University of Glasgow | School of Computing Science

Intrinsically-Typed Interpreter for Generalized Multiparty Session Calculus

Wind Wong
2612916w

ABSTRACT

In the age of modern computing where the majority of programs written involve a certain level of exchanges with other programs, be it microservices orchestration or client-server communication, there arises a need for some programming language construct to help programmers cope with the level of complexity involved. *Multiparty Session Types (MPST)* are type systems that allow for the specification of a protocol as a type, and let us statically verify an implementation matches the protocol specification by type-checking at compile time. For us to trust a MPST theory, one must first prove that the theory is *sound*, as in it only allows for the expression of valid protocols, and that its semantics indeed capture the progression of the protocol correctly. These proofs, however, are often very complicated and error-prone when done on pen-and-paper. A more rigorous proof formalized in a proof assistant would greatly increase our confidence in the type system.

This thesis project addresses this problem by the novel technique of formalizing and mechanizing a simple MPST calculi as a well-typed interpreter, and formalizing the proofs necessary to show that the MPST is type-sound. This lays the groundwork for future research into mechanizing type-soundness proofs of MPST extensions. All of the work is done in the proof assistant Agda.

1 INTRODUCTION

In the realm of distributed software systems, ensuring that communication between components adheres to predefined protocols is crucial. From microservices orchestrating complex business processes to APIs facilitating cross-platform interactions, these systems rely on precise and predictable communication patterns. A significant challenge in distributed software development is guaranteeing that implementations accurately reflect the intended protocol specifications. This is particularly important as it helps address key issues such as detecting potential mismatches between design and implementation early, reducing communication-related errors, and enhancing overall system reliability.

As distributed architectures become increasingly prevalent, the ability to verify that software components interact exactly as specified becomes paramount. Developers need robust tools and methodologies to bridge the gap between protocol design and implementation, ultimately leading to the creation of more dependable and interoperable distributed systems. One such approach is Multiparty Session Types (MPSTs).

MPSTs provide a formal framework for specifying and verifying communication protocols involving multiple participants. By defining a protocol using MPSTs, developers can ensure that each component in the system adheres to its role in the overall communication flow. If each participant's implementation successfully typechecks against its designated session type, the system as a whole is guaranteed to correctly implement the specified protocol, catching potential communication errors at compile-time rather than during runtime.

Consider a modern food delivery service as an example. Such a system involves multiple parties: customers placing orders, restaurants preparing meals, delivery drivers, and a central coordination system. Using MPSTs, we can define a protocol that ensures:

(1) When a customer places an order, it's always received by the central system.
(2) The restaurant is notified of new orders in the correct format.
(3) A driver is assigned to orders only after the restaurant confirms preparation.
(4) The customer receive an update when a driver has been assigned
(5) Finally, the customer receives an update when the driver arrives.

We can imagine from the client's implementation, there would be a channel of the type:

$$S \oplus \mathit{place}(\mathit{Order}).S \,\&\, \mathit{assigned}(\mathit{Update}).S \,\&\, \mathit{arrival}(\mathit{Update}).\mathit{end}$$

Here, S represents the target participant of a communication, and in this case S is the server. $\oplus$ and & represents the direction of the communication, i.e. send and receive respectively. The text in brown represents a message tag denoting what the message is supposed to be, and the text in blue represents the payload type of the message. By enforcing that the channel variable be used *exactly once* with linear typing rules, we can enforce the programmer to perform a send operation to the server to place an order as specified in the protocol, which then gives us a new channel variable of the type:

$$S \,\&\, \mathit{assigned}(\mathit{Update}).S \,\&\, \mathit{arrival}(\mathit{Update}).\mathit{end}$$

The programmer is now forced to implement receiving from the server a driver assignment update by consuming the new channel variable *exactly once*, this repeats itself until the programmer implements the entire protocol from the client's perspective until they obtain a channel variable of type *end*, at which point they can discard the variable. This demonstrates how MPSTs can encode protocol as types and forces programmers to correctly implement a protocol given a specification as type.

However, developing a theory for MPSTs presents significant challenges. The complexity arises from the need to ensure that all participants in a distributed system adhere strictly to their defined roles and that the exchange of messages aligns precisely with the specified protocol. One of the primary difficulties lies in verifying that participants send and receive the correct types of messages in the proper sequence. For instance, if a client expects to receive an update packet from the server, but the server instead sends an order packet, this constitutes as an invalid protocol. Developing a system to check for such discrepancies across multiple participants and various message types is a complex task.

As the expressiveness of an MPST theory increases, so does the complexity of proving the validity of the protocols it can express. More expressive theories allow for the specification of intricate communication patterns and dependencies between participants, but this increased flexibility comes at the cost of more complex proof that all protocols that can be expressed are indeed valid. A more complex proof implies a more *error-prone*, and Scalas and Yoshida in their *Less is More* paper [26] demonstrated this by showing the safety proofs in previous works on MPST extensions are in fact incorrect.

This prompts for a rigorous formalization of MPST theory using proof assistants to ensure the correctness of safety proofs. By encoding a generalized MPST theory in a proof assistant like Agda, we can leverage its powerful type system and theorem-proving capabilities to construct machine-checked proofs of critical properties. This approach not only provides a higher degree of confidence in the correctness of the theory but also allows for the exploration of more expressive extensions while maintaining provable safety guarantees. This serves as a reliable basis for developing practical MPST-based tools and type checkers, ensuring that the underlying theory is sound and that the protocols expressed within it are indeed valid.

1.1 Contributions

This thesis project's contributions are:

(1) Formalize and mechanize a type system for multiparty process calculi with an intrinsically-typed interpreter, providing a mechanization of MPST following the *Less is More* approach in Agda.
(2) Formalize the proof that subject reduction and session fidelity can be derived from a general definition of safety property.
(3) Provide different formal definitions and instantiations of common safety properties such as deadlock-freedom, termination, and liveness.

We implement the type system, operational semantics, and proof mechanisms for verifying safety properties in Agda. This work addresses a significant gap in current research, as no mechanization of MPST allowing for general safety properties in the *Less is More* approach has been attempted. Through this implementation, we demonstrate the practical advantages of working directly with local session types and provide a foundation for future development of verified multiparty communication protocols.

2 BACKGROUND

Session types [15][27], which originated in originated in process calculi and particularly typed π-calculi [21], are type systems used to model protocols in concurrent and distributed systems. They allow us to model protocols verify the described protocol to not contain any deadlock or communication errors, e.g. the data type being sent must match the data type the receiver is expecting, and that our implementation conforms to the specification of the protocol.

To elaborate further, we can draw comparisons to typed channels in Golang [11], a modern programming language where concurrency is treated as a first-class citizen. A channel of type `chan int` permits the transmission and reception of integer values exclusively. The type system enforces this constraint by rejecting operations attempting to transmit incompatible types, thus providing basic type safety guarantees, but it can only support one data type for one channel; whereas session types extend this paradigm by introducing more sophisticated structural constraints, allowing us to send different data types over the same channel in a type-safe manner. For example, consider this formal representation of a simplified SMTP protocol from the client side:

$$C = \oplus \left\{ \begin{array}{l} \mathit{ehlo}().\oplus \mathit{domain}(\mathit{Str}).\oplus \mathit{toAddr}(\mathit{Str}).\oplus \mathit{msg}(\mathit{Str}).\mathit{end} \\ \mathit{quit}().\mathit{end} \end{array} \right\}$$

This SMTP session type describes a protocol where the client can send either a message tag of `EHLO` or `QUIT`, both with an empty payload. If it chooses the `EHLO` branch, it must then send the tag `DOMAIN` with a string payload, the tag `TOADDR` with a string payload, the tag `MSG` with a string payload, and finally the protocol repeats. Otherwise, the client chooses the `QUIT` branch, and the protocol ends.

The session type of the SMTP protocol from the server side would be the dual type of the client type, where all operations are in inverse, i.e. send ($\oplus$) is replaced with receive (&).

$$S = \& \left\{ \begin{array}{l} \mathit{ehlo}().\&\mathit{domain}(\mathit{Str}).\&\mathit{toAddr}(\mathit{Str}).\&\mathit{msg}(\mathit{Str}).\mathit{end} \\ \mathit{quit}().\mathit{end} \end{array} \right\}$$

Fundamentally, the syntax of binary session types (without recursion) is:

$$S = \&_{i\in I} m_i(T_i).S_i' \mid \oplus_{i\in I}\, m_i(T_i).S_i' \mid \mathit{end}$$

The **external choice** $\&_{i\in I} m_i(T_i).S_i'$ says a process respecting this type will receive one input of the form $m_i(T_i)$, with message tag m_i and payload type T_i for any $i \in I$ chosen by the sender, then continuing as S_i. The **internal choice** $\oplus_{i\in I} m_i(T_i).S_i'$ is the dual of the former, saying that a process respecting this type will send one input of the form $m_i(T_i)$, for some $i \in I$ decided by the sender, then continuing as S_i. end says that a process respecting this type has terminated.

