree, A, C, S);
```

```
fn client(
  s: RecC<i32>,
  i: i32
  ) -> R {
  if (i<MAX) {
  let s = choose_c!(s,
  ChoiceA::Video, ChoiceS::
     Video)
  let n = get_video(i);
  let s = s.send(n)?;
  let (_,s) = s.recv()?;
  client(s, i+1)
  } else {
  let s = choose_c!(s,
  ChoiceA::Close, ChoiceS::
     Close);
  s.close()
  }
}
```

```
fn auth(s: RecA<i32>)
  -> R {
  offer_mpst!(
  s, {
  ChoiceA::Video(s)
  => {
  let (x,s) = s.recv()?;
  let s = s.send(x)?;
  let (x,s) = s.recv()?;
  let s = s.send(x)?;
  auth(s)
  },
  ChoiceA::Close(s)
  => {
  s.close()
  } }
  )
}
```

```
fn server(s: RecS<i32>)
  -> R {
  offer_mpst!(
  s, {
  ChoiceS::Video(s)
  => {
  let v = attempt!{{
    let (x, s) = s.recv()?;
    let f = get_file(x);
    read_video_file(f)
  } catch (e) {
    cancel(s);
    panic!("Err: {:?}", e)
  } }()?;
  let s = s.send(x)?;
  server(s)}
  ChoiceS::Close(s)
  => {s.close()
} } ) }
```

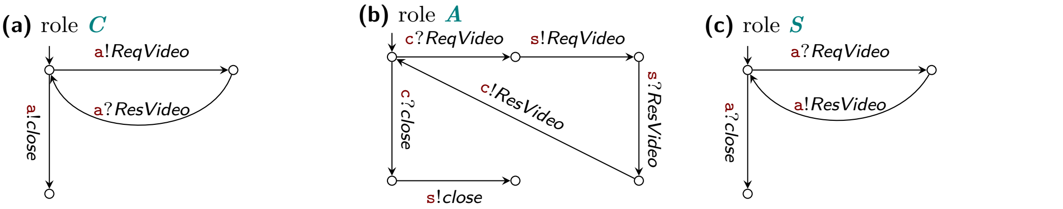

(a) role C (b) role A (c) role S

Figure 2 Rust implementations and respective CFSMs of role C (a), role A (b) and role S (c)

generated, or (2) write the Rust types manually and check that the types are compatible. Note that both approaches rely on concurrent programs written with `MultiCrusty` API, and both approaches rely on the Rust compiler to type check the concurrent programs against their respective types. Overall, the framework guarantees that well-typed concurrent programs implemented using `MultiCrusty` API with Scribble-generated types or k-MC-compatible types, will be free from deadlocks, reception errors, and protocol deviations.

The main primitives of `MultiCrusty` API are summarised in Table 1. Next, we briefly explain them through an example. A more detailed explanation is provided in § 4.

2.1 Example: Video streaming service

The *Video streaming service* is a usecase that can take full advantage of affine multiparty session types and demonstrate the need for multiparty channels with cancellation. Each streaming application connects to servers, and possibly other devices, to access services and follows a specific protocol. To present our design, we use a simplified version of the protocol, omitting the authentication part, illustrated in the diagram of Figure 1b. The diagram should be read from top to bottom. The protocol involves three services – an *Authenticator* (role A) service, a *Server* (role S) and a *Client* (role C). The protocol starts with a choice on the *Client* to either request a video or end the session. The first branch is, *a priori*, the main service provided, i.e., request for a video. The *Client* cannot directly request videos from the *Server* and has to go through the *Authenticator* instead. On the diagram, the choice is denoted as the frame `alt` and the choices are separated by the horizontal dotted line. The protocol is recursive, and the *Client* can request new videos as many times as needed. This recursive behaviour is represented by the arrow going back on the *Client* side in Figure 1b. To end the session, the *Client* first sends a `Close` message to the *Authenticator*, which then subsequently sends a `Close` message to the *Server*.

Affine meshed channels and multiparty session programming with `MultiCrusty` The implementations in `MultiCrusty` of the three roles are given in Figure 2. They closely follow the behaviour that is prescribed by the protocol. The global protocol does not explicitly specify cancellation. However, in a distributed setting, timeout or failure can happen at any time: a request from a CDN network or cloud storage to the server might be a timeout or the result message might be lost. Our implementation accounts for failure by providing communication primitives for two different types of session cancellation, called *implicit* and *explicit*: either we run a block of code and upon any error at any point, we go to the **catch** branch, or we explicitly test each step.

The implementation of the three concurrent programs starts by generating all communication primitives for affine channels between three roles. This is done by the macro `gen_mpst!(MeshedChannelsThree, A, C, S)`, see line 2 in Figure 2. The macro `gen_mpst!` takes two kinds of arguments: the name of the data structure for affine meshed channels, `MeshedChannelsThree`, and the name for each role, `A`, `C` and `S`. `MeshedChannelsThree` is a string literal that must be supplied by the developer, any name can be chosen. In our case, `gen_mpst!` will generate a data structure called `MeshedChannelsThree` that can be used for communication between three participants. In our example, three roles are provided, but the macro can handle any number of roles. Then, using the Rust procedural macro system, it generates communication primitives for programming between affine meshed channels. This generation is done at compile time. For instance, the primitive `s.send(p)` sends a payload `p` on an affine meshed channel `s`. Note that we do not have to explicitly specify the destination channel, this is determined from the type: the stack specifies which binary channel can be used, regardless of the type of those binary channels. See Figure 6 for an example of `MeshedChannels`.

To explain affine meshed channels and all `MultiCrusty` communication primitives, we focus on the implementation for role $\boldsymbol{A}$ given in Figure 2b. The implementations of the other roles are similar. First, line 1 declares an `auth(s)` function that is parametric on an affine meshed channel `s` of type `RecA<i32>`, the result type of the function is irrelevant to our explanation, hence we have simply denoted it by `R`. The type `RecA<i32>`, an alias for the full type described in Figure 6, specifies the operations allowed on `s`. As mentioned previously, this type can be either written by the developer or generated by Scribble. We defer the explanation of the (generated) types to § 4, i.e., the full Rust type is given in Figure 6. For clarity, here we only give a high-level view of the behaviour for each channel by representing its respective local session types as a communicating finite state machine (CFSM [4]), where ! (resp. ?) denotes sending (resp. receiving). The CFSMs for each role (channel) can be seen in Figure 2. For example, `c!ResVideo` means that role $\boldsymbol{A}$ is receiving from the role $\boldsymbol{C}$ a message labelled as `Video`, while `s!ReqVideo` says that role $\boldsymbol{A}$ sends a message to role $\boldsymbol{S}$.

The thread for role $\boldsymbol{A}$ uses an affine meshed channel `s` to implement the given CFSM behaviour. In essence, the meshed channel is implemented as an indexed tuple of binary channels – one binary channel for each pair of interacting processes, i.e., a binary channel for role $\boldsymbol{A}$ and role $\boldsymbol{S}$ and a binary channel for role $\boldsymbol{A}$ and role $\boldsymbol{C}$.

The implementation starts by realising a choice: role $\boldsymbol{C}$ broadcasts its choice, which can either be to request a video at line 7 or to close the connection at line 14 (Figure 2a). Role $\boldsymbol{C}$ broadcasts the choice to every other role. This choice is received by role $\boldsymbol{A}$, which will either receive a `Video` or a `Close` label. This behaviour is implemented by the `MultiCrusty` macro `offer_mpst!` (line 3), which is applied to a multiparty channel `s` and a sum type, either `ChoiceA::Video` or `ChoiceA::Close` here. The behaviour of each branch in the protocol is implemented as an anonymous function. For example, the code in lines 5 – 12 supplies such a function that implements the behaviour when role $\boldsymbol{C}$ sends a `Video` label, while lines 13 – 16 handle the `Close` request. At each step, the channel `s` is rebound to a new meshed channel

Table 1 Primitives provided by `MultiCrusty`. `s` is an affine meshed channel; `p` is a payload of a given type; I is a subset of all roles in the protocol but the current role; K is a subset of all branches;

Primitives	Description
`let s = s.send(p)?;`	Sends a payload `p` on a channel `s` and assigns the continuation of the session (a new meshed channel) to a new variable `s`.
`let (p, s) = s.recv()?;`	Receives a payload `p` on channel `s` and assigns the continuation of the session to a new variable `s`.
`s.close()`	Closes the channel `s` and returns a unit type.
`attempt! {{ ... } catch(e) { ... }}`	Attempts to run the first block of code and, upon error, catches the error in the variable `e` and runs the second block of code.
`offer_mpst!(s, {` $\mathtt{enum_i} :: \mathtt{variant_k}(\mathtt{e}) => \{...\}_{k\in K}$ `} )`	Role $\boldsymbol{i}$ receives a message label on channel `s`, and, depending on the label value which should match one of the variants $\mathtt{variant_k}$ of $\mathtt{enum_i}$, runs the related block of code.
`choose!(s,` $\{\mathtt{enum_i} :: \mathtt{variant_k}\}_{i\in I}$ `)`	Sends the chosen label, which corresponds to $\mathtt{variant}_k$, to all other roles.

`s` returned by the respective communication primitive. For example, the communication primitive `recv()` at line 7 is for receiving a value on the binary channel between role $\boldsymbol{A}$ and role $\boldsymbol{C}$ that is stored in the meshed channel `s`. Note that this is the only primitive available for that type at that point. This communication primitive, if everything goes right, returns a tuple containing the received value and the new meshed channel to be used for subsequent communications which are bound resp. to the variable `x` and `s`. An error is returned instead of the tuple if the reception of the value fails. Similarly, `send(x)` in line 8 sends the value `x` to the process that implements the role $\boldsymbol{S}$ and rebinds the new meshed channel to `s`. Finally, because the protocol is recursive, line 11 calls the recursive function `auth(s)`.

Alternatively, the anonymous function for branch `Close` calls the primitive `close()` which safely and cleanly closes all binary channels stored inside the affine meshed channel `s`. This last primitive ensures that the type of all the binary channels of `s` is `Close End`: the only primitive implemented for such meshed channels is `close()`. Forgetting either `s.close()` or `auth(s)` at the end of their respective branch will throw an error during compilation because the output type of the `auth(s)` function will be the wrong one. All communication functions used in the example (i.e., `s.recv()`, `s.send()`, `s.close()`, `offer_mpst!`) are generated for all roles and all possible interactions through the macro `gen_mpst!`, see line 2.

The types of the affine meshed channels, as well as the generic types in the declaration of the `MultiCrusty` communication functions, enable compile-time detection of protocol violations. Examples of protocol violations include swapping lines 10 and 9, using another communication primitive or using the wrong payload type. The Rust type system, on the other hand, ensures that all affine channels are used at most once. For example, using channel `s` twice (without rebinding) will be detected by the compiler. All the errors mentioned above will be reported as compile-time errors. In the case that an unexpected runtime error occurs, all roles are guaranteed to terminate safely. This is ensured by two mechanisms – explicit session cancellation (that can be triggered by the user) and implicit session cancellation (that is embedded in the `MultiCrusty` primitives and the channel destructors).

Implicit and explicit cancellations The processes of role $\boldsymbol{A}$ and role $\boldsymbol{S}$ illustrate resp. implicit and explicit cancellations. The primitive `cancel(s)` drops the affine meshed channel `s`, and its binary channels, making it inaccessible to other participants. This is convenient when an error related to the computation aspects of the program occurs. For example, In Figure 2c the session is cancelled in line 12 after an error occurs as a result of reading a corrupted video. We have used the `attempt!{ { ... } catch(e) { ... } }` macro (Rust version of a `try-catch` block) in lines 7 to 14 to catch the error message, and explicitly cancel the session. The macro tries to go through the `attempt`-block of code, and upon any error in this block, stops the process and calls the `catch(e)`-block with the error message `e`. Line 13 executes a `panic!`, which allows a program to terminate immediately and provide feedback to the caller of the program. Forgetting to call `cancel(s)` before `panic!` will result in the same outcome as when both `cancel(s)` and `panic!` drop `s`. Forgetting both will throw an error because the output type will not match the one of `fn server(s)`, unless replaced with an `Ok(())`. In any cases, an error will be thrown on other threads linked to other roles because role $\boldsymbol{S}$'s sessions are inaccessible in the `catch(e)`-block.

