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 autodetects 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).

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...
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
N. Lagaillardie, R. Neykova and N. Yoshida

Top-down approach
Global types in Scribble
Full project
Local types CFMS
Rust types
Programs written with MultiCrusty API
Type checking

Bottom-up approach
CFMS are (in)compatible
A-MC check
CFMS

(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
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 $\text{gen_mpst!}((\text{MeshedChannelsThree}, \ A, \ C, \ S)$, see line 2 in Figure 2. The macro $\text{gen_mpst!}$ takes two kinds of arguments: the name of the data structure for affine meshed channels, $\text{MeshedChannelsThree}$, and the name for each role, $\ A$, $\ C$ and $\ S$. $\text{MeshedChannelsThree}$ is a string literal that must be supplied by the developer, any name can be chosen. In our case, $\text{gen_mpst!}$ will generate a data structure called $\text{MeshedChannelsThree}$ 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 $\text{MeshedChannels}$. To explain affine meshed channels and all $\text{MultiCrusty}$ communication primitives, we focus on the implementation for role $\ A$ given in Figure 2b. The implementations of the other roles are similar. First, line 1 declares an $\text{auth}(s)$ function that is parametric on an affine meshed channel $s$ of type $\text{RecA<i32>}$, the result type of the function is irrelevant to our explanation, hence we have simply denoted it by $R$. The type $\text{RecA<i32>}$, 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, $c!\text{ResVideo}$ means that role $\ A$ is receiving from the role $\ C$ a message labelled as $\text{Video}$, while $s!\text{ReqVideo}$ says that role $\ A$ sends a message to role $\ S$.

The thread for role $\ A$ 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 $\ A$ and role $\ S$ and a binary channel for role $\ A$ and role $\ C$. The implementation starts by realising a choice: role $\ C$ 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 $\ C$ broadcasts the choice to every other role. This choice is received by role $\ A$, which will either receive a $\text{Video}$ or a $\text{Close}$ label. This behaviour is implemented by the $\text{MultiCrusty}$ macro $\text{offer_mpst!}$ (line 3), which is applied to a multiparty channel $s$ and a sum type, either $\text{ChoiceA::Video}$ or $\text{ChoiceA::Close}$ 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 $\ C$ sends a $\text{Video}$ label, while lines 13 – 16 handle the $\text{Close}$ request. At each step, the channel $s$ is rebound to a new meshed channel.
Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types

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;

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>let <em>s</em> = <em>s</em>.send(<em>p</em>);</td>
<td>Sends a payload <em>p</em> on a channel <em>s</em> and assigns the continuation of the session (a new meshed channel) to a new variable <em>s</em>.</td>
</tr>
<tr>
<td>let (<em>p</em>, <em>s</em>) = <em>s</em>.recv();</td>
<td>Receives a payload <em>p</em> on channel <em>s</em> and assigns the continuation of the session to a new variable <em>s</em>.</td>
</tr>
<tr>
<td><em>s</em>.close()</td>
<td>Closes the channel <em>s</em> and returns a unit type.</td>
</tr>
<tr>
<td>attempt!({ { ... } catch(<em>s</em>) { ... } })</td>
<td>Attempts to run the first block of code and, upon error, catches the error in the variable <em>s</em> and runs the second block of code.</td>
</tr>
<tr>
<td>offer_mpst!(<em>s</em>, { enum <em>i</em>: variants(<em>s</em>) =⇒ { ... } ∈ <em>K</em> })</td>
<td>Role <em>i</em> receives a message label on channel <em>s</em>, and, depending on the label value which should match one of the variants variants, of enum, runs the related block of code.</td>
</tr>
<tr>
<td>choose!(<em>s</em>, { enum <em>i</em>: variants[<em>s</em>] ∈ <em>I</em> })</td>
<td>Sends the chosen label, which corresponds to variants[<em>s</em>], to all other roles.</td>
</tr>
</tbody>
</table>

$s$ 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 *A* and role *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 *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 *A* and role *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 `S`’s sessions are inaccessible in the `catch(e)`-block.