2.1 Subject Reduction

To show the type system of a process calculi is valid, we need to prove that the type system satisfy subject reduction. Informally, this means a well-typed process would never reduce to an erroneously typed process. Using the SMTP example, suppose we have a session $C|S$ (where | denote composition), and the client picks the `EHLO` branch, then the session would reduce to $C'|S'$, where C' and S' are:

$$\begin{aligned} C' &= \oplus\mathit{domain}(\mathit{Str}).\oplus \mathit{toAddr}(\mathit{Str}).\oplus \mathit{msg}(\mathit{Str}).\mathit{end} \\ S' &= \&\mathit{domain}(\mathit{Str}).\& \mathit{toAddr}(\mathit{Str}).\& \mathit{msg}(\mathit{Str}).\mathit{end} \end{aligned}$$

Subject reduction states that processes would never reduce to an erroneous process, i.e. one process is sending a message tag which the other party is not expecting, or one process is sending an accepted message tag but with the wrong payload type. This is done by showing any well-typed processes, if they can reduce, must reduce to a process that is also well-typed. By requiring processes to be dual to each other, we can trivially prove subject reduction for binary session types [33].

2.2 Session Fidelity

Another important property of a session type system is *session fidelity*, which states that if the type of a process can reduce, then the underlying process must also be able to reduce. Using the example in 2.1, we know that the type $C|S$ can reduce to $C'|S'$. By session fidelity, this implies that a process of type $C|S$ must be able to reduce to a new process of type $C'|S'$. This ensures that:

(1) The communication action that emerges from executing the typed process aligns with one of the permissible actions defined by the protocol's current phase.

(2) After the transition occurs, the resulting process maintains its well-typed status in relation to the unfinished portions of the protocol.

2.3 Multiparty Session Types

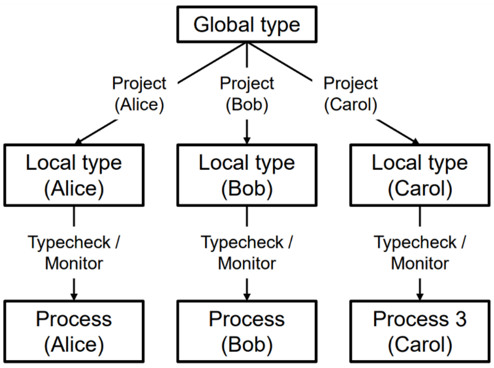

Figure 1: Projection of global type to local types

Multiparty Session Types (MPST) generalize binary session types and allow for specifying protocols involving multiple participants playing multiple roles. In the traditional view of MPSTs, the interactions between roles is described as a global type and then projected into multiple local types as demonstrated in 1. The main challenge of MPST comes from generalizing the notion of duality in binary session types, which is needed to prove subject reduction of the type system. Honda et al. [16] propose the notion of *consistency*, where a global session type G is *projected* to local session types $G \restriction p$ for each role p, as shown in figure 1. A notion of *partial projection* is then defined by taking a projection $G \restriction q$ and remove all communications except regarding another party q. Finally, G is said to be consistent if all for all $\forall p, q$ such that $p \neq q$, the partial projection of $G \restriction p$ regarding q is dual to the partial projection of $G \restriction q$ regarding p.

One problem with this approach is the complexity and limitations of projections. For example, it is not clear how to project $p \to q \colon \{l_i(S_i).G_i\}_{i \in I}$ to an uninvolved party r. The plain merge approach requires all branches G_i project to the same type, but this is very restrictive and leaves the projections of many valid global types undefined. Ghilezan et al. [10] remedy this by defining full merge which is a more permissive operation which merges compatible local types. However, even with full merge projections the consistency requirement is still overly restrictive. We cite an example from [26] which demonstrates a simple and valid protocol but is not consistent:

$$s \to c \colon \left\{ \begin{array}{l} login.c \to a \colon passwd(Str).a \to s \colon auth(Bool).end, \\ cancel.c \to a \colon quit.end \end{array} \right\}$$

Scalas and Yoshida in their *Less is More* paper [26] proposes an alternative approach by side-stepping global types entirely. In essence, reduction steps are directly defined on local session types, and if two roles p and q can take reduction steps that match each other, for example, $p \oplus q \colon m(S)$ and $q \& p \colon m(S)$, then the protocol can reduce in a synchronized step. Scalas and Yoshida then defined a generalized notion of safety property, proved subject reduction by showing that any type context satisfying a safety property φ must satisfy φ after reduction. They then gave several formal definitions of common safety properties, such as deadlock-freedom, termination, non-termination, and liveness. This approach is easily extensible and allows for a cleaner proof that can easily be adapted for MPSTs with custom extensions (e.g. crash-stop semantics [4], quantum communication [18]). It is also compatible with and generalizes the previous top-down method of defining protocol, and they have also proven that by the virtue of being projections of a global type, the composed local session types satisfy a safety property and therefore still retain the key properties of subject reduction and session fidelity.

2.4 Dependent Types

Traditional type systems in programming languages do not vary with program values, i.e. it is impossible to define a variable whose type varies depending on the value of the variable at compile-time. Dependent types extend this paradigm by allowing types to be parameterized by values. This additional expressiveness enables dependent types to encode richer properties of functions and data structures, facilitating the formal verification of both programs and mathematical proofs.

For instance, while most languages allow definitions such as `List<Int>` type for lists containing integers, they cannot restrict this type based on the length of the list or other properties of its contents. Dependent types make it possible to encode such properties directly within the type system. For example, one could define a type for lists of integers with length 3 as `Vec<Int, 3>`. It is also possible to make type signatures varies by runtime values, thus allowing the definition of functions such as: `append : Vec<n, a> -> Vec<m, a> -> Vec<n + m, a>`

This added expressiveness makes dependent types an essential tool for formal verification, as they allow correctness properties to be captured at the type level and verified mechanically during compile-time.

The utility of dependent types is further deepened by the **Curry-Howard correspondence**, which establishes an equivalence between constructive logic and typed functional programming. Specifically:

(1) Propositions in logic correspond to types, i.e., logical statements are represented as types.
(2) Proofs in logic correspond to programs that inhabit these types.
(3) Logical reasoning corresponds to program execution.

For example, in a dependently typed language such as Agda, the commutativity of addition over natural numbers $x + y = y + x$ can be expressed as type:

```
data _=_ : A → A → Set where
  refl : ∀ {x} → x = x

commutativity : (x y : Nat) → x + y = y + x
```

The `_=_` type can only be constructed with `refl` if both values at the left hand side and right hand side are the same. We can then state $x + y = y + x$ as a type signature as so, and proceed to prove commutativity for natural number to be true. This depends on the definition of the natural number datatype and the addition function, the most common being that from Peano arithmetic [23].

2.5 Agda

Agda is a dependently typed functional programming language and proof assistant. It implements an extension of Martin-Löf's constructive type theory [20] with features designed for practical programming. Proof assistants, like Agda, are software tools designed to help mathematicians and computer scientists develop and verify formal proofs of mathematical theorems and properties of computer programs. These systems combine a programming language with a powerful type system and logical framework, allowing users to express complex mathematical statements and construct proofs interactively. Proof assistants provide a high level of confidence in the correctness of proofs by mechanically checking each step, providing absolute certainty in mathematical proofs and complex system specifications, eliminating human error and implicit assumptions that often creep into paper proofs.

For example, the famous Four Color Theorem was verified using Coq [12], putting to rest decades of uncertainty about its correctness. In software verification, mechanization has been crucial in proving the correctness of critical systems like the seL4 microkernel (verified in Isabelle/HOL) [17] and CompCert C compiler (verified in Coq) [19], where bugs could have catastrophic consequences. In mathematics, the formal verification of the complex mathematical proof of Kepler's Conjecture using HOL Light [13], which helped validate Thomas Hales' original proof that had initially received only 99% confidence from human reviewers. This rigorous approach becomes increasingly important as both mathematical proofs and software systems grow more complex and harder to verify by hand.