Alternatively, we explain implicit cancellation as implemented by role $\boldsymbol{A}$ in Figure 2b. The construct `let x = f()?`, as seen in line 7, is Rust's *monadic bind* notation for programs and functions that may return errors: their usual output type is `Result<T, Error>` where `T` is the expected type if everything goes right and `Error` is the error type returned. For any program and function returning such type, the users have to *unwrap* it. The two usual ways of doing so are by using the `?`-operator, or by pattern matching on the result using `match`. In our case, if `recv()` succeeds, the `?`-operator unpacks the result and returns the tuple containing the received payload and the continuation. If `recv()` fails, the `?`-operator short-circuits, skips the rest of the statements, and returns the error. We use this mechanism to catch any session cancellation. In the case that a `recv()` (or `send()`) does not succeed, the implementation of the underlying communication primitive will cancel the channel and broadcast the cancellation to all other binary channels that are part of the session.

Finally, we look at the implementation of role $\boldsymbol{C}$ to demonstrate the final mechanism of session cancellation. For this purpose, we have to comment out lines 9 – 11 in Figure 2a or replace them with a `panic!` as to simulate a wrongly implemented role $\boldsymbol{C}$. With such modification, this function will still compile despite the protocol not being fully implemented (since the last received action from role $\boldsymbol{A}$ is missing, the meshed channel `s` will be dropped prematurely). Even in this case, `MultiCrusty` ensures that all processes will terminate safely, i.e., all parties are notified that an affine channel has been dropped. Prematurely dropping a channel can happen due to incorrectly implemented behaviour (as we demonstrated above), or by unhandled user error, for example, the function `get_video()` in line 8 can invoke a `panic!` because there is no video associated with the index `i`. Safe session termination is realised by customising the native destructor `Drop` in Rust, as proposed for binary meshed channels by [27]. When an affine meshed channel goes out of scope, the channel destructor is called, the session is cancelled, dropping every channel value used in the session, and only then is the memory deallocated.

Memory management should not pose a problem in our case. Our library uses only the safe fragment of Rust. This ensures that variables that are out of scope are automatically collected. We utilise this mechanism to ensure that closed and cancelled channels are collected in the same way. Hence, memory leaks are ruled out.

In short, a session can be cancelled for three reasons: (1) an error affecting the computation aspects of the program, as in Figure 2c; (2) an error during communication, e.g., a timeout on a channel, as in Figure 2b; or (3) a premature drop of the affine meshed channel due to incorrect implementation, as in Figure 2a. Our mechanisms for session cancellation cover all the above cases. In this way, our framework provides *affine multiparty session compliance* by ensuring that (1) if all results are returned without failure, the processes follow the given Scribble global protocol (Theorem 3.14) or (2) once a cancellation happens, all processes in the same session terminate with an error (Theorem 3.22). We have proven the above results by formalising affine meshed channels in an extension of a multiparty π-calculus.

3 Affine multiparty session processes for Rust programming

3.1 Affine multiparty session processes

Our calculus (AMPST) is an extension of a full multiparty session π-calculus [44] which includes session delegation (channel passing) and session recursion. We shade additions to [44] in this colour .

▶ **Definition 3.1.** The **affine multiparty session π-calculus** (AMPST) is defined as follows:

$$
\begin{array}{rcll}
c, d & ::= & x \mid s[\mathtt{p}] \qquad \dagger ::= \emptyset \mid \mathbf{?} & \text{(variable, channel with role } \mathtt{p}\text{, error, flag)}\\
P, Q & ::= & \mathbf{0} \mid P \mid Q \mid (\nu s)\, P & \text{(inaction, composition, restriction)}\\
 & & \mathbf{?}\, c[\mathtt{q}] \oplus \mathtt{m}\langle d\rangle . P \mid \mathbf{?}\, c[\mathtt{q}] \sum_{i\in I} \mathtt{m}_i(x_i) . P_i & \text{(affine selection, branching } I \neq \emptyset\text{)}\\
 & & c[\mathtt{q}] \oplus \mathtt{m}\langle d\rangle . P \mid c[\mathtt{q}] \sum_{i\in I} \mathtt{m}_i(x_i) . P_i & \text{(selection, branching } I \neq \emptyset\text{)}\\
 & & \mathbf{def}\ D\ \mathbf{in}\ P \mid X\langle \widetilde{c}\rangle & \text{(process definition, process call)}\\
 & & \mathbf{try}\ P\ \mathbf{catch}\ Q \mid \mathbf{cancel}(c).P \mid s\lightning & \text{(catch, cancel, kill)}\\
D & ::= & X(\widetilde{x}) = P & \text{(declaration of process variable } X\text{)}
\end{array}
$$

A set $\mathscr{P}$ denotes **participants**: $\mathscr{P} = \{\mathtt{p}, \mathtt{q}, \mathtt{r}, \dots\}$, and $\mathbb{A}$ is a set of alphabets. A **channel** c can be either a variable or a **channel with role** $s[\mathtt{p}]$, i.e., a multiparty communication endpoint whose user plays role $\mathtt{p}$ in the session s. $\widetilde{c}$ denotes a vector $c_1 c_2 \dots c_n$ $(n \geq 1)$ and similarly for $\widetilde{x}$ and $\widetilde{s}$.

The two processes with $\mathbf{?}$ model the option `?`-operator in Rust. Process $\mathbf{?}\, c[\mathtt{q}] \oplus \mathtt{m}\langle d\rangle . P$ performs an **affine selection (internal choice)** towards role $\mathtt{q}$, using the channel c: if the *message label* $\mathtt{m}$ with the *payload* channel d is successfully sent, then the execution continues as P; otherwise (if the receiver has failed or timeout), it triggers an exception. The **affine branching (external choice)** $\mathbf{?}\, c[\mathtt{q}] \sum_{i\in I} \mathtt{m}_i(x_i) . P_i$ uses channel c to wait for a message from role $\mathtt{q}$: if a message label $\mathtt{m}_k$ with payload d is received (for some $k \in I$), then the execution continues as P_k, with x_k replaced by d; if not received, it triggers an exception. Note that message labels $\mathtt{m}_i$ are pairwise distinct and their order is irrelevant, and variable x_i is bound with scope P_i.

The following two failure handling processes follow the program behaviour of Figure 2c. The **try-catch** process, **try** P **catch** Q, consists of a *try process* P which is ready to communicate with parallel composed one; and a *catch process* Q which becomes active when a cancellation or an error happens. The **cancel** process, $\mathbf{cancel}(c).P$, cancels other processes whose communication channel is c. The **kill** $s\lightning$ kills all processes with session s and **is generated only at runtime** from affine or cancel processes.

The other syntax is from [44]. The **inaction 0** represents a terminated process (and is often omitted). The **parallel composition** $P \mid Q$ represents two processes that can execute concurrently, and potentially communicate. The **session restriction** $(\nu s)\, P$ declares a new session s with a scope limited to process P. The linear **selection** and the linear **branching** can be understood as their affine versions but without failure handling. **Process definition**, **def** $X(\widetilde{x}) = P$ **in** Q and **process call** $X\langle\widetilde{c}\rangle$ model recursion: the call invokes X by expanding it into P, and replacing its formal parameters with the actual ones.

Linear or affine branching and selection are denoted as either $\dagger\, c[\mathtt{q}] \sum_{i\in I} \mathtt{m}_i(x_i) . P_i$ and $\dagger\, c[\mathtt{q}] \oplus \mathtt{m}\langle d\rangle . P$. We use $\mathrm{fv}(P) \,/\, \mathrm{fc}(P)$ and $\mathrm{dpv}(P) \,/\, \mathrm{fpv}(P)$ to denote *free variables/channels* and *bound/free process variables* of P. We call a process P such that $\mathrm{fv}(P) = \mathrm{fpv}(P) = \emptyset$ *closed*. A *set of subjects of* P, written $\mathrm{sbj}(P)$, is defined as: $\mathrm{sbj}(\mathbf{0}) = \emptyset$; $\mathrm{sbj}(P \mid Q) = \mathrm{sbj}(P) \cup \mathrm{sbj}(Q)$; $\mathrm{sbj}((\nu s)\, P) = \mathrm{sbj}(P) \setminus \{s[\mathtt{p}_i]\}_{i\in I}$; $\mathrm{sbj}(\dagger\, c[\mathtt{q}] \sum_{i\in I} \mathtt{m}_i(x_i) . P_i) = \mathrm{sbj}(\dagger\, c[\mathtt{q}] \oplus \mathtt{m}\langle d\rangle . P) = \{c\}$; $\mathrm{sbj}(\mathbf{def}\ X(\widetilde{x}) = P\ \mathbf{in}\ Q) = \mathrm{sbj}(Q) \cup \mathrm{sbj}(P) \setminus \{\widetilde{x}\}$ with $\mathrm{sbj}(X\langle\widetilde{c}\rangle) = \mathrm{sbj}(P\{\widetilde{c}/\widetilde{x}\})$; $\mathrm{sbj}(\mathbf{try}\ P\ \mathbf{catch}\ Q) = \mathrm{sbj}(P)$; and $\mathrm{sbj}(\mathbf{cancel}(c).P) = \{c\}$.

The set of subjects is the key definition which enables us to define the typing system for the `try-catch` process with recursive behaviours.

$$
\begin{array}{ll}
\text{[R-Com]} & \mathbb{E}_1[\dagger\, s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i] \mid \mathbb{E}_2[\dagger\, s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{m}_k\langle s'[\mathtt{r}]\rangle.Q] \;\rightarrow\; P_k\left\{{}^{s'[\mathtt{r}]}/_{x_k}\right\} \mid Q \quad \text{if } k\in I \\
\text{[C-?Sel]} & ?\, s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P \rightarrow s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P \mid s\lightning \\
\text{[T?Sel]} & \textbf{try } ?\, s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P \textbf{ catch } Q \rightarrow Q \mid s\lightning \\
\text{[C-Sel]} & s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P \mid s\lightning \rightarrow P \mid s\lightning \mid s'\lightning \\
\text{[C-?Br]} & ?\, s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \rightarrow s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s\lightning \\
\text{[T?Br]} & \textbf{try } ?\, s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \textbf{ catch } Q \rightarrow Q \mid s\lightning \\
\text{[C-Br]} & s[\mathtt{p}][\mathtt{q}]\textstyle\sum_{i\in I}\mathtt{m}_i(x_i).P_i \mid s\lightning \rightarrow (\nu s')\,(P_k\left\{{}^{s'[\mathtt{r}]}/_{x_k}\right\} \mid s'\lightning) \mid s\lightning \quad s'\notin \mathrm{fc}(P_k)\,, k\in I \\
\text{[R-Can]} & \mathbb{E}[\textbf{cancel}(s[\mathtt{p}]).Q] \rightarrow s\lightning \mid Q \qquad \text{[C-Cat]}\;\; \textbf{try } P \textbf{ catch } Q \mid s\lightning \rightarrow Q \mid s\lightning \quad \exists\mathtt{r}.\; s[\mathtt{r}] = \mathrm{sbj}(P) \\
\text{[R-Def]} & \textbf{def } X(x_1,\ldots,x_n) = P \textbf{ in } (X\langle s_1[\mathtt{p}_1],\ldots,s_n[\mathtt{p}_n]\rangle \mid Q) \\
 & \qquad \rightarrow \textbf{def } X(x_1,..,x_n) = P \textbf{ in } (P\{{}^{s_1[\mathtt{p}_1]}/_{x_1}\}\cdot\cdot\{{}^{s_n[\mathtt{p}_n]}/_{x_n}\} \mid Q) \\
\text{[R-Ctx]} & P\rightarrow P' \text{ implies } \mathbb{C}[P]\rightarrow\mathbb{C}[P'] \qquad \text{[R-Struct]}\;\; P\equiv P'\rightarrow Q'\equiv Q \text{ implies } P\rightarrow Q
\end{array}
$$

Figure 3 AMPST π-calculus reduction between closed processes (we highlight the new rules from [44])

▶ **Example 3.2** (Subjects of processes). Assume $R_1 = \textbf{def } X(x) = x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0} \textbf{ in } X\langle c\rangle$ which repeats the action at c and emits a message d with label repeatedly interacting with the dual input (but reduction with this process only happens if there is a corresponding input at c, i.e., on-demand). We calculate $\mathrm{sbj}(R_1)$ as:

$$
\begin{aligned}
&\mathrm{sbj}(\textbf{def } X(x) = x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0} \textbf{ in } X\langle c\rangle) = \mathrm{sbj}(X\langle c\rangle)\cup\mathrm{sbj}(\textbf{def } X\langle x\rangle = x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0})\\
&= \mathrm{sbj}(X\langle c\rangle) = \mathrm{sbj}((x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0})\{{}^{c}/_{x}\}) = \{c\}
\end{aligned}
$$

Another example is: $\mathrm{sbj}(\textbf{try } x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0} \textbf{ catch } \textbf{cancel}(x[\mathtt{q}]).\mathbf{0}) = \mathrm{sbj}(x[\mathtt{q}]\oplus\mathtt{m}\langle d\rangle.\mathbf{0}) = \{x\}$.