Alternatively, we explain implicit cancellation as implemented by role `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 `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 `C`. With such modification, this function will still compile despite the protocol not being fully implemented (since the last received action from role `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{align*}
\text{variable, channel with role } p, \text{ error, flag} & : c, d ::= x \mid s[p] \mid \top ::= \emptyset \mid \oplus

\text{inaction, composition, restriction} & : P, Q ::= 0 \mid P \mid Q \mid (\nu s) P

\text{affine selection, branching } I \neq \emptyset & : \oplus c[q] \oplus m(d).P_i \mid \oplus c[q] \sum_{i \in I} m_i(x_i).P_i

\text{affine selection, branching } I \neq \emptyset & : c[q] \oplus m(d).P_i \mid c[q] \sum_{i \in I} m_i(x_i).P_i

\text{affine selection, branching } I \neq \emptyset & : \text{def } D \text{ in } P \mid X(\vec{c})

\text{catch process} & : \text{try } P \text{ catch } Q \mid \text{cancel}(c).P \mid s^2

\text{catch, cancel, kill} & : \text{catch } (\text{catch, cancel, kill})

\text{declaration of process variable } X & : D ::= X(\vec{x}) = P
\end{align*}
\]

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

The two processes with \( ? \) model the option \( ? \)-operator in Rust. Process \( \oplus c[q] \oplus m(d).P \) performs an affine selection (internal choice) towards role \( q \), using the channel \( c \) if the message label \( m \) 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) \( \oplus c[q] \sum_{i \in I} m_i(x_i).P_i \) uses channel \( c \) to wait for a message from role \( q \) if a message label \( m_i \) 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 \( m_i \) 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, \( \text{try } P \text{ catch } Q \), 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, \( \text{cancel}(c).P \), cancels other processes whose communication channel is \( c \). The kill \( s^2 \) 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, \( \text{def } X(\vec{x}) = P \text{ in } Q \) and process call \( X(\vec{c}) \) 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 \( \oplus c[q] \sum_{i \in I} m_i(x_i).P_i \) and \( \oplus c[q] \oplus m(d).P \). We use \( \text{fv}(P) / \text{lv}(P) \) and \( \text{dpv}(P) / \text{fpv}(P) \) to denote free variables/channels and bound/free process variables of \( P \). We call a process \( P \) such that \( \text{fv}(P) = \text{fpv}(P) = \emptyset \) closed. A set of subjects of \( P \), written \( \text{sbj}(P) \), is defined as: \( \text{sbj}(0) = \emptyset \); \( \text{sbj}(P \mid Q) = \text{sbj}(P) \cup \text{sbj}(Q) \); \( \text{sbj}( (\nu s) P ) = \text{sbj}(P) \setminus \{ s[p] \}_{s \in I} \); \( \text{sbj}(\oplus c[q] \sum_{i \in I} m_i(x_i).P_i) = \text{sbj}(\text{def } X(\vec{x}) = P \text{ in } Q) = \text{sbj}(Q) \cup \text{sbj}(P) \setminus \{ \vec{x} \} \) with \( \text{sbj}(X(\vec{c})) = \text{sbj}(P[\vec{x}/\vec{c}]) \); \( \text{sbj}(\text{try } P \text{ catch } Q) = \text{sbj}(P) \); and \( \text{sbj}(\text{cancel}(c).P) = \{ c \} \).