This also applies to session types as the more complicated the type theory becomes, the more *error-prone* it is. Scalas and Yoshida showed that proofs of subject reduction depending on full merge projections in existing literature are flawed [26], and thus this prompts for mechanization of session calculi in proof assistants. Many existing works have been done on the mechanization of session types in proof assistants such as Agda [38][25], Coq [8][14], and Isabelle/HOL [9]. These formalizations all come with a proof of the type systems' subject reduction property, but they all concern session types involving only two parties. MPSTs based on the consistency of projections approach have been mechanized, for example in Coq by Tirore et al. [32], but compared to binary session types, mechanization of MPSTs are very scarce. We will review these works in depth in section 6. To the best of our knowledge, no mechanization of MPST allowing for a general safety property in the "Less is More" approach has been attempted.

2.6 Intrinsically-typed interpreter

One approach to implementing a programming language interpreter is writing it as an intrinsically-typed interpreter (also referred to as well-typed interpreters), which embeds the type system of the interpreted language directly into the interpreter's structure. In contrast to extrinsically typed interpreters, which perform type checking as a separate pass, intrinsically typed interpreters leverage the host language's type system to enforce typing rules. This approach eliminates the need for explicit runtime type checks, as type correctness is guaranteed by construction. The general structure of an intrinsic-typed interpreter has been standardized ever since Augustsson and Carlsson gave the overall framework of writing a well-typed interpreter in [2]. Several other works [5][3] then gave the framework on how to integrate de-Bruijn indices into intrinsic-typed interpreters to remove the complexity that comes with named variables representations. The structure has more or less settled to something akin to figure 2, where expressions are indexed by the type of the expression and a context containing the types of variables by their de-Bruijn indices.

```
data Exp : Ty → List Ty → Type where
    tt : Exp unit Γ
    var : a ∈ Γ → Exp a Γ
    lam : Exp b (a :: Γ) → Exp (fun a b) Γ
    app : Exp (fun a b) Γ → Exp a Γ → Exp b Γ

data Val : Ty → Type where
    tt : Val unit
    clos : Env Γ → Exp b (a :: Γ) → Val (fun a b)

 eval : Exp a Γ → Env Γ → Val a
 eval tt env = tt
 eval (var x) env = lookup env x
 eval (lam e) env = clos env e
 eval (app f e) env with eval f env
 ... | clos env' body = eval body (eval e env :: env')
```

Figure 2: Intrinsically-typed interpreter STLC in Agda

In the above example, we define two main data types: `Exp` for expressions and `Val` for values. The `Exp` type is indexed by the expression's type and a context (represented as a list of types Γ), while `Val` is indexed only by type. This indexing is key to the intrinsic typing approach.

The `Exp` data type includes constructors for:

(1) `tt`: Unit values, of type `unit`. This constructor represents a trivial value in the language, similar to () in many programming languages. It can be created in any context, hence the type `Exp unit Γ` for any context Γ.
(2) `var`: Variables, given a proof that the n-th variable in the context has type a, construct a value of type a. The $a \in \Gamma$ argument is a proof that type a is in the context Γ. This uses de-Bruijn indices to represent variables, where the index indicates how many binders we need to go outward to find the variable's declaration.
(3) `lam`: Lambda abstractions, given an expression of type b in a context extended with a variable of type a, construct a function of type $a \longrightarrow b$. The `Exp b (a :: Γ)` argument represents the body of the lambda, where the context is extended with the new variable of type a. This results in a function type `fun a b` in the original context Γ.
(4) `app`: Function applications, given a function of type $a \longrightarrow b$ and an argument of type a, construct a value of type b. Both the function and its argument must be well-typed in the same context Γ.

The `Val` data type includes constructors for:

(1) `tt`: Unit value, of type `unit`.
(2) `clos`: Closure values, representing functions. A closure consists of two parts:
    - `Env Γ`: An environment capturing the values of free variables at the time of function definition. This environment corresponds to the context Γ in which the function was defined.
    - `Exp b (a :: Γ)`: The function body, which is an expression of type b in a context extended with a new variable of type a.

The resulting closure has type `Val (fun a b)`, representing a function that takes an input of type a and returns a value of type b.

Finally, we define `eval`, which takes in an STLC expression of type a and context Γ, and the values of all the free variables in Γ, and returning a value of type a that is either a unit value or a closure. The type signature of `eval` not only encodes type soundness but also reflects the strong normalization property of STLC. In STLC, every well-typed term is guaranteed to evaluate to a normal form in a finite number of steps, which is precisely what this function type represents. The `eval` function, being total in Agda, must terminate for all inputs, directly encoding strong normalization. This means for any well-typed STLC expression (represented by `Exp a Γ`), evaluation always produces a value (`Val a`) in a finite number of steps, with no possibility of infinite loops or divergence. Thus, the type signature itself serves as a proof that STLC is not just type-sound but also strongly normalizing, demonstrating how intrinsically-typed interpreters can encode deep semantic properties of the language they implement by construction.

In this thesis, we apply intrinsically-typed interpreters to the domain of multiparty session calculi. Unlike STLC, which is well-behaved and strongly normalizing, process calculi present unique challenges due to their concurrent nature. Despite these complexities, we demonstrate that it is possible to construct an intrinsically-typed interpreter for a multiparty session calculus. This approach not only implements the calculus but also encodes its type system and key behavioral properties directly into the interpreter's structure. By doing so, we provide a mechanized proof of type soundness for our calculus and offer new insights into the semantics of concurrent systems.

3 IMPLEMENTATION

In this section, we first elaborate on the implementation of the intrinsically typed interpreter and main results proven in 3.1. We then elaborate on the design decisions and the flaws with previous iterations of the interpreter in 4.

3.1 Overview & Technical Approach

The syntax and typing rules of the multiparty session π-calculus we implemented are defined as follow.

3.1.1 Syntax of types and type environments.

$$
\begin{array}{rrcl}
\text{Ground value types} & V & ::= & \text{String} \mid \text{Int} \mid \text{Unit} \\
\text{Output session types} & S^! & ::= & p \oplus \{(V_i).S_i\}_{i\in I} \\
\text{Input session types} & S^? & ::= & p \,\&\, \{(V_i).S_i\}_{i\in I} \\
\text{Local session types} & S,T & ::= & S^! \mid S^? \mid \text{end} \\
\text{Session types} & \theta & ::= & \cdot \mid p \triangleleft S, \theta \\
\text{Types} & A & ::= & S \mid V \\
\text{Type environments} & \Gamma & ::= & \cdot \mid \Gamma, x\colon A
\end{array}
$$

3.1.2 Syntax of multiparty session π-calculus.

$$
\begin{array}{rrcl}
\text{Local Process} & P,Q & ::= & \mathbf{0} \mid p \oplus i(v).P \mid p \,\&_{i\in I}\, i(v_i).P_i \\
\text{Sessions} & M & ::= & \cdot \mid p \triangleleft M
\end{array}
$$

We define $pt(M)$ to be the set of all participants in M.

3.1.3 Value typing.

$$\frac{x\colon A \in \Gamma}{\Gamma \vdash x\colon A}\ \text{P-var} \qquad \frac{x \text{ has base type } V}{\Gamma \vdash x\colon V}\ \text{P-const}$$

3.1.4 Process typing.

$$\frac{}{\Gamma \vdash \mathbf{0}\colon \text{end}}\ \text{P-end} \qquad \frac{\exists i \in I \quad \Gamma \vdash P\colon S_i \quad \Gamma \vdash v\colon V_i}{\Gamma \vdash p \oplus i(v).P\colon p \oplus \{(V_i).S_i\}_{i\in I}}\ \text{P-send}$$

$$\frac{\forall i \in I \quad \Gamma, x\colon V_i \vdash P_i\colon S_i}{\Gamma \vdash p \,\&_{i\in I}\, i(v_i).P_i\colon p \,\&\, \{(V_i).S_i\}_{i\in I}}\ \text{P-recv}$$

3.1.5 Session typing.

$$\frac{\vdash P\colon S \quad \vdash M\colon \theta \quad p \notin pt(M)}{\vdash p \triangleleft P \mid M\colon p \triangleleft S \mid \theta}\ \text{P-sess}$$

3.1.6 Structural congruence and operation semantics.

$$\frac{\text{if } I \text{ is a permutation of } J}{|_{i\in I}(p_i \triangleleft P_i) \approx |_{j\in J}(p_j \triangleleft P_j)}$$

$$\frac{k \in I}{p \triangleleft q \oplus (v).P \mid q \triangleleft p \,\&_{i\in I}\, (x_i).Q_i \mid M \longrightarrow p \triangleleft P \mid q \triangleleft Q_k\{^{x_k}/_v\} \mid M}$$

$$\frac{M \longrightarrow N \quad M \approx M' \quad N \approx N'}{M' \longrightarrow N'}$$

We define $\longrightarrow^*$ to be the reflexive and transitive closure of the process reduction relation $\longrightarrow$.