▶ Remark 3.3 (Syntax and semantics). AMPST extends MPST incorporating some design choices from [36], aiming to distil the implementation essence of `MultiCrusty`. The design of our `try-catch` process follows the binary affine session types in [36], but models more cancellations for arbitrary processes with affine branchings/selections and cancel processes non-deterministically (whose semantics follow the implementation behaviours, see § 4.4). We list the essential differences from [36]. (1) (Nondeterministic failures) The kill process is a runtime syntax and generated only during reductions unlike [36]. Our calculus also allows *nondeterministic failures* caused by either (1) affine selection/branching or (2) `try-catch` processes. See [30] for examples. (2) (Recursion parameterised by linear names) One of the novelties of our formalism which is not found in [36] is a combination of session recursions, affinity, and interleaved sessions, i.e., the **def** agents (linearly parameterised recursions), which are the most technical part when designing the typing system with `try-catch` processes. The combination of all features is absent from [36, 13, 16]: see § 6 for more detailed comparisons.

▶ **Definition 3.4** (Semantics). *A **try-catch context** $\mathbb{E}$ is: $\mathbb{E} ::= \textbf{\textit{try}}\ \mathbb{E}\ \textbf{\textit{catch}}\ P \;\big|\; [\,]$ and a **reduction context** $\mathbb{C}$ is: $\mathbb{C} ::= (\nu s)\mathbb{C} \;\big|\; \textbf{\textit{def}}\ D\ \textbf{\textit{in}}\ \mathbb{C} \;\big|\; \mathbb{C}\mid P \;\big|\; P\mid\mathbb{C} \;\big|\; [\,]$. **Reduction** $\rightarrow$ is inductively defined in Figure 3, which uses the **structural congruence** $\equiv$ which is defined by $s\lightning \mid s\lightning \;\equiv\; s\lightning$ and $(\nu s)\, s\lightning \;\equiv\; \mathbf{0}$ together with other rules in [44].*

▶ Remark 3.5 (Nested try-catches and $\mathbb{E}$). The context $\mathbb{E}$ is only used for defining the reductions at the top parallel composed processes, *not* used nested exception handling like [13, 16]. Our (typable) try-catch processes allow any form of processes such as recursions, parallel, session delegations, and restriction/scope opened processes under a guarded process:

$$R = \textbf{try}\ \ s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{m}_1.(\nu s')\,(s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{m}_3\langle s'[\mathtt{r}]\rangle.\mathbf{0} \mid \textbf{try}\ \ s'[\mathtt{q}][\mathtt{r}]\oplus\mathtt{m}_2.\mathbf{0}\ \ \textbf{catch}\ \ \textbf{cancel}(s'[\mathtt{q}]).\mathbf{0}) \\ \textbf{catch}\ \ \textbf{cancel}(s[\mathtt{p}]).\mathbf{0}$$

See [30] for more typable processes with nested **try-catch** blocks.

We explain each rule highlighting the new rules.

Communication Rule [R-Com] is the main communication rule between an affine/linear selection and an affine/linear branching. Linear selections/branching are placed in the *try* position but can interact with affine counterparts. Once they interact, processes are spawned from *try-block*s (notice that $\mathbb{E}_1$, $\mathbb{E}_2$ are erased after the communication), and start communicating on parallel with other parallel composed processes. Note that the context $\mathbb{E}$ is discarded after the successful communication.

Error-Cancellation Rules [C-?Sel] and [C-?Br] model the situations that an error handling occurs at the affine selection/branching. This might be the case if its counterpart has failed (hence [R-Com] does not happen) or timeout. It then triggers the kill process at s. Rules [T?Sel] and [T?Br] model the case that the affine selection/branching are placed inside the try-block and triggered by the error. In this case, it will go to the *catch-block*, generating a kill process.

Cancelling Processes Rule [C-Sel] cancels the selection prefix s, additionally generating the kill process at the delegated channel for all the session processes at s' to be cancelled. Rule [C-Br] cancels only one of the branches – this is sufficient since all branches contain the same channels except x_i (ensured by rule [T-&] in Figure 4). After the cancellation, it additionally instantiates a fresh name $s'[\mathtt{r}]$ to x_k into P_k. The generated kill process at s' kills prefixes at $s'[\mathtt{r}]$ in $P_k\{^{s'[\mathtt{r}]}/_{x_k}\}$.

Cancellation from Other Parties Rule [R-Can] is a cancellation and generates a kill process. Note that the **try-catch** context $\mathbb{E}$ is thrown away. Rule [C-Cat] is prompted to move to Q by kill $s\lightning$. The side condition $\mathrm{sbj}(P)$ ensures that P is a prefix at s (up to $\equiv$ for a recursive process). All mimic the behaviour of the programs in Figure 2c.

Other Rules Rules [R-Def], [R-Ctx], and [R-Struct] are standard from [44]. In Figure 3, the two new rules are for garbage collections of kill processes.

▶ **Example 3.6** (Syntax and reductions). A process might be completed, or cancelled in many ways, and also interacts non-deterministically. We demonstrate the reduction rules using the running example with a minor modification. We use a nested **try-catch** block, and for simplicity we use shorter label names, and we use a constant, i.e., d, as a message payload.

Assume the process for role $\boldsymbol{S}$ is $P = \,?\,s[\mathtt{p}][\mathtt{q}](Q + \mathtt{close}(x).\mathbf{0})$ where

$Q = \mathtt{video}(x).\textbf{try}\ \ ?\,s[\mathtt{p}][\mathtt{q}]\mathtt{req}(x).\textbf{try}\ \ ?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}\ \ \textbf{catch}\ \ \textbf{cancel}(s[\mathtt{p}]).\mathbf{0}\ \ \textbf{catch}\ \ \textbf{cancel}(s[\mathtt{p}]).\mathbf{0}$

The following shows a possible reduction.

$$P \mid s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{video}\langle\mathtt{d}\rangle.s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{req}\langle\mathtt{d}\rangle.s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \tag{1}$$

$$\text{[R-Com]} \rightarrow \begin{array}{l}\textbf{try}\ \ (?\,s[\mathtt{p}][\mathtt{q}]\mathtt{req}(x).\textbf{try}\ \ (?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0})\ \ \textbf{catch}\ \ \textbf{cancel}(s[\mathtt{p}]).\mathbf{0}) \\ \quad \textbf{catch}\ \ \textbf{cancel}(s[\mathtt{p}]).\mathbf{0}\end{array} \tag{2}$$

$$\mid s[\mathtt{q}][\mathtt{p}]\oplus\mathtt{req}\langle\mathtt{d}\rangle.s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \tag{3}$$

$$\text{[R-Com]} \rightarrow \textbf{try}\ \ ?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}\ \ \textbf{catch}\ \ \textbf{cancel}(s[\mathtt{p}]).\mathbf{0} \mid s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \tag{4}$$

$$\text{[T?Sel]} \rightarrow \textbf{cancel}(s[\mathtt{p}]).\mathbf{0} \mid s\lightning \mid s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \tag{5}$$

$$\text{[R-Can]} \rightarrow s\lightning \mid s\lightning \mid \mathbf{0} \mid s[\mathtt{q}][\mathtt{p}]\mathtt{res}(x).\mathbf{0} \tag{6}$$

$$\text{[C-Br]} \rightarrow s\lightning \mid s\lightning \mid \mathbf{0} \mid \mathbf{0} \equiv s\lightning \tag{7}$$

$\mathbb{E}_6[P_6]$ for Equation (1) is $P_6 = \,?\,s[\mathtt{p}][\mathtt{q}]\mathtt{req}(x).\textbf{try}\ \ \ldots\ \ \textbf{catch}\ \ \textbf{cancel}(s[\mathtt{p}]).\mathbf{0}$ and $\mathbb{E}_9[P_9]$ for Equation (4) is $P_9 = \,?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}$ both because of rule [C-Cat].

Initially we reduce using the communication rule for the branching and selection. Next, we apply [R-Com] demonstrating how the affine branching reduces under **try**. Then we apply [T?Sel] assuming an error (or a timeout) occurs during the selection of `res`. This generates a kill process $s\lightning$ and spawns the process in the **catch**-block. **Cancel** spawns a kill process $s\lightning$ and hence reduces to $s\lightning \mid \mathbf{0}$, following rule [R-Can] with $\mathbb{E} = [\,]$. Finally, applying [R-Can] cancels the linear selection. To conclude, we garbage collect all kill processes. Given that our initial parallel composition has name restrictions (νs) at the top level, $(\nu s)\, s\lightning \equiv \mathbf{0}$.

3.2 Affine multiparty session typing system

Global and local types The advantage of affine session frameworks is that no change of the syntax of types from the original system is required. We follow [44] which is the most widely used syntax in the literature. A *global type*, written $G, G', \ldots$, describes the whole conversation scenario of a multiparty session as a type signature, and a *local type*, written by $S, S', \ldots$, represents a local protocol for each participant. The syntax of types is given as:

▶ **Definition 3.7** (Global types)**.** The syntax of a **global type** G is:

$$G \;::=\; \mathtt{p}\to\mathtt{q}\colon \{\mathtt{m_i}(S_i).G_i\}_{i\in I} \;\big|\; \mu\mathbf{t}.G \;\big|\; \mathbf{t} \;\big|\; \mathbf{end} \qquad \text{with } \mathtt{p}\neq\mathtt{q},\;\; I\neq\emptyset,\;\text{ and }\; \forall i\in I : \mathrm{fv}(S_i)=\emptyset$$

The syntax of **local types** is:

$$S, T \;::=\; \mathtt{p}\&_{i\in I}\mathtt{m}_i(S_i).S'_i \;\big|\; \mathtt{p}\oplus_{i\in I}\mathtt{m}_i(S_i).S'_i \;\big|\; \mathbf{end} \;\big|\; \mu\mathbf{t}.S \;\big|\; \mathbf{t} \quad \text{with } I\neq\emptyset\text{, and } \mathtt{m}_i \text{ pairwise distinct.}$$

Types must be closed, and recursion variables to be guarded.

$\mathtt{m} \in \mathbb{A}$ corresponds to the usual message labels in the session type theory. Global branching type $\mathtt{p}\to\mathtt{q}\colon \{\mathtt{m_i}(S_i).G_i\}_{i\in I}$ states that participant $\mathtt{p}$ can send a message with one of the $\mathtt{m}_i$ labels and a *message payload type* S_i to the participant $\mathtt{q}$ and that interaction described in G_i follows. We require $\mathtt{p} \neq \mathtt{q}$ to prevent self-sent messages and $\mathtt{m}_i \neq \mathtt{m}_k$ for all $i \neq k \in J$. Recursive types $\mu\mathbf{t}.G$ are for recursive protocols, assuming those type variables $(\mathbf{t}, \mathbf{t}', \ldots)$ are guarded in the standard way, i.e., they only occur under branching. Type $\mathbf{end}$ represents session termination (often omitted). We write $\mathtt{p} \in \mathrm{roles}(G)$ (or simply $\mathtt{p} \in G$) iff, for some $\mathtt{q}$, either $\mathtt{p}\to\mathtt{q}$ or $\mathtt{q}\to\mathtt{p}$ occurs in G. The function $\mathsf{id}(G)$ gives the participants of G.

For local types, the *branching type* $\mathtt{p}\&_{i\in I}\mathtt{m}_i(S_i).S'_i$ specifies the reception of a message from $\mathtt{p}$ with a label among the $\mathtt{m}_i$ and a payload S_i. The *selection type* $\mathtt{p}\oplus_{i\in I}\mathtt{m}_i(S_i).S'_i$ is its *dual* – its opposite operation. The remaining type constructors are as for global types. We say a type is *guarded* if it is neither a recursive type nor a type variable.

The relation between global and local types is formalised by projection [8, 18]. The *projection of G onto $\mathtt{p}$* is written $G{\upharpoonright}\mathtt{p}$ and the standard subtyping relation, $\leqslant$. See [30].

We define typing contexts which are used to define properties of type-level behaviours.

