A Mechanisation of Multiparty Session Types

IT University of Copenhagen

Dawit Tirore

October 2024

Abstract

In the modern age of computing, distributed systems are ubiquitous. Besides the positive aspects of distributed systems, such as their scalability, they remain hard to implement due to the intricacies of parallel execution. A type discipline known as session types offers an appealing theoretical foundation for verifying the correctness of distributed systems. Session types are increasingly used in practice, seen by its integration into many mainstream programming languages and the emergence of standalone session type based tools, making it increasingly important to verify its theoretical foundations. Guarantees offered by session types rely on formal proofs, and errors in these proofs have been uncovered in multiple heavily cited session type papers, and this has motivated the use of proof assistants to verify results within the field. Mechanised results exist for the restricted formalism called binary session types, supporting the verification of two communicating roles. Mechanised results for the more expressive multiparty session types, allowing an arbitrary number of roles, is however far more limited. In particular, the claims of the original paper by Honda et al, who introduced multiparty session types more than a decade ago, remain unverified.

This thesis addresses this problem by providing a faithful mechanisation of results from Honda et al.. The contributions are twofold. Our first contribution is a novel definition of the critical procedure of multiparty session types known as projection, extending on prior results by proving our definition not only sound, as others have for other formulations, but also complete, doing so against a coinductive specification of projection. Our second contribution is that we identify a counterexample to the subject reduction theorem of Honda et al., and we define a new type system that addresses this counterexample, and mechanise a subject reduction theorem for this type system. Designed with decidability in mind, we provide decision procedures for several of the properties the type system relies on.

Resumé

Brugen af distribuerede systemer rækker vidt og bredt. På trods af de positive aspekter af distribuerede systemer, så som deres evne til at skalere, forbliver de svære at implementere grundet udfordringerne bag parallel programeksekvering. En type disciplin, kendt som session types, tilbyder et tiltrækkende teoretisk fundament til at sikre korrektheden af distribuerede systemer. Session types er i stigende grad blevet brugt i praksis, hvilket gør det vigtigt at sikre korrektheden af det teoretiske fundament session types er baseret på. Garantier som man opnår ved brug af session types afhænger af formelle beviser, og fejl i disse beviser er fundet i flere velciterede artikler om session types, og dette har motiveret brugen af bevis assistenter til at producere beviser der mekanisk kan blive checket for korrekthed af en computer. Mekaniserede resultater eksisterer for den begrænsede formalise, binary session types, som understøtter kommunikation blandt to deltagere. Mekaniserede resultater for den mere udtryksfulde multiparty session types, som tillader arbitrært mange deltagere, er til gengæld mere begrænset. Værd at bemærke er at resultaterne fra den originale artikel af Honda et al., som introducerede multiparty session types for mere end et årti siden, til den dag i dag ikke er blevet verificeret.

Denne afhandling takler dette problem ved at mekanisere resultater fra Honda et al., og i den sammenhæng præsenterer vi to overordnede videnskabelige bidrag. Vi præsenterer en ny definition af en central operation i multiparty session types, som kaldes projection, hvor vi forbedrer eksisterende resultater ved at bevise vores definition både sund, som andre har gjort, og komplet. Vores andet bidrag er identificeringen af et modeksempel til Subject Reduction teoremet af Honda et al., og et nyt type system der takler dette modeksempel, som vi producerer et maskine-checket bevis for opfylder Subject Reduction. Designet med decidability i mente, definerer vi algoritmer til at afgøre flere egenskaber som type systemet afhænger af.

Acknowledgements

First and foremost I would like to thank my supervisors Marco Carbone and Jesper Bengtson. I am grateful for your guidance and support, which has helped me become a better researcher. In the periods where developing Coq proofs became tiring and overwhelming, the two of you provided stimulating and fun white-board problem solving sessions, and I appreciate that this was interleaved with more personal digressions beyond work. I am grateful to have had you as my supervisors.

I would also like to thank my colleagues at IT University of Copenhagen for creating a lovely research environment. In particular, I would like to thank my office mates Maaike Annebet Zwart, who makes the most delicious baked goods, and Maryam Sheikhi Garjan, with whom I had many conversations about dietary hacks and carbohydrate restriction. I see the irony. Among my colleagues, a special thanks goes out to Anastassia Vybornova, my PhD twin who began her enrollment on the same day as me. Among other topics, we discussed the balance between work and leisure, a balance we both found in the canals of Christianshavn during our winter bathing sessions. I also thank my colleague, and Brazilian jiu-jitsu buddy, David Sasu, who I initially lent my charger when we first met at a writing seminar, and later my neck, when we practiced the effective submission technique that is the rear naked choke.