3.1.7 Typing reduction.

$$\frac{k \in I}{p \oplus \{(V_i).S_i\}_{i\in I} \xrightarrow{\oplus p\colon k(V_k)} S_k}\ \theta\text{-}\oplus$$

$$\frac{k \in I}{p \,\&\, \{(V_i).S_i\}_{i\in I} \xrightarrow{\& p\colon k(V_k)} S_k}\ \theta\text{-}\&$$

$$\frac{S \xrightarrow{\oplus q\colon k(V)} S' \quad T \xrightarrow{\& p\colon k(V)} T'}{p \triangleleft S \mid q \triangleleft T \mid \theta \xrightarrow{p,q\colon k} p \triangleleft S' \mid q \triangleleft T' \mid \theta}\ \theta\text{-Comm}$$

$$\frac{\theta \text{ is a permutation of } \delta}{\theta \simeq \delta} \qquad \frac{\theta \xrightarrow{\alpha} \theta' \quad \theta \simeq \delta \quad \theta' \simeq \delta'}{\delta \xrightarrow{\alpha} \delta'}\ \theta\text{-Cong}$$

The typing reduction rules relate session types of processes and allow us to express how a protocol progresses by its specification in the form of a labelled transition system. This allows us to consider all possible state space of the protocol when reasoning with the progression of a protocol, and later on allow us to consider a subset of all the state space and possible progression paths by imposing a safety property φ (e.g. deadlock-freedom).

We define $\longrightarrow^*$ to be the reflexive and transitive closure of the typing reduction relation $\longrightarrow$, and $\theta \longrightarrow$ to mean $\exists\, \theta'.\, \theta \longrightarrow \theta'$.

3.1.8 Safety property. Let φ be a predicate on session types. φ is said to be a safety property iff:

(1) $\varphi(p \triangleleft q \oplus \{(V_i).S_i\}_{i\in I} \mid q \triangleleft p\&\{(V_j).T_j\}_{j\in J} \mid \theta)$ implies $I \subseteq J$ and $\forall i \in I.\, S_i = T_i$.
(2) $\varphi(\theta)$ implies $\varphi(\theta')$ if $\theta \approx \theta'$.
(3) $\varphi(\theta)$ implies $\varphi(\theta')$ if $\theta \longrightarrow \theta'$.

3.1.9 Claims. We make the claims that there exist correspondences between the LTS-based typing reduction rules defined in 3.1.7 and the operation semantics defined in 3.1.6. More specifically, the correspondences are:

(1) Subject reduction holds for the presented MPST. Formally speaking, assume $\vdash M : \theta$ and θ is safe by some safety property φ, then $M \longrightarrow M'$ implies $\exists\, \theta'$ such that θ' is φ-safe, $\theta \longrightarrow^* \theta'$, and $\vdash M' : \theta'$.
(2) Session fidelity holds for the presented MPST. Formally speaking, assume $\vdash M : \theta$, $\theta \longrightarrow$ and θ is safe by some safety property φ, then $\exists\, \theta', M'$ such that $\vdash M' : \theta'$, $\theta \longrightarrow \theta'$, and $M \longrightarrow^* M'$.

In our formalization, this is defined as:

```
SubjectReduction : Set _
SubjectReduction = ∀ θ θ'
  → (M : SExpr θ) (M' : SExpr θ')
  → M ⇒ M'
  → (θ ⇒* θ') × (φ θ')
SessionFidelity : Set _
SessionFidelity = ∀ θ (M : SExpr θ)
  → φ θ
  → ∃ (θ ⇒_)
  → ∃ λ θ' → (θ ⇒ θ') × Σ (SExpr θ') (M ⇒*_)
```

Our formalization of MPST is based on [37], with the exception that we do not have recursion and global types. To define a session type which specifies the global conversation scenarios of multiparty sessions, we instead use a collection of local session types as specified in the *P-sess* rule, which allow us to append a participant p's local process to a session, so long as the process contains no free variables (hence it is typable under an empty context), and p is not already in the session. The definition of safety property, typing reduction, subject property, and session fidelity are translated from [26], slightly modified since [26] uses channel-oriented typing, whereas ours uses process-oriented typing. For simplicity's sake, we also opted not to define session subtyping. This decision is elaborated more in 4.

While this formalization is not as generalized as a MPST could be, namely the lack of recursion implies no non-terminating sessions can be expressed, this suffices as an example MPST for the goal of formalizing the proof of subject reduction of generalized MPST in Agda. It is also worth noting that while at first glance the lack of nested sessions and the requirement of one-process-per-participant in a session restricts the level of non-determinism that the calculus can express, the one-process-per-participant restriction is inherently implied by session fidelity in the *Less is More* calculus [26], and subsequent works such as [4] also make this requirement more explicit.

First we formalize ground value types and local session types:

```
data VTy : Set where
  `str : VTy
  `nat : VTy
  `unit : VTy

data RTy : Set where
  `recv : (p : Role) → (I : Tag)
      → (S : I ⇒ VTy) → (T : I ⇒ RTy) → RTy
  `send : (p : Role) → (I : Tag)
      → (S : I ⇒ VTy) → (T : I ⇒ RTy) → RTy
  `end  : RTy

data DTy : Set where
  `proc : RTy → DTy
  `val : VTy → DTy

DCtx : Set
DCtx = List DTy
```

VTy and RTy correspond to ground value types and local session types respectively. We group both VTy and RTy under DTy, which allow us to express the context as a simple list of DTy, following the convention in 2. In the constructors RTy, Role is a parametrizable type, and Tag is simply a shorthand for List String. We require that Role has decidable equality, which is necessary for the proof of subject reduction and session fidelity later on. The type $I \Rightarrow A$ is a shorthand for $\forall x \rightarrow x \in I \rightarrow A$.

We then define PExpr, which encodes the syntax of local processes and the typing rules of values and processes.

```
data PExpr (Γ : DCtx) : DTy → Set where
  var : ∀ {t} → t ∈ Γ → PExpr Γ t
  -- Ground value expressions
  unit : PExpr Γ (`val `unit)
  str : String → PExpr Γ (`val `str)
  int : ℕ → PExpr Γ (`val `nat)
  -- Communication expressions
  0 : PExpr Γ (`proc `end)
  recv : ∀ q I V S
    → (P : Π I λ i →
        PExpr (`val (V i) :: Γ) (`proc (S i)))
    → PExpr Γ (`proc (`recv q I V S))
  send : ∀ q I V S → (i : Elem# I)
   → (v : PExpr Γ (`val (V i)))
   → (P : PExpr Γ (`proc (S i)))
   → PExpr Γ (`proc (`send q I V S))
```

In an intrinsically-typed interpreter, since the ADT of expressions are indexed by their types, we merge syntax and typing rules together into one single data type. The var, unit, str, int constructors correspond to the *P-var* and *P-const* rules. Similarly, **0**, *recv*, *send* correspond to the *P-end*, *P-recv*, and *P-send* rules directly. In the *recv* constructor, we take in an argument P which gives PExprs of type S_i typable under $\Gamma, v : S_i$ indexed by $i \in I$. In the *send* constructor, we instead take in an i such that $i \in I$, and then we take in a PExpr of type V_i and a PExpr of type S_i to construct a PExpr of type ‘send q I V S. We merge ground values and process expressions in a single datatype for reason we elaborate more in 4.

We can then define session expressions:

```
STy : Set
STy = List (Role × RTy)

pt : STy → List Role
pt = map proj1

data SExpr : STy → Set where
  [] : SExpr []
  act : ∀ p {S ty}
    → ¬ p ∈ pt S
```

```
  → PExpr [] (`proc ty)
  → SExpr S
  → SExpr ((p , ty) :: S)
```

Similarly, we directly encode the syntax and typing rules of sessions into one single data type. For readability, we define a pattern such that `p by p-uniq ◃ P | M` is expanded into `act p p-uniq P M`, such that the Agda formalization would resemble the paper notation $p \triangleleft P \,|\, M$, with the difference being that in Agda we have to explicitly pass in the proof $p \notin pt(M)$. We can then define the structural congruence relation on `SExpr`.

```
-- Structural congruence
data _≈_ : SExpr θ → SExpr θ' → Set where
  ≈-[] : [] ≈ []
  ≈-cong : ∀ {p-uniq p-uniq'}
    → M ≈ M'
    → (p by p-uniq ◃ pP | M)
        ≈ (p by p-uniq' ◃ pP | M')
  ≈-swap : ∀ {p-uniq q-uniq p-uniq' q-uniq'}
    → (p by p-uniq  ◃ pP | (q by q-uniq  ◃ qP | M))
    ≈ (q by p-uniq' ◃ qP | (p by q-uniq' ◃ pP | M))
```

The `≈-swap` and `≈-cong` constructors together allow us to relate any two `SExpr` that are different by only one pair of swapped adjacent elements. The `≈-cong` constructor is also sufficiently relaxed such that ≈ would be transitive. This allows us to relate any `SExpr` where the processes within are swapped, or equivalently, are permutations of each other.