▶ **Definition 3.8** (Typing contexts)**.** *Θ denotes a partial mapping from process variables to n-tuples of types, and Γ denotes a partial mapping from channels to types, defined as:*

$$\Theta \;::=\; \emptyset \;\big|\; \Theta,\, X{:}S_1,\ldots,S_n \qquad \Gamma \;::=\; \emptyset \;\big|\; \Gamma, c{:}S$$

The composition Γ_1, Γ_2 is defined iff $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. We write $s \notin \Gamma$ iff $\forall \mathtt{p} : s[\mathtt{p}] \notin \mathrm{dom}(\Gamma)$ (i.e., session s does not occur in Γ).We write $\mathrm{dom}(\Gamma) = \{s\}$ iff $\forall c \in \mathrm{dom}(\Gamma)$ there is $\mathtt{p}$ such that $c = s[\mathtt{p}]$ (i.e., Γ only contains session s); and $\Gamma \leqslant \Gamma'$ iff $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\forall c \in \mathrm{dom}(\Gamma){:}\, \Gamma(c) \leqslant \Gamma'(c)$. We write $\Gamma \to \Gamma'$ with $\Gamma = \Gamma_0,\, s[\mathtt{p}]{:}\mathtt{q}\oplus_{i\in I}\mathtt{m}_i(S_i).S'_i,\, s[\mathtt{q}]{:}\mathtt{p}\&_{j\in J}\mathtt{m}_j(T_j).T'_j$ and $\Gamma' = \Gamma_0,\, s[\mathtt{p}]{:}S'_i,\, s[\mathtt{q}]{:}T'_j$ where types are defined modulo unfolding recursive types. We write $\Gamma \to^* \Gamma'$ for a transitive and reflexive closure of $\to$; and $\Gamma \to$ if there exists Γ' such that $\Gamma \to \Gamma'$.

$$\frac{\Theta(X) = S_1, \dots, S_n}{\Theta \vdash X{:}S_1, \dots, S_n}\ [\text{T-}X] \qquad \frac{S \leqslant S'}{c{:}S \vdash c{:}S'}\ [\text{T-sub}] \qquad \frac{\forall i \in 1..n \quad c_i{:}S_i \vdash c_i{:}\mathbf{end}}{\text{end}(c_1{:}S_1, \dots, c_n{:}S_n)}\ [\text{T-end}] \qquad \frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}}\ [\text{T-}\mathbf{0}]$$

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\&_{i\in I}\mathtt{m}_i(S_i).S_i' \quad \forall i\in I \quad \Theta \cdot \Gamma, y_i{:}S_i, c{:}S_i' \vdash P_i}{\Theta \cdot \Gamma, \Gamma_1 \vdash \dagger\, c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(y_i).P_i}\ [\text{T-}\&] \qquad \frac{\Theta \cdot \Gamma_1 \vdash P_1 \qquad \Theta \cdot \Gamma_2 \vdash P_2}{\Theta \cdot \Gamma_1, \Gamma_2 \vdash P_1 \mid P_2}\ [\text{T-}|]$$

$$\frac{\Gamma_1 \vdash c{:}\mathtt{q}\oplus\mathtt{m}(S).S' \quad \Gamma_2 \vdash c'{:}S \quad \Theta \cdot \Gamma, c{:}S' \vdash P}{\Theta \cdot \Gamma, \Gamma_1, \Gamma_2 \vdash \dagger\, c[\mathtt{q}]\oplus\mathtt{m}\langle c'\rangle.P}\ [\text{T-}\oplus] \qquad \frac{\Theta \cdot \Gamma \vdash P \qquad \text{sbj}(P) = \{c\} \qquad \Theta \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \textbf{try}\ \ P\ \ \textbf{catch}\ \ Q}\ [\ \text{T-try}\ ]$$

$$\frac{\text{end}(\Gamma) \qquad 0 \leq n}{\Theta \cdot \Gamma, s[\mathtt{p}_1]{:}S_1, \dots, s[\mathtt{p}_n]{:}S_n \vdash s\lightning}\ [\ \text{T-kill}\ ] \qquad \frac{\Theta \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma, c{:}S \vdash \textbf{cancel}(c).Q}\ [\ \text{T-cancel}\ ]$$

$$\frac{\Theta, X{:}S_1, \dots, S_n \cdot x_1{:}S_1, \dots, x_n{:}S_n \vdash P \qquad \Theta, X{:}S_1, \dots, S_n \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \textbf{def}\ \ X(x_1{:}S_1, \dots, x_n{:}S_n) = P\ \ \textbf{in}\ \ Q}\ [\text{T-def}]$$

$$\frac{\Theta \vdash X{:}S_1, \dots, S_n \quad \text{end}(\Gamma_0) \quad \forall i \in 1..n \quad \Gamma_i \vdash c_i{:}S_i}{\Theta \cdot \Gamma_0, \Gamma_1, \dots, \Gamma_n \vdash X\langle c_1, \dots, c_n\rangle}\ [\text{T-call}]$$

$$\frac{\Gamma' = \{s[\mathtt{p}]{:}S_{\mathtt{p}}\}_{\mathtt{p}\in I} \quad s \notin \Gamma \quad \text{safe}(\Gamma') \quad \Theta \cdot \Gamma, \Gamma' \vdash P}{\Theta \cdot \Gamma \vdash (\nu s{:}\Gamma')\, P}\ [\text{T-}\nu]$$

$$\frac{\Gamma' = \{s[\mathtt{p}]{:}G\restriction\mathtt{p}\}_{\mathtt{p}\in\text{roles}(G)}\ \text{or}\ \ \text{end}(\Gamma') \quad s \notin \Gamma \quad \Theta \cdot \Gamma, \Gamma' \vdash P}{\Theta \cdot \Gamma \vdash (\nu s{:}\Gamma')\, P}\ [\ \text{T-init}\ ]$$

Figure 4 Multiparty session typing rules. We highlight the new rules from [44].

Next, we define typing context properties defined by its reduction.

We say Γ *is safe*, written safe(Γ), if $\varphi(\Gamma)$ for some safety property φ. Similarly, for *deadlock-freedom* (df(Γ)) and *liveness plus* (live$^+$(Γ)). See [30] for the definitions. The reader can refer to [44] for more explanations of the typing context properties.

▶ **Definition 3.9** (Typing judgement)**.** The typing judgement for processes has the form:

$$\Theta \cdot \Gamma \vdash P \ \text{(with } \Theta/\Gamma \text{ omitted when empty)} \tag{8}$$

and are defined by the typing rules in Figure 4 with the judgements for process variables and channels. For convenience, we type-annotate channels bound by process definitions and restrictions. Note that end(Γ) denotes that Γ only contains type $\mathbf{end}$.

We explain each rule highlighting the new rules from [44].

(Affine) Branching/Selection [T-&] and [T-$\oplus$] are the standard rules for branching and selection, which can also type affine branching and selection. Note that the premise Γ in $\Theta \cdot \Gamma, y_i{:}S_i, c{:}S_i' \vdash P_i$ in [T-&] ensures that selecting one branch in the reduction rule defined by [C-Br] is sufficient for ensuring type soundness.

Try-Catch and Cancellation [T-try] is typing a try process: we ensure P has a unique subject and catch block process Q has the same session typing (similar with branching). [T-cancel] generates a kill process at its declared session.

Kill process [T-kill] types a kill process that appears during reductions: the cancellation of $s[\mathtt{p}]$ is broadcasting the cancellation to all processes which belong to session s.

Recursions [T-def] and [T-call] are identical to those of [44].

Restriction Processes are initially typed projecting a global type by [T-init], while running processes are typed by [T-ν] (see the proof of Theorem 3.12).

▶ **Example 3.10** (Typing AMPST processes). To demonstrate the typing rules we type the inner *try* process from the reduction example. Let $Q = \mathsf{try}\ R\ \mathsf{catch}\ \mathsf{cancel}(s[\mathtt{p}]).\mathbf{0}$ where $R = ?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}$ and $\mathtt{d}$ is of type $S_1 = \mathbf{end}$. We show that $\Gamma \vdash Q$ where $\Gamma = \mathtt{d}{:}S_1, s[\mathtt{p}]{:}S_2$ with $S_2 = \mathtt{r}\oplus\mathtt{res}(S_1).\mathbf{end}$.

$$\dfrac{\dfrac{s[\mathtt{p}]{:}S_2 \vdash s[\mathtt{p}]{:}S_2 \quad \mathtt{d}{:}S_1 \vdash \mathtt{d}{:}S_1 \quad \dfrac{\cdots}{s[\mathtt{p}]{:}\mathbf{end} \vdash \mathbf{0}}\,[\text{T-}\mathbf{0}]}{\Gamma \vdash ?\,s[\mathtt{p}][\mathtt{r}]\oplus\mathtt{res}\langle\mathtt{d}\rangle.\mathbf{0}}\,[\text{T-}\oplus] \quad \mathrm{sbj}(R) = \{s[\mathtt{p}]\} \quad \dfrac{\dfrac{\cdots}{\mathtt{d}{:}S_1 \vdash \mathbf{0}}\,[\text{T-}\mathbf{0}]}{\Gamma \vdash \mathsf{cancel}(s[\mathtt{p}]).\mathbf{0}}\,[\text{T-cancel}]}{\Gamma \vdash Q}\,[\text{T-try}]$$

3.3 Properties of affine multiparty session types

This subsection proves the main properties of AMPST processes. We first prove basic properties such as Subject Congruence and Reduction Theorems, then prove important properties, session fidelity, deadlock-freedom and liveness. The highlight is cancellation termination, which guarantees that once an exceptional behaviour is triggered, all parties in a single session can terminate as nil processes.

Unlike linear-logic based typing systems [36], we do *not* assume that the typing system is closed modulo $\equiv$. Instead, we prove closedness of $\equiv$ for tricky cases, e.g., kill and try-catches.

▶ **Theorem 3.11** (Subject Congruence). If $\Theta \cdot \Gamma \vdash Q$ and $Q \equiv P$, then we have $\Theta \cdot \Gamma \vdash P$.

By Theorem 3.11, AMPST processes satisfy *type soundness.*

▶ **Theorem 3.12** (Subject Reduction). Suppose $\Theta \cdot \Gamma \vdash P$ and Γ safe. Then, $P \to P'$ implies there exists Γ' such that Γ' is safe and $\Gamma \to^* \Gamma'$ and $\Theta \cdot \Gamma' \vdash P'$.

A single agent in a multiparty session s is a participant playing a single role $\mathtt{p}$ in s. We use the definition from [44] except the highlighted part, which now includes affine processes.

▶ **Definition 3.13** (A unique role process). *Assume $\emptyset \cdot \Gamma \vdash P$. We say that P:*

1. ***has guarded definitions** iff in each subterm of the form **def** $X(x_1{:}S_1, \ldots, x_n{:}S_n) = Q$ **in** P', for all $i \in 1..n$, $S_i \not\leqslant \mathbf{end}$ implies that a call $Y\langle \ldots, x_i, \ldots\rangle$ can only occur in Q as subterm of $\dagger\, x_i[\mathtt{q}]\sum_{j\in J}\mathtt{m}_j(y_j).P_j$ or $\dagger\, x_i[\mathtt{q}]\oplus\mathtt{m}\langle c\rangle.P''$ (i.e., after using x_i for selection/branching);*
2. ***only plays role** $\mathtt{p}$ **in** s**, by** Γ**,** iff: i) P has guarded definitions; ii) $\mathrm{fv}(P) = \emptyset$; iii) $\Gamma = \Gamma_0, s[\mathtt{p}]{:}S$ with $S \not\leqslant \mathbf{end}$ and $\mathrm{end}(\Gamma_0)$; iv) in all subterms $(\nu s'{:}\Gamma')\,P'$ of P, we have $\Gamma' = s'[\mathtt{p}']{:}\mathbf{end}$ (for some $\mathtt{p}'$).*

*We say "P **only plays role** $\mathtt{p}$ **in** s" iff $\exists\Gamma : \emptyset\cdot\Gamma\vdash P$, and item 2 holds.*

Note that by definition, a unique role process in s includes $s\lightning$.

Session fidelity is an important property to ensure liveness and deadlock-freedom, as well as termination. We extend that in [44] by taking a kill process into account. A set of unique role processes of a single multiparty session, together with kill processes always make progress if a typing context has progress, satisfying a protocol compliance.

Below we write $Q\lightning$ if Q contains only a parallel composition of kill processes.