The set of subjects is the key definition which enables us to define the typing system for the try-catch process with recursive behaviours.
Another example is: which repeats the action at \([13, 16]\). Our (typable) try-catch processes allow any form of processes such as recursions, reductions at the top parallel composed processes,

\[ \text{Remark 3.5 (Nested try-catches and } P \text{). The context } E \text{ is only used for defining the reductions at the top parallel composed processes, not used nested exception handling like} \]

\[ [13, 16], \text{ Our (typable) try-catch processes allow any form of processes such as recursions,} \]
parallel, session delegations, and restriction/scope opened processes under a guarded process:

\[
R = \text{try } s[p][r] \oplus m_1 \cdot \{s'[r] \leftrightarrow m_3\} \cdot \text{try } s'[q][r] \oplus m_2 \cdot 0 \text{ catch cancel(s'[q]),} 0
\]

\[
\text{catch cancel(s[p]),} 0
\]

See [30] for more typable processes with nested \text{try-catch} blocks.

We explain each rule highlighting the new rules.

**Communication Rule** \(\text{[R-Com]}\) is the main communication rule between an affine/linear selection and an affine/linear branching. Linear selections/branching are placed in the \text{try} position but can interact with affine counterparts. Once they interact, processes are spawned from \text{try-blocks} (notice that \(E_1, E_2\) are erased after the communication), and start communicating on parallel with other parallel composed processes. Note that the context \(E\) is discarded after the successful communication.

**Error-Cancellation Rules** \(\text{[C-7Sel]}\) and \(\text{[C-7Br]}\) model the situations that an error handling occurs at the affine selection/branching. This might be the case if its counterpart has failed (hence \(\text{[R-Com]}\) does not happen) or timeout. It then triggers the kill process at \(s\). Rules \(\text{[T\#Sel]}\) and \(\text{[T\#Br]}\) model the case that the affine selection/branching are placed inside the \text{try-block} and triggered by the error. In this case, it will go to the \text{catch-block}, generating a kill process.

**Cancelling Processes** Rule \(\text{[C-Sel]}\) 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 \(\text{[C-Br]}\) cancels only one of the branches – this is sufficient since all branches contain the same channels except \(x_1\) (ensured by rule \(\text{[T-K]}\) in Figure 4). After the cancellation, it additionally instantiates a fresh name \(s'[r]\) to \(x_k\) into \(P_k\). The generated kill process at \(s'\) kills prefixes at \(s'[r]\) in \(P_k\{s'[r]/x_k\}\).

**Cancellation from Other Parties** Rule \(\text{[R-Can]}\) is a cancellation and generates a kill process.

Note that the \text{try-catch} context \(E\) is thrown away. Rule \(\text{[C-Cat]}\) is prompted to move to \(Q\) by kill \(s_j\). The side condition \text{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 \(\text{[R-Def]}, \text{[R-Ctx]}, \text{[R-Struct]}\) 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 \text{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 \(S\) is \(P = ?s[p][q](Q + \text{close}(x))\) where

\[
Q = \text{video}(x).\text{try } ?s[p][q]\text{req}(x).\text{try } ?s[p][r] \oplus \text{res}(d).\text{catch cancel}(s[p]).0 \text{ catch cancel}(s[p]).0
\]

The following shows a possible reduction.

\[
P | s[q][p] \oplus \text{video}(d).s[q][p] \oplus \text{req}(d).s[q][p] \oplus \text{res}(x).0 \quad (1)
\]

\[
\text{[R-Com]} \quad \text{try } (?s[p][q]\text{req}(x).\text{try } (?s[p][r] \oplus \text{res}(d).0 \text{ catch cancel}(s[p]).0
\]

\[
\quad \text{catch cancel}(s[p]).0)
\]

\[
| s[q][p] \oplus \text{req}(d).s[q][p] \oplus \text{res}(x).0 \quad (2)
\]

\[
\text{[R-Com]} \quad \text{try } ?s[p][r] \oplus \text{res}(d).0 \text{ catch cancel}(s[p]).0 \mid s[q][p] \oplus \text{res}(x).0 \mid s[q][p] \oplus \text{res}(x).0 \quad (3)
\]

\[
\text{[T\#Sel]} \quad \text{cancel}(s[p]).0 | s | s[q][p] \oplus \text{res}(x).0 \quad (4)
\]