We can then define safety properties as follows:

```
record Safety (φ : STy → Set) : Set where
  constructor is-safe
  field
    S-⊕&-α : φ ((p , `send q I pV pS)
      :: (q , `recv p J qV qS) :: Δ)
      → I ⊆ J
    S-⊕&-β : φ ((p , `send q I pV pS)
      :: (q , `recv p J qV qS) :: Δ)
      → ∀ i → pV i = qV i
    S-≈ : φ θ → IsPermutation θ θ' → φ θ'
    S→ : φ θ → θ ⇒ θ' → φ θ'
```

The first two constructors encode the first rule in 3.1.8. For ease of use, we split the implications $I \subseteq J$ and $\forall i \in I. S_i = T_i$ into two different constructors. The third and fourth constructor then encode the second and third rule 3.1.8, namely, a safety property must respect reduction and structural congruence.

We define substitution on `PExpr`, using standard constructions demonstrated in [36]:

```
_[_]' : PExpr (B :: Γ) A → PExpr Γ B → PExpr Γ A
```

This allows us to take in a process expression M such that $\Gamma, v : B \vdash M$, a process expression N such that $\Gamma \vdash N$, and returns $M[N]$ which is M with variable v substituted with the value of N, where v is the outermost variable in the context $\Gamma, v : B$ by de-Bruijn indices. While this substitution is limited to replacing only the topmost variable in the context, this suffices for defining the operation semantics of our MPST below:

```
-- Process reduction
data _⇒_ : SExpr θ → SExpr θ' → Set where
  R-Comm : (θ-safe : φ ((p , `send _ I pS pT)
          :: (q , `recv _ J qS qT) :: Δ))
      → p by p-uniq ◃
        send q I pS pT (m of m∈I) pE pP
        | q by q-uniq ◃ recv p J qS qT qP | M
      ⇒ p by p-uniq ◃ pP
        | q by q-uniq ◃ qP (m of m∈J)
         [ subst (PExpr [] . `val)
           (S-⊕&-β θ-safe m) pE ]' | M
  R-Cong : P ≈ P' → Q ≈ Q' → P ⇒ Q
      → P' ⇒ Q'

data _⇒*_ : SExpr θ → SExpr θ' → Set where
  ϵ : ∀ P → P ⇒* P
  _◃_ : P ⇒ Q → Q ⇒* R → P ⇒* R
```

This encodes the operation semantics specified in 3.1.6. To construct a proof that P reduces to Q, one constructs the datatype $P \Rightarrow Q$. To translate $Q_k\{^{x_k}/_v\}$ into our Agda formalization, one needs take into account that $x_k : V_q$ and $v : V_p$ might potentially have different types, therefore before we can perform substitution we need to prove that $V_q = V_p$, perform a substitution to obtain $v : V_q$, hence the use of the `subst` inside the `PExpr` substitution. The `subst : ∀ P -> x = y -> P x -> P y` is a standard Agda function which allows us to turn `pE : PExpr [] (‘val (pS m))` into `pE : PExpr [] (‘val (qS m))` provided a proof that `pS m = qS m`. We derive this proof by assuming that the type of the expression to be reduced is safe, hence the argument `θ-safe`, and extract the proof via `S-⊕&-β`. Finally, we can use the `PExpr` substitution that we have defined earlier, and formalize the operation semantics in Agda. We also define a reflexive and transitive closure of ⇒, which adds a reflexive constructor and a transitive constructor that allows chaining of ⇒.

To define typing reductions, we first define local type transitions. These correspond to θ-⊕ and θ-& in 3.1.7, and only act on local session types. An element of $S \rightarrow^l T$ represents a witness that S can reduce to T by either θ-⊕ or θ-&, indexed by a local label.

```
-- Transition labels
data LocalLabel : Set where
  send! : (q : Role) → (m : String)
    → (S : VTy) → LocalLabel
  recv! : (q : Role) → (m : String)
    → (S : VTy) → LocalLabel

-- Type transition
data _→ˡ_·_ : RTy → RTy → LocalLabel → Set where
  L-⊕ : ∀ q m S T k →
    `send q m S T →ˡ T k · send! q (Elem#.x k) (S k)
  L-& : ∀ q m S T k →
    `recv q m S T →ˡ T k · recv! q (Elem#.x k) (S k)
```

We can then define type reduction on session types as follows:

```
data SyncLabel : Set where
  sync! : (p q : Role) → (m : String) → SyncLabel

data _→ᵗ_·_ : STy → STy → SyncLabel → Set where
  S-Comm : ∀ p q m
```

```
    → st →ˡ st' · send! q m v
    → rt →ˡ rt' · recv! p m v
    → ((p , st) :: (q , rt) :: xs)
      →ᵗ ((p , st') :: (q , rt') :: xs)
      · sync! p q m
  S-Perm-α : ∀ xs xs' ys t → IsPermutation xs xs'
    → xs →ᵗ ys · t
    → xs' →ᵗ ys · t
  S-Perm-β : ∀ xs xs' ys t → IsPermutation xs xs'
    → ys →ᵗ xs · t
    → ys →ᵗ xs' · t

-- Type reduction
data _⇒_ : STy → STy → Set where
  `step : θ →ᵗ θ' · sync! p q m → θ ⇒ θ'

_⇒*_ : STy → STy → Set
```

Similarly, an element of $\theta \to^t \theta'$ represents θ can reduce to θ' by either θ-Comm or θ-Cong. For ease of use in later proofs, θ-Cong is split into two constructors without changing the semantics. S-Comm can be constructed if θ is of the form $S, T, \ldots$, given witnesses that S can reduce to S' locally by θ-$\oplus$, and T can reduce to T' locally by θ-&. This encapsulates the semantics of θ-Comm, and a combination of S-Perm-α and S-Perm-β encapsulate the semantics of θ-Cong. The type $\theta \to^t \theta'$ is indexed by a synchronization label which contains the necessary information to identify which reduction path θ took, but this information is not needed when we simply wants to state "θ reduces to θ'". For brevity and ergonomics, we wrap $\theta \to^t \theta'$ in a new type $\theta \Rightarrow \theta'$, which is not indexed by a synchronization label. Again, we also create a reflexive and transitive closure for the typing reduction relation $\Rightarrow$.

We can finally state our main results, subject reduction and session fidelity. We outline paper sketches of the proofs in our Agda formalization below.

3.2 Subject Reduction

```
SubjectReduction : Set _
SubjectReduction = ∀ θ θ'
  → (M : SExpr θ) (M' : SExpr θ')
  → M ⇒ M'
  → (θ ⇒* θ') × (φ θ')
```

The Agda formalization of subject reduction can be translated to "Assume $\vdash P : \theta$ and $\vdash Q : \theta'$, $P \to Q$ implies $\theta \to^* \theta'$ and θ' is safe". This is equivalent to the stated theorem in 3.1.9 minus the fact that θ is φ-safe. This is not a problem however since in our formalization $P \longrightarrow Q$ implies θ is φ-safe by construction, therefore $\varphi(\theta)$ can be omitted from the type signature. The Agda formalization of session fidelity is more straightforward and can be translated into the stated definition directly.

Lemma 3.1. *Given $\vdash M : \theta$ and $\vdash M' : \theta'$. $M \approx M'$ implies θ is a permutation of θ' and vice versa.*

Theorem 3.2. *Subject reduction holds for the presented MPST. Formally speaking, assume $\vdash M : \theta$ and $\vdash M' : \theta'$, $M \longrightarrow M'$ implies $\theta \to^* \theta'$ and θ' is safe.*

Proof outline We prove by induction on $M \to M'$:

(1) $M \to M'$ is constructed with R-Comm. This implies θ to be `(p , ‘send _ I pS pT) :: (q , ‘recv _ J qS qT) ::` Δ. We pattern match on p's expression to extract the sent tag m, We can construct the local labels `‘send _ I pS pT => pT m` and `‘recv _ J qS qT => qT m`. By $\varphi(\theta)$, we can prove `pT m = qT m` and construct a synchronization step from the two local labels.
(2) $M \to M'$ is constructed with R-Cong. This implies there exists some $N : \Delta, N' : \Delta'$ such that $N \longrightarrow N'$, $M \approx N$, and $N \approx N'$. By induction, we obtain $\Delta \to^* \Delta'$. By 3.1, we obtain θ, θ' being permutations of Δ, Δ' respectively. We apply S-$\approx$ to obtain a proof $\theta \to^* \theta'$.