▶ **Theorem 3.14** (Session Fidelity). *Assume $\emptyset\cdot\Gamma\vdash P$, where Γ is safe, $P \equiv |_{\mathtt{p}\in I} P_{\mathtt{p}} \mid Q\lightning$, and $\Gamma = \bigcup_{\mathtt{p}\in I}\Gamma_{\mathtt{p}} \cup \Gamma_0$ such that, for each $P_{\mathtt{p}}$, we have $\emptyset\cdot\Gamma_{\mathtt{p}}\vdash P_{\mathtt{p}}$; and $\emptyset\cdot\Gamma_0\vdash Q\lightning$. Assume that each $P_{\mathtt{p}}$ is either $P_{\mathtt{p}} \equiv \mathbf{0}$, or only plays $\mathtt{p}$ in s, by $\Gamma_{\mathtt{p}}$. Then, $\Gamma\!\to$ implies $\exists\Gamma', P'$ such that $\Gamma\to\Gamma'$, $P\to^* P'$ and $\emptyset\cdot\Gamma'\vdash P'$, with Γ' safe, $P' \equiv |_{\mathtt{p}\in I} P'_{\mathtt{p}} \mid Q'\lightning$, and $\Gamma' = \bigcup_{\mathtt{p}\in I}\Gamma'_{\mathtt{p}} \cup \Gamma'_0$ such that, for each $P'_{\mathtt{p}}$, we have $\emptyset\cdot\Gamma'_{\mathtt{p}}\vdash P'_{\mathtt{p}}$, and each $P'_{\mathtt{p}}$ is either $\mathbf{0}$, or only plays $\mathtt{p}$ in s, by $\Gamma'_{\mathtt{p}}$; and $\emptyset\cdot\Gamma'_0\vdash Q'\lightning$.*

By the above theorem, we can prove deadlock-freedom and liveness for a single session multiparty session in the presence of affine processes.

▶ **Definition 3.15** (Deadlock-freedom and liveness).

1. P is **deadlock-free** iff $P \to^* P' \not\to$ implies $P' \equiv \mathbf{0}$.
2. P is **live** iff $P \to^* P' \equiv \mathbb{C}[Q]$ implies: i) if $Q = c[\mathtt{q}] \oplus \mathtt{m}\langle s'[\mathtt{r}]\rangle.Q'$ (for some $\mathtt{m}, s', \mathtt{r}, Q'$), then $\exists \mathbb{C}'$: $P' \to^* \mathbb{C}'[Q']$; and ii) if $Q = c[\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).Q'_i$ (for some $\mathtt{m}_i, x_i, Q'_i$), then $\exists \mathbb{C}', k \in I, s', \mathtt{r}$: $P' \to^* \mathbb{C}'[Q'_k\{^{s'[\mathtt{r}]}/_{x_k}\}]$.

Note that liveness is defined for *linear* selection or *linear* branching processes which appear at the top level, i.e., under the reduction context $\mathbb{C}$, not under `try-catch` construct, cancel nor affine branching and selection processes.

▶ **Theorem 3.16** (Deadlock-freedom). *Assume* $\emptyset \cdot \Gamma \vdash P$*, with* Γ *safe,* $P \equiv \prod_{\mathtt{p}\in I} P_{\mathtt{p}}$*, each* $P_{\mathtt{p}}$ *either* $P_{\mathtt{p}} \equiv \mathbf{0}$*, or only playing role* $\mathtt{p}$ *in* s*. Then,* $\mathrm{df}(\Gamma)$ *implies that* $(\nu\tilde{s}{:}\Gamma)\,P$ *with* $\{\tilde{s}\} = \mathrm{dom}(\Gamma)$ *is deadlock-free.*

As discussed in [44, Definition 5.11], we require $\mathrm{live}^+(\Gamma)$ for proving liveness.

▶ **Theorem 3.17** (Liveness). *Assume* $\emptyset \cdot \Gamma \vdash P$*, with* Γ *safe,* $P \equiv \prod_{\mathtt{p}\in I} P_{\mathtt{p}}$*, each* $P_{\mathtt{p}}$ *either* $P_{\mathtt{p}} \equiv \mathbf{0}$*, or only playing role* $\mathtt{p}$ *in* s*. Then,* $\mathrm{live}^+(\Gamma)$ *implies that* P *is live.*

Now we consider a user-written Rust program with one session as an *initial program.*

▶ **Definition 3.18** (Initial program). We say $\vdash Q$ is an *initial program* if

1. $Q \equiv (\nu\tilde{s}{:}\Gamma) \prod_{\mathtt{p}\in G} P_{\mathtt{p}}$ with $\{\tilde{s}\} = \mathrm{dom}(\Gamma)$;
2. $P_{\mathtt{p}}$ only plays $\mathtt{p}$ in s;
3. in each subterm of the form, **def** $X(\tilde{x}) = Q$ **in** P', (1) Q is of the form **try** Q' **catch** P''; and (2) P'' does not contain any (free or bound) process call.
4. $\Gamma = \{s[\mathtt{p}]{:}G{\upharpoonright}\mathtt{p}\}_{\mathtt{p}\in G}, \Gamma'$ for some G and $\mathrm{end}(\Gamma')$;
5. $\vdash Q$ is derived using [T-init] instead of [T-ν]; and without [T-kill].

Condition (3) ensures that once a process moves to the *catch-block*, then it ensures finite computation; (4,5) state that the initial program starts conforming to a global protocol.

▶ Remark 3.19 (Initial processes). Condition (3) does not limit the expressiveness since the *try-block* can include infinite computations; and conditions (4,5) imply that an initial program typed by condition (1) has started. Notice that running (runtime) processes generated from the initial program are typed using [T-ν] and [T-kill]; hence the proof of the subject reduction holds with Lemma 3.20 below.

Before proving the main theorems, we state that a set of local types projected from a well-formed global type satisfy the safety property.

▶ **Lemma 3.20** ([44, Lemma 5.9]). *Let* $\Gamma = \{s[\mathtt{p}]{:}G{\upharpoonright}\mathtt{p}\}_{\mathtt{p}\in\mathrm{roles}(G)}$*. Then* $\mathrm{safe}(\Gamma)$*,* $\mathrm{df}(\Gamma)$ *and* $\mathrm{live}^+(\Gamma)$*.*

Now we state the two main theorems of this paper: deadlock-freedom, liveness and cancellation termination. The cancellation termination theorem states that once a kill signal is produced by cancellation or affine processes (due to a timeout or an error), then all processes are enabled to terminate. We start from deadlock-freedom.

▶ **Corollary 3.21** (Deadlock-freedom and liveness for an initial program). Suppose $\vdash Q$ is an initial program. Then for all P such that $Q \to^* P$, P is deadlock-free and live.

▶ **Theorem 3.22** (Cancellation Termination). *Suppose* $\vdash Q$ *is an initial program. If* $Q \rightarrow^* \mathbb{C}[s\lightning] = P'$*, then we have* $P' \rightarrow^* \mathbf{0}$.

▶ **Corollary 3.23** (Cancellation Termination of Affine and Cancel Processes). *Suppose* $\vdash Q$ *is an initial program.*

1. *If* $Q \rightarrow^* \mathbb{C}[\textbf{\textsf{cancel}}(s[\mathtt{p}]).Q'] = P'$*, then we have* $P' \rightarrow^* \mathbf{0}$.
2. *If* $Q \rightarrow^* \mathbb{C}[\mathbb{E}[\,?\,s[\mathtt{p}][\mathtt{q}]\sum_{i\in I}\mathtt{m}_i(x_i).P_i]] = P'$ *or* $Q \rightarrow^* \mathbb{C}[\mathbb{E}[\,?\,s[\mathtt{p}][\mathtt{q}]\oplus\mathtt{m}\langle s'[\mathtt{r}]\rangle.P]] = P'$*, then we have* $P' \rightarrow^* \mathbf{0}$.

▶ Remark 3.24 (Termination theorem). The cancellation termination theorem means that *there always exists a path* which leads to $\mathbf{0}$; and an initial program might not terminate even if it contains a process with $s\lightning$. This differs from the total termination, i.e., all paths are finite – a program will definitely stop as $\mathbf{0}$. However, if we apply *fair traversal sets*, i.e., fair scheduling, from [44, Definition 5.5], applying to processes in $\mathbb{C}[s\lightning]$, we can prove the total termination. Since these extensions require an introduction of labelled transition systems for processes, we leave it as future work.

4 Design and implementation of `MultiCrusty`

4.1 Challenges for the implementation of `MultiCrusty`

The three main challenges underpinning the implementation of AMPST in Rust are related to multiparty communications and ensuring correctness for affine channels.

(Challenge 1) Realising a multiparty channel by binary channels. AMPST relies on a *multiparty channel* – a channel that can communicate with several roles. In Rust, communication channels are peer-to-peer, e.g., they are *binary* [27]. To overcome this limitation, we extend an encoding of MPST into binary channels [42]. In this encoding, a multiparty channel can be represented as an *indexed tuple* of *one-shot binary channels* used in a sequence depending on the ordering specified by the type. This design ensures reception error safety by construction. Since each pair of binary channels is dual, then no communication mismatch can occur. We piggyback on this result by introducing meshed channels, which reuse an existing library of binary session types in Rust [27] with built-in duality guarantees. We explain the implementation of meshed channels in § 4.2. See [30] for usecases that demonstrate how to use `MultiCrusty` for programming distributed protocols.

(Challenge 2) Deadlock-freedom, liveness and termination. Duality is unfortunately insufficient to guarantee deadlock-freedom. The naive decomposition of binary channels leads to *hard to* detect deadlock errors [42]. To ensure liveness properties and correct termination of cancellation behaviour, we integrate `MultiCrusty` with two state-of-the-art verification toolchains – Scribble [24] and k-MC [32], that ensures meshed channel types are *correct*. The former generates correct meshed channel types in Rust, while the latter verifies a set of existing meshed channel types. In both cases, well-typed processes implemented using well-typed meshed channels are free from deadlocks, orphan messages and reception errors. We display the Rust types for our running example in § 4.3.

(Challenge 3) Affinity with `try-catch` and optional types. Rust does not have a native `try-catch` construct, but macros and optional types. We use them to design and implement a `try-catch` block and affine selection and branching. Channels can be implicitly or explicitly cancelled, and all processes are guaranteed to terminate gracefully in the event of a cancellation, avoiding endless cascading errors. We discuss our design choices in § 4.4.

```
pub struct MeshedChannels< S1: Session, S2: Session, R: Role, N: Role> {
pub session1: S1, pub session2: S, pub stack: R, pub name: N }
```

Figure 5 Generated MeshedChannles structure.

4.2 Meshed Channels in `MultiCrusty`

A multiparty channel in `MultiCrusty` is realised as an *affine meshed channel* (hereafter meshed channel), which has three ingredients: (1) a list of separate binary channels (one binary channel for each pair of participants); (2) a stack that imposes the ordering between the binary channels; and (3) the name of the role, whose behaviour is implemented by the meshed channel. For example, a meshed channel for a 3-party protocol can be generated using the macro `gen_mpst!(MeshedChannels, A, C, S)`: the name of the structure (`MeshedChannels`), and the three roles involved in the communication (`A, C, S`). Figure 5 shows the generated structure.

The generated structure, `MeshedChannels`, holds four fields. The first two fields, `session1` and `session2`, are of type `Session` which is a binary session type. Therefore, these fields store binary channels. `Session` in Rust is a `trait` and a `trait` is similar to an interface. The `Session` trait can be instantiated to three generic (binary session) types: an `End` type; a `Recv<T, S>` or a `Send<T, S>` type, with their respective payload of type `T` and their continuation of a binary session type `S`. This has important implications for the design and safety of our system. Since all pairs of binary channels are created and distributed across meshed channels at the start of the protocol, the binary type `Session` enforces that each pair of binary channels are dual. For example, the binary channel for role S inside the meshed channels for role A; and the binary channel for role A inside the meshed channels for role S are dual. This design ensures that, without using any external tools, our system is communication safe, no reception error can occur. This is insufficient to guarantee deadlock-freedom, which is why we utilise Scribble or bounded model checking, i.e., k-MC, as an additional verification step.

The rest of the fields of the `struct MeshedChannels` are stack-like structures, `stack` and `name`, which represent respectively the order of the interactions (in what order the binary channels should be used) and the associated role. For instance, the behaviour where role A has to communicate first with role S, then with role C and then the session ends, can be specified using a stack of type `RoleS<RoleC<RoleEnd>>`. Note that all stack types such as `RoleS` and `RoleC` are generated singleton types. Role names are codified as `RoleX<RoleEnd>` where `X` is the actual name of the participant. For instance, role A is realised as the singleton type `RoleA<RoleEnd>`. We chose this design for its readability and its ease of implementation: one can guess at a glance the current state of a participant.

The code generation macro `gen_mpst!` produces *meshed channels for any finite number of communicating processes*. For example, in the case of a protocol with four roles, the macro `gen_mpst!` will generate a meshed channel with five fields – one field for the binary session between each pair of participants (which is 3 fields in total), one field for the stack and one field for the name of the role that is being implemented.

4.3 Types for affine meshed channels

Meshed channel types – `MeshedChannels` – correspond to local session types. They describe the behaviour of each meshed channel and specify which communication primitives are permitted on a meshed channel. To better illustrate meshed channel types, we explain the type `RecA<N>` for role A (Authenticator) from Figure 2b. The types are displayed in Figure 6. The types of the meshed channels for the other roles, i.e., C and S are available in [30].