\[
\text{[R-Can]} \quad s[j] | s[j] | 0 | s[q][p] \oplus \text{res}(x).0 \quad (5)
\]

\[
\text{[C-Br]} \quad s[j] | s[j] | 0 | 0 \equiv s[j] \quad (6)
\]

\[
E_0[P_0] \text{ for Equation (1) is } P_0 = ?s[p][q]\text{req}(x).\text{try } \ldots \text{ catch cancel}(s[p]).0 \text{ and } E_0[P_0] \text{ for Equation (4) is } P_0 = ?s[p][r] \oplus \text{res}(d).0 \text{ both because of rule } \text{[C-Cat]}.
\]
Initially we reduce using the communication rule for the branching and selection. Next, we apply [R-Con] demonstrating how the affine branching reduces under try. Then we apply [T[72]Se] assuming an error (or a timeout) occurs during the selection of res. This generates a kill process $s_j^i$ and spawns the process in the catch-block. Cancel spawns a kill process $s_j^i$ and hence reduces to $s_j^i | 0$, following rule [R-Con] with $E = [\cdot]$. Finally, applying [R-Con] cancels the linear selection. To conclude, we garble collect all kill processes. Given that our initial parallel composition has name restrictions ($\nu s$) at the top level, $(\nu s) s_j^i \equiv 0$.

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:

$G ::= p \rightarrow q ; \{m_i(S_i).G_i\}_{i \in I} \mid \mu t.G \mid t \mid \text{end}$ with $p \neq q, I \neq \emptyset$, and $\forall i \in I : \text{fv}(S_i) = \emptyset$

The syntax of local types is:

$S, T ::= p \& \{m_i(S_i).S_i'\} \mid p \oplus \{m_i(S_i).S_i'\} \mid \text{end} \mid \mu t.S \mid t$ with $I \neq \emptyset$, and $m_i$ pairwise distinct.

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

$m \in A$ corresponds to the usual message labels in the session type theory. Global branching type $p \rightarrow q; \{m_i(S_i).G_i\}_{i \in I}$ states that participant $p$ can send a message with one of the $m_i$ labels and a message payload type $S_i$ to the participant $q$ and that interaction described in $G_i$ follows. We require $p \neq q$ to prevent self-sent messages and $m_i \neq m_k$ for all $i \neq k \in J$.

Recursive types $\mu t.G$ are for recursive protocols, assuming those type variables $(t, t', \ldots)$ are guarded in the standard way, i.e., they only occur under branching. Type end represents session termination (often omitted). We write $p \in \text{roles}(G)$ (or simply $p \in G$) iff, for some $q$, either $p \rightarrow q$ or $q \rightarrow p$ occurs in $G$. The function $\text{id}(G)$ gives the participants of $G$.

For local types, the branching type $p \& \{m_i(S_i).S_i'\}$ specifies the reception of a message from $p$ with a label among the $m_i$ and a payload $S_i$. The selection type $p \oplus \{m_i(S_i).S_i'\}$ 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 $p$ is written $G | p$ and the standard subtyping relation, $\leq$. See [30].

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

Definition 3.8 (Typing contexts). $\Theta$ denotes a partial mapping from process variables to $n$-tuples of types, and $\Gamma$ denotes a partial mapping from channels to types, defined as:

$\Theta ::= \emptyset \mid \theta, X:S_1, \ldots, S_n \quad \Gamma ::= \emptyset \mid \Gamma, c:S$

The composition $\Gamma_1, \Gamma_2$ is defined iff $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$. We write $s \not\in \Gamma$ iff $\forall p : s[p] \not\in \text{dom}(\Gamma)$ (i.e., session $s$ does not occur in $\Gamma$). We write $\text{dom}(\Gamma) = \{s\} \iff \forall c \in \text{dom}(\Gamma)$ there is a $p$ such that $c = s[p]$ (i.e., $\Gamma$ only contains session $s$); and $\Gamma \leq \Gamma'$ iff $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ and $\forall c \in \text{dom}(\Gamma) : \text{dom}(\Gamma') \leq \Gamma'(c)$. We write $\Gamma \rightarrow \Gamma'$ with $\Gamma = \Gamma_0, s[p] : q \in \Gamma_0, S'_i, s[q] : p \& \{m_i(S_i).S_i'\}$. Then $\text{dom}(\Gamma) = \text{dom}(\Gamma') \iff \text{dom}(\Gamma) \leq \Gamma'(c)$.