Once we obtain a proof that $\theta \to^* \theta'$, we can prove $\varphi(\theta')$ by $\varphi(\theta)$ and S$\to$.

3.3 Session Fidelity

```
SessionFidelity : Set _
SessionFidelity = ∀ θ (M : SExpr θ)
  → φ θ
  → ∃ (θ ⇒_)
  → ∃ λ θ' → (θ ⇒ θ') × Σ (SExpr θ') (M ⇒*_)
```

Theorem 3.3. *Session fidelity holds for the presented MPST. Formally speaking, assume $\vdash M : \theta$, $\theta \longrightarrow$ and θ is safe by some safety property φ, then $\exists\, \theta', M'$ such that $\vdash M' : \theta'$, $\theta \to \theta'$, and $M \to^* M'$.*

Proof outline We extract θ_1 and $\theta \Rightarrow \theta_1$ from `∃ (θ ⇒_)`, and prove by induction on $\theta \Rightarrow \theta_1$.

(1) $\theta \Rightarrow \theta_1$ is constructed with S-Comm. This implies θ to be `(p , ‘send _ I pS pT) :: (q , ‘recv _ J qS qT) ::` Δ. We pattern match on p's expression to extract the sent tag m, and construct θ' to be `(p , pT m) :: (q , qT m) ::` Δ. To do this, we would need to prove $m \in J$ with $\varphi(\theta)$ by $I \subseteq J$. We also pattern match on M to extract p and q's continuation expressions to construct M'. We can then construct a synchronization step from θ to θ'.
(2) $\theta \Rightarrow \theta_1$ is constructed with S-Perm-α or S-Perm-β. In either case, we prove by induction and 3.1 as we did in 3.2.

With subject reduction proven we can conclude type soundness of the formalized MPST. With session fidelity we can show that any reduction of a safely typed process aligns with the set of actions permitted at the current stage of the session, and that the process that emerges after the reduction remains well-typed with respect to the remaining portion of the protocol yet to be completed. While type soundness does not directly translate from concurrent systems to functional programming, we do get the usual notions of preservation and progress if set of local types is deadlock-free.

4 DISCUSSION

In this section, we discuss several decisions made while attempting this thesis, how they have panned out, and how it led to the current iteration of the intrinsically typed interpreter.

4.0.1 Cubical Agda. The formalization of a multiparty session calculus presents significant challenges that go beyond those encountered with simpler systems like the λ-calculus. While the λ-calculus can be straightforwardly encoded in functional languages such as Haskell, multiparty session calculi require a more sophisticated approach due to complexity of the syntax, e.g. internal and external choices are both defined upon more complicated structures such as a set of message tags. The key challenge lies in finding both an appropriate host language that can express the calculus syntax and semantics accurately and a representation that makes formal reasoning practical. The host language must be expressive enough to prove essential properties about the system. Furthermore, the chosen representation needs to strike a balance between faithfulness to the original calculus and usability, as an overly complex encoding could make proof development unnecessarily burdensome.

Dependent-typed languages such as *Idris2* [6], *Agda* [1], *Coq* [30], and *Lean* [22] all are suitable candidates for this task. Coq and Lean were ruled out due to our unfamiliarity with both languages, and Agda is chosen since it is more designed for theorem proving, while Idris2 is designed more for general programming applications. Originally, a variant of Agda named Cubical Agda [35] was chosen, since it is more expressive than Agda in that quotient types can be expressed in Cubical Agda but not standard Agda.

One use case of Cubical Agda relevant to this thesis is expressing allowed message tags in local session types. The branches in our interpreter are indexed by a message tag and the representation of the accepted tags should be unordered. For example, the two types below should have the same representation.

$$\oplus p_{i \in \{auth, bye\}} m_i(S_i)$$
$$\oplus p_{i \in \{bye, auth\}} m_i(S_i)$$

A quotient type is a type which is "divided" (hence the name "quotient type") by an equality relation. For example, one can naively define the rationals $\mathbb{Q}$ as a tuple of integers $(\mathbb{Z}, \mathbb{Z})$. However, the tuple $(1, 2)$ and $(2, 4)$ would not be equal by itself. With quotient types we can define an equality relation stating that $\forall k. (km, kn) = (m, n)$. In a proof assistant without built-in support for quotient types such as Agda, we can either:

(1) Give up having the same representation and use an external data type to represent an equality relation $\sim$ that specifies $p \oplus q_{i \in \{auth, bye\}} m_i(S_i) = p \oplus q_{i \in \{bye, auth\}} m_i(S_i)$. We can then manually define a proof for every function $f : SessionType \to X$, it would respect the $\sim$ relation: $\forall x, y. x \sim y \to f(x) = f(y)$. This adds a lot of proof obligations to the user and is commonly referred to as "setoid hell".
(2) Use a special representation of a list of message tags that requires the list to be sorted, thus there can only be one representation of a list of accepted message tags, but it comes at the cost of having to prove the list is sorted every time we use it.

Cubical Agda allows us to model quotient types with higher inductive types (HITs) that let us directly encode equalities of the data type as path constructors. For example, we can use the definition of listed finite set from the standard library for Cubical Agda [29] to represent a list of accepted message tags:

```
data LFSet : (A : Type) → Type where
    []    : LFSet A
    _::_  : A → LFSet A → LFSet A
    dup   : ∀ x xs → x :: x :: xs = x :: xs
    comm  : ∀ x y xs → x :: y :: xs = y :: x :: xs
    trunc : isSet (LFSet A)
```

The [] and _::_ constructors are standard constructors from inductive lists. The *dup* path constructor allow us to remove any duplicates from the *LFSet*, and the *comm* path constructor allows us to arbitrarily swap the position of elements in the *LFSet*, giving us the unorderedness property that we desire, and the *trunc* path constructor states that *LFSet* is a *hSet*, the meaning of this is outside the scope of this thesis and we refer the readers to [24].

Another use case with Cubical Agda is that we can directly embed structural congruence rules into the data types of the expressions themselves. For example, consider these two SExpr:

$$p \triangleleft P \mid q \triangleleft Q \mid r \triangleleft R$$
$$q \triangleleft Q \mid r \triangleleft R \mid p \triangleleft P$$

These two expressions differ only by a permutation and therefore should be treated the same. Ideally we should be able to enforce this in SExpr, such that any definitions involving SExpr must respect this structural congruence rule. However, in standard Agda, we can only express structural congruence as an external data type outside of SExpr, allowing for definitions that do not necessarily respect structural congruence. While Cubical Agda would not reduce the proof obligations related to structural congruence, it would allow us to naturally state structural congruence as an *intrinsic* property of SExpr, not an *extrinsic* property that can be optionally disregarded.

However, after experiments with defining the interpreter in Cubical Agda, we reverted to using standard Agda. One draw back with Cubical Agda is its primitive support for indexed inductive types [28]. Pattern matching on indexed inductive types might not reduce, and in many cases pattern matching on inductive types indexed by a HIT would not work at all. This leads to many artifacts that complicates our definitions. Cubical Agda also uses a non-inductive definition of equality which complicates reasoning with equality. With inductive equality one can pattern match on an equality proof $x = y$ and the compiler would automatically unify the values of x and y. However, to reason with Cubical Agda's definition of equality, one must reason with intervals which is oftentimes very complicated to work out manually, or with helper functions that slow down type-checking and complicates proofs. We conclude that Cubical Agda's benefits do not necessarily help and its drawbacks manifest themselves very clearly in the context of intrinsically-typed interpreters, and that standard Agda is more suitable for defining intrinsically-typed interpreters.

4.0.2 Subtyping. While the MPST in [37] and [26] both support subtyping, we decidedly not to include subtyping in our MPST. Originally the formalization included subtyping:

```
data _≤_ : LocalST → LocalST → Type where
    sub-& : ∀ {p tags tags'}
      {tag-⊆-tags' : tags ⊆ tags'}
      → (contPrf : ∀ (l : allOf tags)
        → recvCont l ≤ recvCont' l)
      → `recv p tags' recvPayload' recvCont' ≤
           `recv p tags recvPayload recvCont
    sub-⊕ : ∀ {p tags tags'}
      {tag-⊆-tags' : tags ⊆ tags'}
      → (contPrf : ∀ (l : allOf tags)
        → sendCont l ≤ sendCont' l)
      → `send p tags sendPayload sendCont ≤
           `send p tags' sendPayload' sendCont'
    sub-end : `end ≤ `end
```

```
isPropST≤ : ∀ {S T} → isProp (S ≤ T)
```

To implement subtyping in an intrinsically-typed interpreter we have two options:

(1) We relax the definition of type context such that it would allow for subtypes, i.e. instead of using a list of types to denote "variable 0 is of type *Nat*", "variable 1 is of type *String*", we somehow relax it such that we can represent "variable 0 is of a type which is a subtype of *Int*".
(2) We introduce a new expression *weaken* of type T which can be constructed by an expression of type S and a proof $S \leq T$.