Following the protocol, the first action on A is an external choice. Role A should receive a choice from role C of either `Video` or `Close`. External choice is realised in `MultiCrusty` as an `enum` with a variant for each branch, where each variant is parameterised on the meshed

```
1   // Declare the name of the role
2   type NameA = RoleA<RoleEnd>;
3
4   // Binary session types for A and C
5   type AtoCVideo<N> = Recv<N, Send<N, Recv<ChoiceA<N>, End>>
6
7   // Binary session types for A and S
8   type AtoSVideo<N> = Send<N, Recv<N, End>>;
9
10  // Declare usage order of binary channels inside a meshed channel
11  type StackAInit = RoleC<RoleEnd>; // for the initial meshed channel
12  type StackAVideo = RoleC<RoleS<RoleS<RoleC<RoleEnd>>>>; // for branch Video
13
14  // Declare the type of the meshed channel
15  type RecA<N> = MeshedChannels<Recv<ChoiceA<N>, End>, End, StackAInit, NameA>;
16
17  // Declare an enum with variants corresponding to the different branches, \ie Video and End
18  enum ChoiceA<N> {
19    Video(MeshedChannels<AtoCVideo<N>, AtoSVideo<N>, StackAVideo, NameA>),
20    Close(MeshedChannels<End, End, RoleEnd, NameA>)
21  }
22
```

Figure 6 Local Rust types for role A (Authenticator) from Figure 2b

channel that will be used for that branch. The enum type `ChoiceA<N>` in line 18 precisely specifies this behaviour – two variants with their respective meshed channels. The branch `Close` is trivial since no communication apart from closing all channels is expected in this branch. Hence, the binary channels for S and A, and C and A are all `End`. The type of the meshed channel for the branch `Video` in line 19 is more elaborate. `MeshedChannels<AtoCVideo<N>, AtoSVideo<N>, StackAVideo, NameA>` specifies that the type of the binary channel for C and A is `AtoCVideo<N>`, the type of the binary channel for role S and role A is `AtoSVideo<N>`, the stack of the meshed channel is `StackAVideo`. The declaration `RoleC<RoleS<RoleS<RoleC<RoleEnd>>>>` specifies the order in which binary channels must the used – first the binary channel with C, then with role S, then with S again, and finally with C. The last argument specifies that this is a meshed channel for role A.

The meshed channel types can be written either by the developers and verified using an external tool, k-MC, or generated from a global protocol written in Scribble.

4.4 Exception and cancellation

Exception handling Rust does not have exceptions. Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error. `Result<T, E>` is a variant type with two constructors: `Ok(T)` and `Err(E)` where `T` and `E` are generic type parameters.

We leverage two mechanisms to implement the semantics presented in § 3, both of which rely on the `Result` variant type: (1) the `?` operator and (2) the `attempt!-catch` macro. The `?` is syntactic sugar for error message propagation. More specifically, each communication primitive is wrapped inside a `Result` type. For example, the return type of `recv()` is `Result<(T, S), Box<dyn Error>>`. The call `recv()` on the multiparty channel `s` triggers the attempt of the reception of a tuple containing a payload of type `T` and a continuation of type `S`.

If a peer tries to read a cancelled endpoint then an error message is returned. Therefore, if an error occurs during receive due to, for example, the cancellation of the other end of the channel, the `?` operator stops the `recv()` function and returns an `Err` value to the calling code. Then, the user can decide to handle the error or `panic!` and terminate the program.

Similarly, the `attempt!-catch` block is syntactic sugar that allows exception handling over multiple communication actions. For instance, the `attempt! M catch N` reduces to its failing clause `N` if an error occurs in any of the statements in `M`. The interested users can try the online Rust playground that demonstrates the implementation of `attempt!-catch` using the `and_then` combinator [41]. The `attempt! M catch N` corresponds to the `try-catch` in § 3.

The implementation follows the behaviour formalised by the reduction rules in § 3. In particular, it ensures that whenever an error happens, a session is cancelled ($s\lightning$). We utilise Rust drop mechanism. When a value in Rust goes out of scope, Rust automatically drops it by calling its destructor: the `Drop` method. A variable that cannot be cloned, such as a session `s`, is out scope when used in a function and not returned, such as when used in the `close()` and `cancel()` functions. We have customised this method by implementing the `Drop` trait, which explicitly calls `cancel()`. If an error occurs, and the meshed channel is not explicitly cancelled, the meshed channel is *implicitly cancelled* from its destructor. In the case of a `panic!`, the session `s` will be dropped, alongside all variables within the same function, when `panic!` is called. Similarly to the theory, `cancel(s)` is not mandatory and can be placed arbitrarily within the process. Calling `cancel(s)` is mostly used for expressiveness and mock tests purposes, when a failure, without `panic!`, needs to be simulated.

Session Cancellation We discuss all cases involving session cancellation below:

1. **Implicit vs explicit cancellation** Receiving on or closing disconnected sessions returns an error. As a result of the error, the multiparty channel `s` is cancelled by our underlying library, and all binary channels associated with `s` are disconnected. We call this an implicit cancellation. This behaviour implements rules [C-?Sel] and [C-?Br]. Alternatively, the user can also cancel the session explicitly.
2. **Raising an exception** An error occurs (1) as a result of a communication over a closed/cancelled channel, (2) as a result of a timeout on a channel, or (3) in case of an error in the user code. For example the function `get_video()` can return an error. Then the user can decide to (1) `cancel(s)` the session, (2) silently drop the session, or (3) proceed with the protocol. Even if the user does not explicitly call the `cancel(s)` primitive, Rust runtime ensures that the meshed channel is always cancelled in the end.
3. **Double cancellation** If a peer tries to cancel a session `s` that is already cancelled from another endpoint, then the cancellation is ignored. Note that in our semantics this behaviour is modelled using the structural congruence rules, namely $s\lightning \mid s\lightning \;\equiv\; s\lightning$.
4. **Cancel propagation** When a session is cancelled, no communication action can be used subsequently on that channel. The action `cancel(s)` cancels all binary channels that are a part of the meshed channel, which precisely simulates the kill process $s\lightning$. When a peer attempts to receive on a channel, if either side of the channel is cancelled, the operation returns an error, and the session in scope is dropped. This is exactly the behaviour for the channels from the `crossbeam-channel` library, and we inherit and extend this behaviour to our library. Since our *receive* happens on a binary channel, our extension ensures that all other binary channels that are in scope, and the ones that are in the stack, are also closed. Since these channels are closed, when other peers try to read from them, they will also encounter an error, and will subsequently close their channels.

5 Evaluations: benchmarks, expressiveness and case studies

We evaluate `MultiCrusty` in terms of run-time performance (**§ 5.1**), compilation time (**§ 5.1**) and applications (**§ 5.2**, see [30]). Through this section, we demonstrate the applicability of `MultiCrusty` and compare its performance with programs written in binary sessions and untyped implementations (Bare) using `crossbeam-channel`. The purpose of the microbenchmarks is to demonstrate the best and worst-case scenarios for the implementation: we have not considered performance as a primary consideration in the current implementation. The results show that rewriting multirole protocols from binary channels to affine meshed channels can have a performance gain in addition to the safety guarantees provided by MPST.

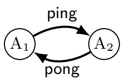
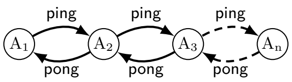
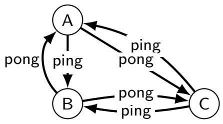
name	ping-pong	ring	full-mesh
diagram			

Figure 7 Protocols for Microbenchmarks

In summary, `MultiCrusty` has only a negligible overhead when compared to the built-in unsafe Rust channels, provided by `crossbeam-channel`, and up to two-fold runtime improvement to binary sessions in protocols with high-degree of synchronisation. The source files of the benchmarks and a script to reproduce the results are included in the artifact.

5.1 Performance

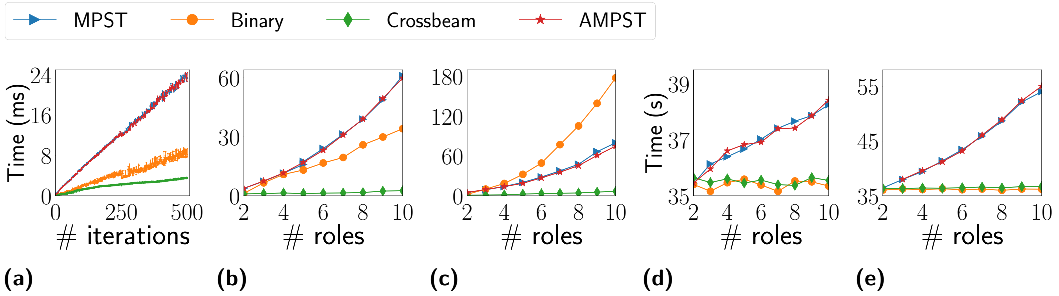
The goal of the microbenchmarks is two-fold. On one hand, it provides assurance that `MultiCrusty` does not incur significant overhead when compared to alternative libraries. The source of the runtime overhead of `MultiCrusty` can be attributed to: (1) the additional data structures that are generated (see § 4.2); and (2) checks for cancellation (as outlined in § 4.4). We also evaluate the efficiency of `MultiCrusty` when implementing multiparty (as opposed to binary) protocols. Multiparty protocols specify interaction dependencies between multiple threads. It is well-understood that a naive decomposition of multiparty protocol to a binary one (without preserving interaction dependencies) not only causes race conditions and wrong results but also deadlocks [42]. One may mitigate this problem by utilising a synchronisation mechanism, which is an off-the-shelf alternative to meshed channels. We compare the performance of `MultiCrusty` and meshed channels to a binary-channels-only implementation that uses thread-synchronisation.

We compare implementations, written using (1) `MultiCrusty` API (MPST) without cancellation; (2) `MultiCrusty` API with cancellation (AMPST); (3) binary channels, following [27] (BC); and (4) a Bare-Rust implementation (Bare) using untyped channels as provided by the corresponding transport library `crossbeam-channel`. As a reminder, `MultiCrusty` uses [27]'s channels (which are binary only and technically non-meshed), and [27]'s channels use `crossbeam-channel` for actually sending and receiving payloads: the scaffolding of all programs differs only in the final communication primitives used. In addition, the BC implementations synchronise between threads when messages must be received in order.

Figure 7 shows simple visualisation, displayed for illustrative purpose, of the three examples that we benchmark. Figure 8 reports the results on runtime performance, i.e., the time to complete a protocol by the implemented endpoints in Rust, and compilation time, i.e., the time to compile the implementations for all roles. We stress tested the library up to 20 participants but only show the results up to 10 participants for readability.

Setup: Our machine configurations are AMD Opteron^{TM} Processor 6282 SE @ 1.30 GHz with 32 cores/64 threads, 128 GB of RAM and 100 GB of HDD with Ubuntu 20.04, and with the latest version available for Rustup (1.24.3) and the Rust cargo compiler (1.56.0). We use *criterion* [26], a popular benchmark framework in Rust. We repeat each benchmark 10000 times and report the average execution time with a fairly narrow confidence interval of 95%.

Ping-pong benchmark measures the execution time for completing a recursive protocol between two roles repeatedly increasing the number of executions for request-response unit messages. Figure 8a displays the running time w.r.t. the number of iterations. This protocol is binary, and this benchmark measures the pure overhead of MPST implementation. MPST directly reuses the BC library, adding the structure `MeshedChannels` on top of it. Since both implementations need the same number of threads, the benchmark compares only the



(a) **(b)** **(c)** **(d)** **(e)**

Figure 8 Execution time (ms) for Ping-pong (a), Ring (b), Mesh (c) and compile time (s) for Ring (d), Mesh (e)

overhead of `MeshedChannels`. Both MPST and AMPST have a linear performance increase compared to BC and Bare. MPST is about 2.5 times slower than BC and about 6.5 times slower than Bare for 500 iterations.

Ring protocol, as seen in Figure 7, specifies `N` roles, connected in a ring, sending one message in a sequence. This example is sequential and stress tests the usage of numerous binary channels in an `MultiCrusty` implementation. Figure 8b displays the running time w.r.t. the number of participants. We measure the time to complete 100 rounds of a message for an increasing number of roles. This benchmark demonstrates a worst-case scenario for `MultiCrusty` since the MPST implementation requires `N*N` binary channels, hence `N*N` interactions at most, meanwhile the other implementations only need `2*N` binary channels. `MultiCrusty` is increasingly slower than the other implementations following a quadratic curve. All the implementations are running at the same speed for 2 participants; MPST becomes almost 2 times slower than BC for 10 participants and almost 3.25 times slower for 20 participants. AMPST implementation has a negligible overhead compared to MPST.