We write $\Gamma \rightarrow \Gamma'$ for a transitive and reflexive closure of $\rightarrow$; and $\Gamma \rightarrow$ if there exists $\Gamma'$ such that $\Gamma \rightarrow \Gamma'$.
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 \( \Gamma \) is safe, written safe(\( \Gamma \)), if \( \varphi(\Gamma) \) for some safety property \( \varphi \). Similarly, for deadlock-freedom \( \text{df}(\Gamma) \) and liveliness \( \text{live}^+(\Gamma) \). See [30] for the definitions. The reader can refer to [44] for more explanations of the typing context properties.

\[ \text{Definition 3.9 (Typing judgement).} \quad \text{The typing judgement for processes has the form:} \]

\[ \Theta \cdot \Gamma \vdash P \quad \text{(with } \Theta/\Gamma \text{ omitted when empty)} \]

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

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

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

Try-Catch and Cancellation \([\text{T-try}]\) 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). \([\text{T-cancel}]\) generates a kill process at its declared session.

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

Recursions \([\text{T-def}]\) and \([\text{T-call}]\) are identical to those of [44].

Restriction Processes are initially typed projecting a global type by \([\text{T-init}]\), while running processes are typed by \([\text{T-\nu}]\) (see the proof of Theorem 3.12).
Example 3.10 (Typing AMPST processes). To demonstrate the typing rules we type the inner \texttt{try} process from the reduction example. Let \( Q = \texttt{try R catch cancel(s[p]).0} \) where \( R = ? s[p][r] @ \texttt{res}(d).0 \) and \( d \) is of type \( S_1 = \texttt{end} \). We show that \( \Gamma \vdash Q \) where \( \Gamma = \texttt{d:S_1,s[p]:S_2} \) with \( S_2 = r @ \texttt{res}(S_1) \).\texttt{end}.

\[
\frac{
\text{def} \ X(x_1:S_1, \ldots, x_n:S_n) = Q \text{ in } P', \text{ for all } i \in 1..n, \ S_i \not\subseteq \texttt{end} \implies \text{that a call } Y(\ldots, x_i, \ldots) \text{ can only occur in } Q \text{ as subterm of } \uparrow x_i[q] \sum_{j \in J} m_j(y_j).P_j \text{ or } \uparrow x_i[q] \oplus m(c).P'' \text{ (i.e., after using } x_i \text{ for selection/branching);}
}{
\text{only plays role } p \text{ in } s, \text{ by } \Gamma, \text{ iff: i) } P \text{ has guarded definitions; ii) } \text{fv}(P) = \emptyset;\iiii \Gamma = \Gamma_0, s[p]:S \text{ with } S \not\subseteq \texttt{end} \text{ and } \text{end}(\Gamma_0); \text{ iii) in all subterms } (\nu s':\Gamma') P' \text{ of } P, \text{ we have } \Gamma' = s'[p]:\texttt{end} \text{ for some } p'.
}
\]

We say “\( P \text{ only plays role } p \text{ in } s \)” iff \( \exists \Gamma : \emptyset \cdot \Gamma \vdash P \), and item 2 holds.

Note that by definition, a unique role process in \( s \) includes \( s \).

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 a single session can terminate as nil processes.

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 \( \Gamma \) safe. Then, \( P \rightarrow P' \) implies there exists \( \Gamma' \) such that \( \Gamma' \) is safe and \( \Gamma \rightarrow \Gamma' \) and \( \Theta \cdot \Gamma' \vdash P' \).