Existing literature on the first option is scarce and most literature such as [31] opt for the second option. This however would involve modifying the operation semantics to take into account of the *weaken* term, which in turn involves defining new semantics for relaxing expressions to a parent type. This greatly complicates the design of the interpreter and distracts from the main goal of showing subject reduction and session fidelity holds for the formalized MPST, therefore we opted to ignore subtyping.

4.0.3 Process oriented typing. The MPST in *Less is More* [26] defines typing in terms of *channels*. More concretely, it allows for arbitrary nesting of sessions from which channels can be created. Typing contexts mainly concern with the types of the channels and safety properties act on the typing contexts, not the type of the process expression itself. Its typing context is defined as follow:

$$\Gamma ::= \emptyset \mid \Gamma, x : S \mid \Gamma, s[p] : S$$

Here, x refers to a channel variable name, s refers to a nested session name, and p is a participant of the session. Defining this for an intrinsically-typed interpreter is especially difficult since this definition cannot be easily translated to a standard typing context representation for de-Brujin indices, and a more complicated structure must be defined to represent this. Since expression datatypes are indexed by the typing context, having a complex typing context definition would complicate reasoning with expressions.

For simplicity's sake, our MPST does not allow arbitrary nesting of sessions, and therefore all communication would be carried out in one global auxiliary channel. Session types are instead done on the process expressions, so that the representation for the MPST expressions would resemble more to intrinsically-typed STLC expressions, and allow us to borrow standard intrinsically-typed interpreters constructions with less friction.

5 EVALUATION

In this section, we evaluate the interpreter on example protocols. We picked two common protocols: the echo protocol to demonstrate basic expressive power of the calculus, the two-buyer protocol [16] to demonstrate the multiparty aspect of the calculus. We then evaluate the short-coming of the intrinsically-typed interpreter as it is.

5.1 Echo Protocol

We can define the session type of the echo protocol as follows:

```
AlicePOV : RTy
AlicePOV =
  `send Bob ("Ping" ::# []) (λ _ → `str)
  λ _ → `recv Bob ("Pong" ::# []) (λ _ → `str)
  λ _ → `end

BobPOV : RTy
BobPOV =
  `recv Alice ("Ping" ::# []) (λ _ → `str)
  λ _ → `send Alice ("Pong" ::# []) (λ _ → `str)
  λ _ → `end

Session : STy
Session = (Alice , AlicePOV) :: (Bob , BobPOV) :: []
```

`AlicePOV` begins by sending Bob a message of label "Ping" and payload type string, it then receives from Bob a message of label "Pong" and payload type string. Conversely, `BobPOV` begins by receiving from Alice a message of label "Ping" of payload type string, and sends Alice a message of label "Pong" and payload type string. We can compose the two local session types into a full session by creating an associative list of roles and local session types. To implement a protocol according to the session type:

```
AliceProcess : PExpr [] (`proc AlicePOV)
AliceProcess =
  send Bob _ _ _ ("Ping" of here refl) (str "hello!")
  $ recv Bob _ _ _ λ _ → 0

BobProcess : PExpr [] (`proc BobPOV)
BobProcess =
  recv Alice _ _ _
  λ { ("Ping" of here refl) →
    send Alice _ _ _
      ("Pong" of here refl) (var (here refl)) 0
  }

Protocol : SExpr Session
Protocol =
  Alice by _ ◃ AliceProcess |
    (Bob by _ ◃ BobProcess | [])
```

`AliceProcess` begins by sending Bob a message of label "Ping" with the payload "hello!", receives a payload from Bob and terminates. Conversely, `BobProcess` begins by receiving a message from Alice, and in the case that it is of tag "Ping" (which is the only option in this example), it sends Alice a message of tag "Pong" with the content it received from Alice before terminating. Here, `var (here refl)` refers to "the first variable according to de-Brujin indices", which would be the payload content of the message that is received from Alice. We can finally compose the entire protocol by chaining them with `SExpr`.

To demonstrate a chain of reduction steps, we can construct the type of the session after one step of communication:

```
AlicePOV1 : RTy
AlicePOV1 =
  `recv Bob ("Pong" ::# []) (λ _ → `str)
  λ _ → `end

BobPOV1 : RTy
BobPOV1 =
  `send Alice ("Pong" ::# []) (λ _ → `str)
  λ _ → `end
```

```
Session1 : STy
Session1 = (Alice , AlicePOV1) :: (Bob , BobPOV1) :: []

step01 : Session ⇒ Session1
step01 = _
```

In this specific example, Agda manages to type-check and infer the value of `step01`. We can continue this to construct a chain of reduction steps until all protocols reach the end state, demonstrating the state of the protocol at each step.

5.2 Two Buyer Protocol

We can define the session type of the two buyer protocol as follow:

```
AlicePOV : RTy
AlicePOV =
  `send Seller ("product" ::# []) (λ _ → `str)
  λ _ → `recv Seller ("quote" ::# []) (λ _ → `nat)
  λ _ → `send Bob ("propose" ::# []) (λ _ → `nat)
  λ _ → `end

SellerPOV : RTy
SellerPOV =
  `recv Alice ("product" ::# []) (λ _ → `str)
  λ _ → `send Alice ("quote" ::# []) (λ _ → `nat)
  λ _ → `send Bob ("quote" ::# []) (λ _ → `nat)
  λ _ → `recv Bob ("bye" ::# "ok" ::# [])
    (λ _ → `unit)
  λ { ("bye" of here refl) →
      `end
    ; ("ok" of there (here refl)) →
      `recv Bob ("address" ::# []) (λ _ → `str)
      λ _ → `send Bob ("date" ::# []) (λ _ → `str)
      λ _ → `end
    }

BobPOV : RTy
BobPOV =
  `recv Seller ("quote" ::# []) (λ _ → `nat)
  λ _ → `recv Alice ("propose" ::# []) (λ _ → `nat)
  λ _ → `send Seller ("bye" ::# "ok" ::# [])
    (λ _ → `unit)
  λ { ("bye" of here refl) →
      `end
    ; ("ok" of there (here refl)) →
      `send Seller ("address" ::# []) (λ _ → `str)
      λ _ → `recv Seller ("date" ::# []) (λ _ → `str)
      λ _ → `end
    }

Session : STy
Session = (Alice , AlicePOV) :: (Seller , SellerPOV)
  :: (Bob , BobPOV) :: []
```

Here we demonstrate a multiparty session with branching depending on the received message tag. The Two Buyer Protocol involves Alice, Bob, and a Seller in a coordinated interaction to potentially purchase a product. Alice initiates by sending a product name to the Seller, who responds with a price quote to both Alice and Bob. Alice then proposes a contribution amount to Bob. At this point, Bob decides whether to proceed with the purchase. If he declines, he sends "bye" to the Seller, ending the protocol. If he agrees, he sends "ok", followed by his shipping address. The Seller then provides a delivery date, concluding the transaction.

```
AliceProcess : PExpr [] (`proc AlicePOV)
AliceProcess =
  send Seller _ _ _ ("product" of here refl)
    (str "Luxury Coat")
  $ recv Seller _ _ _
  λ { ("quote" of here refl) →
    send Bob _ _ _ ("propose" of here refl) (int 220) 0
  }

SellerProcess : PExpr [] (`proc SellerPOV)
SellerProcess =
  recv Alice _ _ _
  λ { ("product" of here refl) →
    send Alice _ _ _ ("quote" of here refl) (int 350)
    $ send Bob _ _ _ ("quote" of here refl) (int 350)
    $ recv Bob _ _ _
    λ { ("bye" of here refl) → 0
      ; ("ok" of there (here refl)) →
        recv Bob _ _ _
        λ { ("address" of here refl) →
          send Bob _ _ _ ("date" of here refl)
            (str "March 14 2025") 0
        }
    }
  }

BobProcess : PExpr [] (`proc BobPOV)
BobProcess =
  recv Seller _ _ _
  λ { ("quote" of here refl) →
    recv Alice _ _ _
    λ { ("propose" of here refl) →
      send Seller _ _ _ ("ok" of there (here refl)) unit
      $ send Seller _ _ _ ("address" of here refl)
        (str "Atlanta, GA, United States")
      $ recv Seller _ _ _
      λ { ("date" of here refl) → 0 }
    }
  }
```

Alice initiates the transaction by informing the Seller of the desired product "Luxury Coat". The Seller responds with the price $350 to both Alice and Bob. Alice proposes $220 to Bob. At this branch, Bob evaluates the proposal and must decide to either terminate by respond with "bye" or proceed with the purchase by responding with "ok". He decides to proceed with "ok" to initiate the next phase. He shares his shipping details with the Seller, who then finalizes the protocol by providing a delivery date.