Full-mesh benchmark measures the execution time for completing a recursive protocol between `N` roles mutually exchanging the same message together: for every iteration, each participant sends and receives once with every other participant. For simplicity, we show the pattern in Figure 7 for three roles only. Figure 8c displays the running time w.r.t. the number of participants. This is a best-case scenario protocol for `MultiCrusty` since the protocol requires a lot of explicit synchronisation if implemented as a composition of binary protocols. The slowdown of BC is explained by the difference of implementation and the management of threads: the `MultiCrusty` needs only one thread for each participant, meanwhile for the binary case, two threads per pair of interactions are required to ensure that the message causalities are preserved. All implementations have similar running time for 2 participants but MPST is about 2.3 times faster than BC, and about 11 times slower than Bare for 10 participants. The figure only displays the results for up to 10 participants, since this is sufficient to show the overhead trend. In practice, we measured for up to 20 participants. For reference, at 20 participants, MPST is about 12 times slower than Bare and about 3.75 times faster than BC. As expected, AMPST has almost the same running time as MPST.

Results summary on execution time Overall, `MultiCrusty` is faster than the BC implementation when there are numerous interactions and participants, thanks to the encapsulation of each participant as a thread; the worst-case scenario for `MultiCrusty` is for protocols with many participants but no causalities between them which results in a slowdown when compared with BC. AMPST adds a negligible running time due to the simple checking of the status of the binary channels.

Results summary on compilation time We also compare the compilation time of the three protocols using `cargo build`. The results are presented in Figures 8d and 8e. As expected, the more participants there are, the higher is the compilation time for MPST, with up to 40% increase for the full-mesh protocol and only 11% for the ring protocol. We omit the graph for the ping-pong protocol since the number of iterations does not affect compilation time and the number of generated types, hence the compilation stays constant at 36.4s (MPST), 36.6s (AMPST), 36.1s (BC) and 36.3s (Bare).

The compilation time of BC and Bare are very close thanks to Rust's features, a mechanism to express conditional compilation and optional dependencies. This allows compiling only specific parts of libraries, instead of the whole libraries, depending on the needs of each file. For BC and Bare, we only compile `MultiCrusty`'s *default* features, meanwhile for MPST and AMPST, we also compile the *macros* features, which include heavy blocks of code and new dependencies for the creation of the new roles, meshed channels and associated functions.

5.2 Expressiveness

We demonstrate the expressiveness and applicability of `MultiCrusty` by implementing protocols for a range of applications. We also draw the examples from the session types literature, well-established *application protocols* (OAuth, SMTP), and distributed protocols (logging, circuit breaker). Protocols with more than 5 participants are not considered since having one global protocol with more participants can quickly become intractable in terms of protocol logic and is considered bad practice. The global protocols and patterns in the literature that have many participants are parameterised [6], participants can be grouped in kinds having the same type. Thereby, this will avoid a combinatorial explosion.

Table 2 displays the examples and related metrics. In particular, we report compilation time (Check./Comp./Rel.), execution time (Exec. Time), the number of lines of code (LoC) for implementing all roles in `MultiCrusty`, the lines of code generated from Scribble (Gen Types) and the total lines of code (All); the two following columns indicate whether the protocol involves three participants or more (MP), and if the protocol is recursive (Rec).

We report three compilation times corresponding to the different compilation options in Rust – `cargo check` which only type checks the code without producing binaries, `cargo build` which compiles the code with binaries and `cargo build -release` which, in addition, optimises the compiled artifact. Each recursive protocol is built/checked 100 times, and we display the average in the table. All protocols are type-checked within 27 seconds, while the basic compilations range between 36s and 41s and the optimised compilations vary between 80s and 97s. Those results represent the longest time we can expect for the respective build/check: Rust compilation is iterative, therefore, the usual compilation time should be shorter. A 30 seconds pause is short enough to not break the *flow* [52] of the mental headspace focused on the current task. Building the binaries takes longer, because of two heavy libraries used by `MultiCrusty` (`tokio` [48] and `hyper` [46]). The execution time of the protocols is measured by implementing only the communication aspects of the protocol, and orthogonal computation-related aspects are omitted. The execution time is the time to complete all protocol interactions, and even for larger protocols, it is negligible.

Table 2 does not contain protocols with more than 5 distinct participants because, in our experience, whenever more participants are needed, the protocol is parameterised [6]. We leave such extension for future investigation.

6 Related work

A vast amount of session types implementations based on theories exist, as detailed in the recent surveys on language implementations [1] and tools [14]. We discuss closely related works, focusing on (1) session types implementations in Rust (§ 6.1); (2) MPST top-down implementations (including other programming languages) (§ 6.2). For related work about

Table 2 Selected examples from the literature

Example (Endpoint)	Check./Comp./Rel./Exec. Time	LoC Impl.	Gen Types/All	MP	Rec
Three buyers [25]	26.7s / 37.1s / 81.3s / 568 μs	143	37 / 180	✓	✓
Calculator [19]	26.5s / 36.9s / 81.3s / 467 μs	136	32 / 168	✗	✗
Travel agency [21]	26.5s / 37.6s / 84.8s / 8 ms	200	47 / 247	✗	✓
Simple voting [19]	26.3s / 36.7s / 82.4s / 396 μs	207	61 / 268	✗	✗
Online wallet [39]	26.4s / 37.8s / 84.4s / 759 μs	231	76 / 307	✗	✓
Fibonacci [19]	26.6s / 36.7s / 80.9s / 9 ms	141	23 / 164	✗	✓
Video Streaming service (**§ 2**)	26.3s / 37.4s / 83.0s / 11 ms	104	39 / 143	✓	✓
oAuth2 [39]	26.4s / 37.5s / 83.2s / 12 ms	215	61 / 276	✓	✓
Distributed logging ([30])	26.5s / 36.8s / 82.6s / 5 ms	252	59 / 311	✗	✓
Circuit breaker ([30])	26.5s / 38.5s / 87.0s / 18 ms	375	142 / 517	✓	✓
SMTP [12]	26.4s / 41.1s / 97.3s / 5 ms	571	143 / 714	✗	✓

Affine types and exceptions/error handling in session types, see [30].

6.1 Session types implementations in Rust

Binary session types (BST) have been implemented in Rust by [24], [27] and [7], whereas, to our best knowledge, [9] is the only implementation of multiparty session types in Rust.

[24] implemented binary session types, following [17], while [27] based their library on the EGV calculus by [13] (See [30]). Both verify at compile-time that the behaviours of two endpoint processes are *dual*, i.e., the processes are compatible. The latter library allows to write and check session typed communications, and supports exception handling constructs. Rust originally did not support *recursive types* so [24] had to use *de Bruijn* indices to encode recursive session types, while [27] uses Rust's native recursive types but only handles failure for `recv()` actions: according to [27], this is generally the case with asynchronous implementations. This is because once an endpoint has received several messages, it makes sense to cancel them at the receiver rather than the sender. In fact, raising an exception on a send operation in an asynchronous calculus actually breaks confluence.

The library by [24] relies on an older version of Rust, hence we build `MultiCrusty` on top of [27]. Notice that we formalised AMPST guaranteeing the MPST properties of `MultiCrusty` (such as deadlock-freedom, liveness and cancellation termination), which are not present in [27]. In addition, our benchmarks confirmed that, in protocols where most of the participants mutually communicate, `MultiCrusty` is up to two times faster than [27].

[7] introduces their library, Ferrite, that implements BST in Rust, adopting *intuitionistic logic-based typing* [5]. The library ensures *linear* typing of channels, and includes a recently shared name extension by [2], but cannot statically handle prematurely dropped channel endpoints. Since Ferrite lacks an additional causal analysis for ensuring deadlock-freedom by [3], deadlock-freedom and liveness among more than two participants are not guaranteed, unlike `MultiCrusty`. Ferrite also lacks documentation and tests, making it hard to use.

[11] presents an implementation of a library for programming typestates in Rust. The library ensures that Rust programs follow a typestate specification. The tool, however, has several limitations. Differently than other works on typestates (e.g., typestates in Java [28]), [11] implements and verifies only binary non-recursive protocols, without a static guarantee that all branches are exhaustively implemented.

Note that all the above implementations are limited to *binary* and no formalism is proposed in their papers (see Table 3).

[9] implements MPST using `async` and `await` primitives. Their main focus is a performance analysis of asynchronous message reordering and comparisons of their asynchronous subtyping algorithm with existing tools, including the k-MC tool [32]. Their algorithm is a sound approximation of the (undecidable) asynchronous subtyping relation [15], by which their tool enables to check whether an unoptimised (projected from a global type) CFSM and its

optimised CFSM are under the subtyping relation or not. The main disadvantage of [9] is that their library depends on external tools for checking not only deadlock-freedom, but also communication-safety. Differently, `MultiCrusty` can guarantee *dual compatibility* (inherited from [27]) in a multiparty protocol, based on our meshed channels implementation.

Unlike `MultiCrusty`, neither failure handling nor cancellation termination is implemented or formalised in any of the above-mentioned works.

6.2 Multiparty session types implementations in other languages

We compare implementations of (top-down) MPST, ordered by date of publication, in Table 3, focusing on statically typed languages: we exclude MPST implementations by runtime monitoring such as Erlang [38] and Python [10].

The table is composed as follows, row by row:

Languages lists the programming languages introduced or used.

Mainstream language states if the language is broadly used among developers or not.

MPST top-down characterises the framework: Multiparty session types (MPST) or binary session types (BST). If the implementation allows the user to write MPST global types, it is called a top-down approach.

Linearity checking describes whether the linear usage of channels is not checked, checked at compile-time (*static*) or checked at runtime (*dynamic*).

Exhaustive choices check indicates whether the implementation can *statically* enforce the correct handling of potential input types. ✗ denotes implementations that do not support pattern-matching to carry out choices (branching) using switch statements on **`enum`** types.

Formalism defines the theoretical foundations of the implementations, such as (1) the end point calculus (the π-calculus (noted as π-cal.) or FJ [22]); (2) the (global) types formalism without any endpoint calculi (no typing system is given, and no subject reduction theorem is proved); or (3) no formalism is given (no theory is developed).

Communication safety outlines the presence or the absence of session type-soundness demonstration. Four languages, marked as △, provide the type safety only at type level. ✗$^\bullet$ means that the theoretical formalism does not provide linear types, therefore only type safety of base values is proved.

Deadlock-freedom is a property guaranteeing that all components are progressing or ultimately terminate (which correspond to deadlock-freedom in MPST). Four languages marked by △ proved deadlock-freedom only at the type level. ✓$^\bullet$ implies the absence of a formal link with the local configurations reduced from the projection of a global type. [1]

Liveness is a property which ensure that all actions are eventually communicated with other parties (unless killed by an exception in the case of AMPST).

Cancellation termination: once a cancellation happens at one of the participants in a multiparty protocol, the cancellation is propagated correctly, and all processes can terminate.

The Rust implementations in the first column of Table 3 are included for reference.

Most of the MPST implementations [20, 42, 37, 6, 35, 54] follow the methodology given by [19], which generates Java communicating APIs from Scribble [53, 45]. They exploit the equivalence between local session types and finite state machines to generate session types APIs for mainstream programming languages. [19, 20, 42, 37, 6] are not completely static: they check linearity dynamically. `MultiCrusty` can check linearity using the built-in

[1] [16] did not prove that any typing context reduced from a projection of a well-formed global type satisfies a safety property (a statement corresponding to Lemma 3.20). Hence, deadlock-freedom is not provided for processes initially typed by a given global type. Note that their typing contexts contain new elements not found in those defined in [20], which weakens the link with the top-down approach.

Table 3 MPST top-down implementations

	[27, 24, 7]	[40]	[19, 20]	[29]	[42]	[37]	[6]	[23]	[35]	[54]	[16]	[51]	[9]	`MultiCrusty`
Language	Rust	MPI-C	Java	Java	Scala	F#	Go	OCaml	Typescript	F*	EnsembleS	Scala	Rust	Rust
Mainstream language	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓
MPST Top-Down	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Linearity check	static	✗	dynamic	✗	dynamic	dynamic	dynamic	static	static	static	static	dynamic	static	static
Exhaustive choices check	✓	✗	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓
Formalism	✗	✗	types	FJ	π-cal.	✗	types	π-cal.	types	types	π-cal.	π-cal.	types	π-cal.
Communication safety	✗	✗	△	✓	✓	✗	△	✗•	△	△	✓	✓	△	✓
Deadlock freedom	✗	✗	△	✗	✓	✗	△	✗	△	△	✓•	✓	△	✓
Liveness	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Cancellation termination	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

affinity type checking from Rust. [40, 37, 6] do not enforce exhaustive handling of input types; and [19, 20, 29] rely on runtime checks to correctly handle branching.