A single agent in a multiparty session \( s \) is a participant playing a single role \( p \) 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 \( \text{def} X(x_1:S_1, \ldots, x_n:S_n) = Q \text{ in } P', \text{ for all } i \in 1..n, \ S_i \not\subseteq \texttt{end} \implies \text{that a call } Y(\ldots, x_i, \ldots) \text{ can only occur in } Q \text{ as subterm of } \uparrow x_i[q] \sum_{j \in J} m_j(y_j).P_j \text{ or } \uparrow x_i[q] \oplus m(c).P'' \text{ (i.e., after using } x_i \text{ for selection/branching);}

2. only plays role \( p \) in \( s \), by \( \Gamma \), iff: i) \( P \) has guarded definitions; ii) \( \text{fv}(P) = \emptyset;\iiii \Gamma = \Gamma_0, s[p]:S \text{ with } S \not\subseteq \texttt{end} \text{ and } \text{end}(\Gamma_0); \text{ iii) in all subterms } (\nu s':\Gamma') P' \text{ of } P, \text{ we have } \Gamma' = s'[p]:\texttt{end} \text{ for some } p'.

We say “\( P \text{ only plays role } p \text{ in } s \)” iff \( \exists \Gamma : \emptyset \cdot \Gamma \vdash P \), and item 2 holds.

Note that by definition, a unique role process in \( s \) includes \( s \).

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 \( Q_\downarrow \) if \( Q \) contains only a parallel composition of kill processes.
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 if P ⊢ P′ holds.
2. P is live if P ⊢ P′ holds.
3. P is deadlock-free and live if P ⊢ P′ holds.

Before proving the main theorems, we state that a set of local types projected from a process type is deadlock-free. As discussed in [44, Definition 5.11], we require live+(Γ) for proving liveness.

Theorem 3.16 (Deadlock-freedom). Assume ⊢ P, with Γ safe, P ≡ |p∈Γ Pp, each Pp either Pp ≡ 0, or only playing role p in s. Then, df(Γ) implies that νs:Γ P with {s} = dom(Γ) is deadlock-free.

Theorem 3.17 (Liveness). Assume ⊢ P, with Γ safe, P ≡ |p∈Γ Pp, each Pp either Pp ≡ 0, or only playing role p in s. Then, live+(Γ) 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 ⊢ Q is an initial program if
1. Q ≡ (νs:Γ) |p∈G Pp with {s} = dom(Γ);
2. Pp only plays p in s;
3. in each subterm of the form, def X(¯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. Γ = {s[p]:G[P] |p∈G}, Γ′ for some G and end(Γ′);
5. ⊢ Q is derived using [T-init] instead of [T-s]; 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-s] 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 Γ = {s[p]:G[p] |p∈roles(G)}. Then safe(Γ), df(Γ) and live+(Γ).

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 ⊢ Q is an initial program. Then for all P such that Q ⊢ P, P is deadlock-free and live.
Theorem 3.22 (Cancellation Termination). Suppose $\vdash Q$ is an initial program. If $Q \xrightarrow{\ast} C[s_i] = P'$, then we have $P' \xrightarrow{\ast} 0$.

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

1. If $Q \xrightarrow{\ast} C[\text{cancel}(s[p]), Q'] = P'$, then we have $P' \xrightarrow{\ast} 0$.
2. If $Q \xrightarrow{\ast} C[E[s[p][q]\sum_{i \in I} m_i(x_i), P_i]] = P'$ or $Q \xrightarrow{\ast} C[E[s[p][q] \oplus m(s'[r]), P]] = P'$, then we have $P' \xrightarrow{\ast} 0$.