Alternatively, Bob can also chooses to terminate:

```
BobProcess' : PExpr [] (`proc BobPOV)
```

```
BobProcess' =
  recv Seller _ _ _
  λ { ("quote" of here refl) →
    recv Alice _ _ _
    λ { ("propose" of here refl) →
      send Seller _ _ _
        ("bye" of here refl) unit 0
    }
  }
```

Both implementation do not deviate from the protocol specification defined in the session types are able to type-check correctly.

5.3 Limitations

To construct a reduction step from a protocol expression to another, one must first show that the expression's type satisfy a safety property φ. This is especially hard to do manually, because one would need to consider all possible state space of the protocol matching the type, and show that reductions from all the state space preserve φ. For a simple protocol such as the echo protocol this is doable, but for more complicated examples such as the two-buyer protocol one would need to manually write out all possible state space, prove that indeed covers all the possible state space, and painstakingly write out all possible reduction steps to demonstrate φ holds under reductions.

Ideally, this is automated such that one can simply write an algorithm to decide if a safety property φ holds without considering structural congruence or if it is preserved under reductions, then a generalized algorithm can use this to decide if φ holds for any given θ. This however goes beyond the scope of this thesis, which is to merely formalize a proof of subject reduction and session fidelity for a simple MPST. As such, this has not been done yet, and our examples are lacking process reduction steps.

6 RELATED WORK

In this section, we survey different existing literature on the session calculi and their mechanization, and list out how these literature might help inspire some design decisions in this thesis.

6.1 Binary session calculi mechanization

There have been several mechanization focus on binary session types. Zalakain and Dardha [38] mechanized an extrinsically-typed, resource-aware π-calculus in Agda, using leftover typing to formalize a type system that allows for not just linear types but also graded types and shared types. While their work is not directly applicable to ours, it helped in understanding how de Bruijn indices would be used for a nameless representation of process calculi in Agda. Niccolò and Vezzosi [34] also formalized π-calculus in Cubical Agda, and while our work is also formalized in Cubical Agda, their work focuses more on the denotational semantics of the π-calculus, and our work focuses more on constructing the operational semantics of a multiparty session calculus.

6.2 Intrinsically-typed interpreter

Thiemann [31] applied the techniques into formalizing intrinsic-typed interpreters for binary session types. Of note is that the interpreter is formalized as an intrinsically typed CEK machine and it uses a global session context to record if each channels are open from which ends. While adapting the global session context to this thesis cannot be directly done due to the problems laid out in section 3, formalizing the intrinsic-typed interpreter as CEK machine is a possible route that this thesis can also take, and his work provides some basic guidelines on how this aims of this thesis can be achieved.

6.3 Mechanization of Multiparty Session Types

There exist very few literature on mechanization of multiparty session types, but there have been work done in this area by Tirore et al. in [32], and Castro-Perez et al. in [7]. The work done by Tirore et al. explicitly opted not to follow the "Less is More" approach and instead focus on formalizing a new computable projection function from global types to local types, citing the reason being that while the "Less is More" approach is indeed more general, but it misses the advantage which is that global types allow you to define a top-to-bottom specification of the protocol. Their formalization of local session types also make use of coinductivity to model non-terminating expressions, but it cannot be directly applied to our work for reasons listed in 4.0.1. Another work in this area is Zooid by [7], which formalizes a Domain Specific Language (DSL) embedded in Coq that allows one to write well-typed communicating processes. Their goal is to address the errors for broken proofs of subject reduction and decidability of subtyping in multiple MPST theory, and formalize a metatheory for MPST in Coq, giving a computable implementation of projection, type-checking, and process validation, with proofs of their correctness. This also differs from our goal of formalizing a MPST theory that side-steps global types and allow for a generalized safety property to prove subject reduction, thus their work is not completely applicable to ours.

7 CONCLUSION

We formalized the syntax and semantics of a generalized multiparty session calculus as an intrinsically-typed interpreter in Agda. Our formalization proves that subject reduction and session fidelity holds for the formalized MPST. This lays the groundwork for future formalizations of MPST in Agda, allowing for the development of extensions of MPST theories with mechanized proofs, giving high confidence that the theories are indeed sound.

Future work can explore the automation of checking whether a common safety properties (e.g. deadlock-freedom) hold for a protocol, and extending the current formalized MPST with recursion so that non-terminating protocols can be expressed and reasoned with.

REFERENCES

[1] Agda Developers. [n. d.]. *Agda*. https://agda.readthedocs.io/
[2] Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. In *Workshop on Dependent Types in Programming, Gothenburg*.
[3] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2017. Intrinsically-typed definitional interpreters for imperative languages. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.
[4] Adam D Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. 2023. Designing asynchronous multiparty protocols with crash-stop failures. *arXiv preprint arXiv:2305.06238* (2023).
[5] Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. 2012. Strongly typed term representations in Coq. *Journal of automated reasoning* 49, 2 (2012), 141–159.
[6] Edwin Brady. 2021. Idris 2: Quantitative type theory in practice. *arXiv preprint arXiv:2104.00480* (2021).
[7] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. [n. d.]. Zooid: a DSL for Certified Multiparty Computation. ([n. d.]).

[8] Joëlle Despeyroux. 2000. A higher-order specification of the π-calculus. In *IFIP International Conference on Theoretical Computer Science*. Springer, 425–439.

[9] Simon J Gay. 2001. A framework for the formalisation of pi calculus type systems in Isabelle/HOL. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 217–232.

[10] Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. 2019. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Algebraic Methods in Programming* 104 (2019), 127–173.

[11] Go Authors. 2025. The Go Programming Language. https://go.dev/

[12] Georges Gonthier et al. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393.

[13] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. 2017. A formal proof of the Kepler conjecture. In *Forum of mathematics, Pi*, Vol. 5. Cambridge University Press, e2.

[14] Loïc Henry-Gréard. 1999. *Proof of the subject reduction property for a pi-calculus in Coq*. Ph. D. Dissertation. INRIA.

[15] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*. Springer, 122–138.

[16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 273–284.

[17] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[18] Ivan Lanese, Ugo Dal Lago, and Vikraman Choudhury. 2024. Towards quantum multiparty session types. In *International Conference on Software Engineering and Formal Methods*. Springer, 385–403.

[19] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[20] Per Martin-Löf. 2021. Intuitionistic Type Theory: Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980. (2021).

[21] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and computation* 100, 1 (1992), 1–40.

[22] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer, 625–635.

[23] Stefan Müller-Stach. 2017. *Richard Dedekind: Was sind und was sollen die Zahlen? Stetigkeit und Irrationale Zahlen*. Springer.

[24] The Univalent Foundations Program. 2013. Homotopy type theory: Univalent foundations of mathematics. *arXiv preprint arXiv:1308.0729* (2013).

[25] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 284–298.

[26] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[27] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An interaction-based language and its typing system. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 398–413.

[28] The Agda Community. 2024. *Cubical - Indexed inductive types*. https://agda.readthedocs.io/en/latest/language/cubical.html#indexed-inductive-types

[29] The Agda Community. 2024. *Cubical Agda Library*. https://github.com/agda/cubical

[30] The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

[31] Peter Thiemann. 2019. Intrinsically-typed mechanized semantics for session types. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. 1–15.

[32] Dawit Tirore, Jesper Bengtson, and Marco Carbone. 2023. A sound and complete projection for global types. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[33] Vasco T Vasconcelos. 2012. Fundamentals of session types. *Information and Computation* 217 (2012), 52–70.

[34] Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing pi-calculus in guarded cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 270–283.

[35] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.* 3, ICFP, Article 87 (July 2019), 29 pages. https://doi.org/10.1145/3341691

[36] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*. https://plfa.inf.ed.ac.uk/22.08/

[37] Nobuko Yoshida and Lorenzo Gheri. 2019. A very gentle introduction to multiparty session types. In *International Conference on Distributed Computing and Internet Technology*. Springer, 73–93.

[38] Uma Zalakain and Ornela Dardha. 2021. π with Leftovers: A Mechanisation in Agda. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 157–174.