[35, 54] provide static checking using the call-back style API generation. `MultiCrusty` uses a decomposition of AMPST to BST; in [42], MPST in Scala is implemented combining binary channels on the top of the existing BST library from [43]. Unlike `MultiCrusty`, [42] lacks static linearity check and uses a continuation-passing style translation from MPST into linear types. [29] implements static type-checking of communication protocols by linking Java classes and their respective typestate definitions generated from Scribble. Objects declaring a typestate should be used linearly, but a linear usage of channels is not statically enforced.

All above implementations generate multiparty APIs from protocols. To our knowledge, [23] is the only type-level embedding of classic multiparty channels in a mainstream language, OCaml. However, the library heavily relies on OCaml-specific parametric polymorphism for variant types to ensure type-safety. Their formalism lacks linear types and deadlock-freedom is not formalised nor proved. In addition, this implementation uses a non-trivial, complicated encoding of polymorphic variant types and lenses, while `MultiCrusty` uses the built-in affine type system in Rust.

The work most closely related to ours is [16] which implements handling of dynamic environments by MPST with explicit connections from [20], where actors can dynamically connect and disconnect. It relies on the actor-like research language, Ensemble; and generates endpoint code from Scribble. Their core calculus includes a syntax of the **try** L **catch** M construction where M is evaluated if L raises an exception. The type system follows [50], and is not as expressive as the previous paper on binary exception handling [13] that extends the richer type system of GV [34, 33]. Due to this limitation of their base typing system, and since their main focus is *adaptation*, there are several differences from AMPST, listed below: (1) they do not model general failure of multiple (interleaved) session endpoints (such as failures of *selection* and *branching* constructs as shown in rules [C-Sel], [C-Br]); (2) their **`try-catch`** scope (handler) is limited to a single action unlike AMPST and [13] where its scope can be an arbitrary process P, participants and session endpoints ([R-Cat]); (3) they do not model any Rust specific ?-options where an arbitrary process P can self-fail ([T-try], [C-?Sel]); and (4) their kill process is weaker than ours (it is point-to-point, it does not broadcast the failure notification to the same session).

As a consequence, their progress result ([13, Theorem 18]) is weaker than our theorems since their configuration can be stuck with an exception process that contains **raise**, while our termination theorem (Theorem 3.22) guarantees that there always exists a path such that the process will move or terminate as **0**, *cleaning up* all intermediate processes which interact non-deterministically. More precisely, in [13, Theorem 18], a cancellation in a session is propagated, but **raise** blocks a reduction when the actor is not involved in a session, and its behaviour is also **stop**, meaning it is terminated. Otherwise, the actor will leave the session and restart. In contrast, `MultiCrusty` ensures the strong progress properties by construction (see § 2). We also implemented interleaved sessions (as shown in [30]), where one participant is involved in two different protocols at the same time.

7 Conclusion and future work

Rust's pledge to guarantee memory safety does not extend to communication safety. Rust's built-in binary channels and affine type system are insufficient to ensure correct interaction and termination of multiple communicating processes. This paper overcomes this limitation by providing two main contributions. We proposed a new typing discipline, *affine multiparty session types*, which captures implicit/explicit cancellation mechanisms in Rust, and proved its cancellation termination theorem. In addition to progress and liveness properties, our end-point processes can guarantee that all processes terminate safely and cancellation is correctly propagated across all channels in a session, whenever and wherever a failure happens. We embedded the theory in Rust and developed a practical library for safe multiparty communication, `MultiCrusty`, which ensures deadlock-freedom and liveness in the presence of cancellations of arbitrary processes. Evaluation of `MultiCrusty` shows that it has only a negligible overhead when compared with the built-in unsafe Rust channels. We demonstrated the use of `MultiCrusty` for programming distributed application protocols with exception handling patterns.

As part of future work, we would like to develop recovery strategies based on causal analysis, along the lines of [38]. In addition, it would be interesting to verify role-parametric session types following [6] in an affine setting. Finally, we plan to study polymorphic meshed channels with different delivery guarantees such as TCP and UDP.

References

1 Davide Ancona, Viviana Bono, and Mario Bravetti. *Behavioral Types in Programming Languages*. Number 2-3. Now Publishers Inc., Hanover, MA, USA, 2016. doi:10.1561/2500000031.

2 Stephanie Balzer and Frank Pfenning. Manifest Sharing with Session Types. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110281.

3 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest Deadlock-Freedom for Shared Session Types. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639, Cham, 2019. Springer. doi:10.1007/978-3-030-17184-1_22.

4 Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983. URL: http://doi.acm.org/10.1145/322374.322380, doi:10.1145/322374.322380.

5 Luís Caires and Frank Pfenning. Session Types as Intuitionistic Linear Propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, pages 222–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-15375-4_16.

6 David Castro, Raymond Hu, SungShik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. Place: New York, NY, USA Publisher: Association for Computing Machinery. doi:10.1145/3290342.

7 Ruofei Chen and Stephanie Balzer. Ferrite: A Judgmental Embedding of Session Types in Rust. *CoRR*, abs/2009.13619, 2020. URL: https://arxiv.org/abs/2009.13619, arXiv:2009.13619.

8 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016. doi:10.1017/S0960129514000188.

9 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Optimising Asynchronous Communication in Rust: Deadlock-Free Message Reordering with Multiparty session Types, 2022. To appear at PPoPP 2022.

10 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *FMSD*, 46(3):197–225, 2015. URL: http://dx.doi.org/10.1007/s10703-014-0218-8, doi:10.1007/s10703-014-0218-8.

11 José Duarte and António Ravara. Retrofitting Typestates into Rust. page 83–91, 2021. doi:10.1145/3475061.3475082.

12 Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Technical Report RFC7230, RFC Editor, June 2014. URL: https://www.rfc-editor.org/info/rfc7230, doi:10.17487/rfc7230.

13 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types Without Tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, January 2019. Place: New York, NY, USA Publisher: ACM. doi:10.1145/3290341.

14 Simon Gay and António Ravara. Behavioural Types: from Theory to Tools. In *Behavioural Types: from Theory to Tools*, Automation, Control and Robotics, pages 1–412. Rivers publishers, Alsbjergvej 10, 9260 Gistrup, Denmark, 2017. URL: https://www.riverpublishers.com/dissertations_xml/9788793519817/9788793519817.xml, doi:10.13052/rp-9788793519817.

15 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434297.

16 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, page 30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://2021.ecoop.org/details/ecoop-2021-ecoop-research-papers/12/Multiparty-Session-Types-for-Safe-Runtime-Adaptation-in-an-Actor-Language, doi:10.4230/LIPIcs.ECOOP.2021.12.

17 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, ESOP '98, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:https://doi.org/10.1007/BFb0053567.

18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *SIGPLAN Not.*, 43(1):273–284, January 2008. doi:10.1145/1328897.1328472.

19 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering*, volume 9633, pages 401–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. URL: http://link.springer.com/10.1007/978-3-662-49665-724, doi:10.1007/978-3-662-49665-724.

20 Raymond Hu and Nobuko Yoshida. Explicit Connection Actions in Multiparty Session Types. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, volume 10202, pages 116–133. Springer Berlin Heidelberg, Berlin, Heidelberg,

2017. Series Title: Lecture Notes in Computer Science. URL: https://link.springer.com/10.1007/978-3-662-54494-57, doi:10.1007/978-3-662-54494-57.

21 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70592-5_22.

22 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.

23 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty Session Programming With Global Protocol Combinators. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13166, doi:10.4230/LIPIcs.ECOOP.2020.9.

24 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*, WGP 2015, page 13–22, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2808098.2808100.

25 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and Blame Assignment for Higher-Order Session Types. *SIGPLAN Not.*, 51(1):582–594, January 2016. doi:10.1145/2914770.2837662.

26 Aparicio Jorge. Crate: Criterion, 2021. Last accessed: July 2021. URL: https://crates.io/crates/criterion.

27 Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science*, 304:48–60, Sep 2019. URL: http://dx.doi.org/10.4204/EPTCS.304.4, doi:10.4204/eptcs.304.4.

28 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking Protocols with Mungo and stmungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, page 146–159, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2967973.2968595.

29 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and stmungo: A session type toolchain for Java. *Science of Computer Programming*, 155:52–75, April 2018. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2016. URL: https://www.sciencedirect.com/science/article/pii/S0167642317302186, doi:https://doi.org/10.1016/j.scico.2017.10.006.

30 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay safe under panic: Affine rust programming with multiparty session types. Technical report, 2022. arXiv: to appear.

31 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From Communicating Machines to Graphical Choreographies. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 221–232, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676964.

32 Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117, Cham, 2019. Springer. doi:10.1007/978-3-030-25540-4_6.

33 Sam Lindley and J. Garrett Morris. A Semantics for Propositions as Sessions. In Jan Vitek, editor, *Programming Languages and Systems*, pages 560–584, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46669-8_23.

34 Sam Lindley and J. Garrett Morris. Talking Bananas: Structural Recursion for Session Types. *SIGPLAN Not.*, 51(9):434–447, September 2016. doi:10.1145/3022670.2951921.

35 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Generating Interactive WebSocket Applications in TypeScript. *Electronic Proceedings in Theoretical Computer Science*, 314:12–22, April 2020. URL: `http://arxiv.org/abs/2004.01321v1`, `doi:10.4204/EPTCS.314.2`.

36 Dimitris Mostrous and Vasco T. Vasconcelos. Affine Sessions. *Logical Methods in Computer Science ; Volume 14*, 8459:Issue 4 ; 18605974, 2018. Medium: PDF Publisher: Episciences.org. URL: `https://lmcs.episciences.org/4973`, `doi:10.23638/LMCS-14(4:14)2018`.

37 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 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 2018, page 128–138, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3178372.3179495`.

38 Rumyana Neykova and Nobuko Yoshida. Let It Recover: Multiparty Protocol-Induced Recovery. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 98–108, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3033019.3033031`.

39 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local Verification of Global Protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *LNCS*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. `doi:https://doi.org/10.1007/978-3-642-40787-1_25`.

40 Nicholas Ng, Jose Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by Default. In Björn Franke, editor, *Compiler Construction*, pages 212–232, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. `doi:https://doi.org/10.1007/978-3-662-46663-6_11`.

41 Developers Rust. Rust: attempt-catch macro, 2018. Last accessed: July 2021. URL: `https://play.integer32.com/?version=stable&mode=debug&edition=2018&gist=95979b17196adbc203c4f563e00d384b`.

42 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–LeibnizZentrum fuer Informatik. ISSN: 1868-8969. URL: `http://drops.dagstuhl.de/opus/volltexte/2017/7263`, `doi:10.4230/LIPIcs.ECOOP.2017.24`.

43 Alceste Scalas and Nobuko Yoshida. Lightweight Session Programming in Scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:28, Dagstuhl, Germany, 2016. Schloss Dagstuhl–LeibnizZentrum fuer Informatik. ISSN: 1868-8969. URL: `http://drops.dagstuhl.de/opus/volltexte/2016/6115`, `doi:10.4230/LIPIcs.ECOOP.2016.21`.

44 Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. `doi:10.1145/3290343`.

45 Authors Scribble. *Scribble home page.* 2021. URL: `http://www.scribble.org`.

46 McArthur Sean. Crate: Hyper, 2021. Last accessed: July 2021. URL: `https://crates.io/crates/hyper`.

47 Company StackOverflow. Stackoverflow: 2020 Developer Survey, 2020. Last accessed: July 2021. URL: `https://insights.stackoverflow.com/survey/2020`.

48 Contributors Tokio. Crate: Tokio, 2021. Last accessed: July 2021. URL: `https://crates.io/crates/tokio`.

49 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’19, page 865–878, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3297858.3304069`.

50 Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theoretical Computer Science*, 368(1):64–87, 2006. URL: https://www.sciencedirect.com/science/article/pii/S0304397506003902, doi:https://doi.org/10.1016/j.tcs.2006.06.028.

51 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, October 2021. URL: https://dl.acm.org/doi/10.1145/3485501, doi:10.1145/3485501.

52 Contributors Wikipedia. Wikipedia: Flow (psychology), 2021. Last accessed: July 2021. URL: https://en.wikipedia.org/wiki/Flow_(psychology).

53 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, pages 22–41, Cham, 2014. Springer International Publishing. doi:https://doi.org/10.1007/978-3-319-05119-2_3.

54 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically Verified Refinements for Multiparty Protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428216.