I thank the members of the committee Alessandro Bruni, Simon Gay and Marino Miculan for reviewing this thesis.

I would also like to thank Fritz Henglein who introduced me to the Coq proof assistant during my master's, and with whom I have pursued exciting projects on the mechanisation of regular expressions.

I extend my gratitude to Nobuko Yoshida for hosting me during my research stay in Oxford, and I thank her research group for welcoming me. I enjoyed our symposiums and I am happy that we managed to squeeze in an dinner at the local Ethiopian restaurant the day before my stay ended. I would also like to thank David Kim, a landlord turned friend, with whom I shared a love for spicy stuff, and enjoyed many engaging evening conversations. Speaking of evening conversations, I also want to thank my childhood friend Jonas Rask Christensen, whose frequent phone calls was an appreciated constant during my stay in Oxford.

Finally I would like to thank my family. I thank my sisters Sarah and Rebecca for being loving and wonderful people who supported me through these three years. I also thank my parents Legesse and Bezunesh. You have been along this ride from the start. Thank you for your love and support.

Contents

Chapter 1

Introduction

The modern age of computing is distributed. The Internet has made it possible to coordinate many machines, yielding a joint capability for computation far beyond what is possible by any of its individual components. Such a collection of independent machines that coordinate the execution of a joint task by message passing, is called a distributed system. Examples of distributed systems are cloud computing services like Amazon Web Services [1] and Google Cloud Platform [3], that provide server clusters with a computational power far exceeding any single processor on the market. Similarly, distributed databases such as MongoDB [5] achieve an enormous storage capacity by locating data across multiple devices. These distributed systems harness the power of multiple independently executing machines. With the performance of a distributed system growing by each additional machine added, distributed systems scale. Distributed systems also offer robustness, since critical functionality may be replicated by several components, preventing a system wide failure in case a single component fails.

Distributed systems have many upsides, but they are hard to get correct. Reasons include network reliability (messages could be dropped) and security concerns (components may act maliciously). A central challenge is however the inherent nondeterminism of distributed systems that arises when independent machines execute in parallel. This affects the order of the messages exchanged, complicating debugging when bugs occur only in some, and not all executions [41]. The challenges of debugging concurrency bugs have been studied in the setting of large software applications [52] (e.g MySQL [7] and Mozilla Web Browser [6]). Here it was found that the initial attempts at fixing many of these concurrency-related bugs were incorrect,

thus remaining hard to solve even after being identified. An example of a concurrency bug is a deadlock where program execution is unintentionally blocked and unable to progress. Another concurrency bug is a race, where multiple concurrent executions compete for access to a shared resource. The consequences of such bugs can be costly. It was estimated by the insurance marketplace Lloyd's that the failure of a large cloud service provider could mean the loss of millions of dollars [4].

In recognition of these challenges, several modern programming languages offer various features that aim at minimising such bugs. The Go programming language has goroutines which are threads that can communicate by message passing, rather than shared memory which can be more error prone [73]. The Rust programming language has a type system which tracks ownership of data, which ensures safe memory access in concurrent programs. A type system defines a set of rules for how programs may be defined and is an efficient approach to ruling out a large class of bugs. Beyond Rust, type systems in general offer an efficient and scalable way to perform static analysis of programs [64]. By constraining the programs one may execute to those that "do not go wrong" [56], many bugs can be prevented. The properties of a well-typed program depends on the type discipline in question. The most commonly used type discipline in many programming languages is simple types [20], ensuring that one does not write nonsensical expressions like `"foo" + 5`. Linear types [75] ensure not only the absence of the nonsensical expression above, but also track how resource sensitive values are used. More precisely, a linear type system ensures that linear resources are used exactly once. The ownership tracking performed in the Rust type system is an example of a linear type system. Even more rigorous properties about well-typed programs can be ensured with *behavioral types* [9]. They act as a specification for a program, describing the sequence of operations that will be performed.

Session Types. The topic of this thesis is a particular behavioral type discipline which has seen much success in both theory and practice known as *session types* [38, 80, 39, 40], which we investigate to support the future development of safe distributed systems. At the heart of session types is the modular and compositional idea of a *session*, which should be thought of as an execution instance of a protocol. An example is the SSH protocol where keys are exchanged to establish a secure connection between a client and a server. The server may be participating in more than one session,

alternating between serving multiple clients and thus alternating between communicating on many sessions. A protocol is specified with session types, and these types define a sequence of, possibly repeating, communication actions that protocol participants must perform. Properties offered by session types include that communication behaves as specified by session types (session fidelity), the system is never stuck (progress) and received messages are of the expected type so that nonsensical calculations do not occur mid-execution (type safety).