Remark 3.24 (Termination theorem). The cancellation termination theorem means that there always exists a path which leads to 0; and an initial program might not terminate even if it contains a process with $s_i$. This differs from the total termination, i.e., all paths are finite – a program will definitely stop as 0. However, if we apply fair traversal sets, i.e., fair scheduling, from [44, Definition 5.5], applying to processes in $C[s_i]$, 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 use cases 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.
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
// Declare the name of the role
type NameA = RoleA<RoleEnd>;

// Binary session types for A and C
type AtoCVideo<N> = Recv<N, Send<N, Recv<ChoiceA<N>, End>>;

// Binary session types for A and S
type AtoSVideo<N> = Send<N, Recv<N, End>>;

// Declare usage order of binary channels inside a meshed channel
type StackAInit = RoleC<RoleEnd>; // for the initial meshed channel
type StackAVideo = RoleC<RoleS<RoleS<RoleC<RoleEnd>>>>; // for branch Video

declare the type of the meshed channel
type RecA<N> = MeshedChannels<Recv<ChoiceA<N>, End>, End, StackAInit, NameA>;

declare an enum with variants corresponding to the different branches, \(\text{ie Video and End} \)
enum ChoiceA<N> {
    Video(MeshedChannels<AtoCVideo<N>, AtoSVideo<N>, StackAVideo, NameA>),
    Close(MeshedChannels<End, End, RoleEnd, NameA>)
}

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 ($s^1$). 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 $s^1 | s^2 \equiv s^1$.

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 $s^j$. 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.
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™ 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
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

<table>
<thead>
<tr>
<th>Example (Endpoint)</th>
<th>Check./Comp./Rel./Exec. Time</th>
<th>LoC Impl.</th>
<th>Gen Types/All</th>
<th>MP</th>
<th>Rec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three buyers [25]</td>
<td>26.7s / 37.1s / 81.3s / 568 µs</td>
<td>143</td>
<td>✓ ✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calculator [19]</td>
<td>26.5s / 36.9s / 81.3s / 467 µs</td>
<td>136</td>
<td>32 / 168</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Travel agency [21]</td>
<td>25.5s / 37.6s / 84.8s / 8 ms</td>
<td>200</td>
<td>47 / 247</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Simple voting [19]</td>
<td>26.3s / 37.5s / 82.4s / 759 µs</td>
<td>207</td>
<td>61 / 268</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Online wallet [39]</td>
<td>26.4s / 37.8s / 84.4s / 7 ms</td>
<td>231</td>
<td>76 / 307</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Fibonacci [19]</td>
<td>26.6s / 37.8s / 80.8s / 9 ms</td>
<td>252</td>
<td>23 / 164</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Video Streaming service ([§ 2])</td>
<td>26.3s / 37.4s / 83.0s / 11 ms</td>
<td>204</td>
<td>39 / 143</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OAuth2 [39]</td>
<td>26.4s / 37.5s / 82.2s / 12 ms</td>
<td>235</td>
<td>61 / 276</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distributed logging ([30])</td>
<td>26.5s / 36.3s / 82.6s / 5 ms</td>
<td>252</td>
<td>59 / 311</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Circuit breaker ([30])</td>
<td>26.5s / 35.5s / 87.0s / 18 ms</td>
<td>275</td>
<td>142 / 517</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SMTP [12]</td>
<td>26.4s / 41.1s / 97.3s / 5 ms</td>
<td>571</td>
<td>143 / 714</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

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. ✓ 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. ✓ implies the absence of a formal link with the local configurations reduced from the projection of a global type.

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.
Stable Safe under Panic: Affine Rust Programming with Multiparty Session Types

Table 3 MPST top-down implementations

<table>
<thead>
<tr>
<th>Language</th>
<th>Rust</th>
<th>MPST</th>
<th>Java</th>
<th>Scala</th>
<th>F#</th>
<th>Go</th>
<th>OCaml</th>
<th>TypeScript</th>
<th>F*</th>
<th>EnsembleS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mainstream</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Linearity check</td>
<td>static</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Exhaustive choices check</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Communication safety</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deadlock freedom</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Termination</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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


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