The Problem. There exist many external tools based on session types [79, 81, 46, 48, 49] and programming languages with session type features such as C [60], Erlang [28], Python [58] and many more [8]. These tools and programming language features aid the development of safe distributed systems by relying on guarantees offered by session types. These guarantees in turn rely on formal proofs, and some of these proofs have been found to be incorrect [80, 67]. The original and more restrictive formulation of *binary* session types [38] (supporting only two components in a session), has limitations in their semantics that have been known of for a while, providing the necessary time to address these limitations on paper [80] and verify their correctness [19] with tools like proof assistants that allow a high degree of trust by producing machine-checked proofs. The more recent and expressive variant known as *multiparty* session types (supporting two or more components in a session) shares similar soundness issues that have been discovered more recently [67]. Efforts have been made to produce machine-checked proofs about multiparty session types in novel settings [18, 44], but machine-checked proofs of the original results by Honda et al. [39, 40], who introduced multiparty session types more than a decade ago, and laid the foundation for all later multiparty session types results, remains an open problem.

Aim and Methodology. The soundness issues of multiparty session types casts doubts on the foundations of the formalism, and demonstrates the importance of accompanying formal results on paper with machine-checked proofs. The aim of this PhD project is:

To increase reliability and trust in tools and programming languages that rely on multiparty session types for the development of safe distributed systems.

The reason why we cast doubts on the correctness of the results in Honda

et al. is because their results rely on proofs by hand, and experience has shown us that this approach is fragile. To avoid having our results subjugated to these same doubts in the future, we take the methodology of using the Coq proof assistant [54] to produce machine-checked proofs of all our results. Proof assistants are software tools that aid the user in writing correct proofs, interactively showing the user where in the proof they are, and mechanically verifying the correctness of each reasoning step. Commonly used proof assistants include Coq [54], Isabelle/HOL [61], Lean [55] and HOL Light [36]. Some of the most impressive and large-scale developments in proof assistants in mathematics include proofs of the Kepler Conjecture [35] and Odd Order Theorem [33]. Comparative results in computer science include the verified C compiler CompCert [50] and verified seL4 microkernel [45].

Through the use of the Coq proof assistant, we attempt to answer the research question:

Is it possible to produce a machine-checked formalisation of multiparty session types that is faithful to the original formulation?

Chapter 2

Session Types

Session types were originally introduced by Takeuchi et al. [69] and later adapted by Honda et al. [38] to a model of distributed systems known as the π-calculus [57], extending this calculus with the notion of session. A session is an execution instance of a protocol and allows a simple specification of the communication pattern in terms of a session type. We illustrate sessions by an example. Below two processes are about to initiate a session on the shared name a:

$$a(x_0).\overbrace{x_0!\langle 5\rangle; x_0?(y_0); \mathbf{0}}^{Q} \mid \overline{a}(x_1).\overbrace{x_1?(y_1); x_1!\langle y_1 + 5\rangle; \mathbf{0}}^{R}$$

The example consists of two sub-processes, $a(x).Q$ and $\overline{a}(x).R$, separated by the vertical line ($|$) which denotes parallel composition, expressing that they execute in parallel. The R process is attempting the initiation of a session on a as the requesting party, denoted by $\overline{a}(x_1)$, and the Q process is attempting the initiation as the accepting party, denoted by $a(x_0)$. Here, (x_0) and (x_1) act as binders used by the processes to implement their communication in the session. The communication that Q will perform in this session is to initially send the integer 5 on its session channel, followed by receiving a value on this channel. The sending operation is denoted by $x_0!\langle 5\rangle$, and the receiving operation is denoted by $x_0?(y_0)$, where (y_0) acts as a binder for the received value. The communication performed by R in the session is to receive a value on its channel, denoted by $x_1?(y_1)$ where the received values is bound to y_1. The y_1 value is then incremented by 5 and sent on the channel, denoted by $x_1!\langle y_1 + 5\rangle$. After these communications both processes are done, which is denoted by $\mathbf{0}$.

2.1 Binary Session Types

A session is initiated on a shared name, and the communication that takes place in a session is specified by the session types associated with this name. In binary session types, a session consists of two parties, and a shared name is therefore associated with two session types. Before we look at the session types associated to the shared name a in our example, we inspect the behavior of Q and R and consider why it is safe. Q sends and receives and integer, while R receives and then sends an integer. R additionally performs the addition $y_1 + 5$. This seems safe in the sense that the session will not deadlock, nor will the addition $y_1 + 5$ result in a type error, since we know y_1 will be substituted for an integer. A type system for binary session types makes this intuition formal, associating the shared name a with the following session types for the requesting (right) and accepting (left) parties:

$$!\langle \mathsf{int} \rangle; ?\langle \mathsf{int} \rangle \qquad ?\langle \mathsf{int} \rangle; !\langle \mathsf{int} \rangle$$

The types describe the communication of Q (left) and R (right), specifying by $(!\langle \mathsf{int} \rangle)$ the sending of an integer and by $(?\langle \mathsf{int} \rangle)$ the reception of an integer, separating these actions by semicolon $(;)$ to indicate their order.

Duality and Typing. The session types of R and Q are complimentary in the sense that when one sends the other receives and vice versa. This property is known as *duality*, which makes precise our intuition for why the interactions between Q and R are safe. In Honda et al. [38] duality is defined as an operation. For a session type T, its dual is computed as $\overline{T}$. Duality holds between two session types T and T' when $\overline{T} = T'$. Duality of the sessions types for Q and R is witnessed by satisfying this equality:

$$\overline{!\langle \mathsf{int} \rangle; ?\langle \mathsf{int} \rangle} = ?\langle \mathsf{int} \rangle; !\langle \mathsf{int} \rangle$$

Duality is used in the type system of binary session types. The typing judgment is $\Gamma \vdash P \triangleright \Delta$ where Γ is an unrestricted environment and Δ is a linear environment. Shared names are typed by Γ using duality in a way than can be seen in the typing rules for session request and accept:

$$\frac{\Gamma \vdash a \triangleright \langle T, \overline{T} \rangle \quad \Gamma \vdash P \triangleright \Delta, x : \overline{T}}{\Gamma \vdash \overline{a}(x).P \triangleright \Delta} \text{ [Req]}$$

$$\frac{\Gamma \vdash a \triangleright \langle T, \overline{T} \rangle \quad \Gamma \vdash P \triangleright \Delta, x : T}{\Gamma \vdash a(x).P \triangleright \Delta} \text{ [Acc]}$$

Both rules carry the premise $\Gamma \vdash a \triangleright \langle T, \overline{T} \rangle$, associating to a the two dual session types T and $\overline{T}$. In [Req] the requesting party has its linear environment extended with session channel x of type $\overline{T}$, while x is extended with T in [Acc]. Session channels are typed by Δ in a way that also uses duality, and to see how we observe the shape of Q and R after they have initiated a session:

$$(\boldsymbol{\nu} k)(\overbrace{k^+!\langle 5\rangle; k^+?(y_0); \mathbf{0}}^{Q[k^+/x_0]} \mid \overbrace{k^-?(y_1); k^-!\langle y_1\rangle; \mathbf{0}}^{R[k^-/x_1]})$$

The initiation has created a session restriction, denoted by $(\boldsymbol{\nu} k)$, which binds to channel endpoints k^+ and k^-, used by Q and R respectively. Duality can be seen in the typing rule for session restriction:

$$\frac{\Gamma \vdash P \triangleright \Delta, k^+ : T, k^- : \overline{T}}{\Gamma \vdash (\boldsymbol{\nu} k)P \triangleright \Delta} \text{ [CRes]}$$

The linear environment in the premise is extended with dual session types for the session channels k^+ and k^-. These session channels must be used linearly, which means that a session channel should be used by exactly one party. This is enforced by the typing rule for parallel composition:

$$\frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta, \Delta'} \text{ [Par]}$$

Here, the Δ part of the linear environment is used to type P and the Δ' part is used to type Q, and because these environments are disjoint it is ensured that a session channel never is used by multiple parties at the same time.

Subject Reduction. We say Δ is *balanced* whenever the joint occurrence of $k^+ : T \in \Delta$ and $k^- : T' \in \Delta$ implies duality of the session types (e.g. $\overline{T} = T'$). Balancedness thus extends duality to an entire environment, and this is used to state subject reduction. This property states that a well-typed process remains well-typed after performing some communications, and we denote the execution of zero or more communications as $P \longrightarrow^* P'$. The formal statement that is proved by Yoshida and Vasconcelos [80] can be seen below:

If $\Gamma \vdash P \triangleright \Delta$ with balanced Δ and $P \longrightarrow^* P'$, then there exists Δ' such that $\Gamma \vdash P' \triangleright \Delta'$ and balanced Δ'

They use this property to prove type safety which states that well typed programs do not reduce to errors (e.g expecting int but receiving string).

2.2 Multiparty Asynchronous Session Types

Multiparty session types were introduced by Honda et al. [39, 40]. They generalise binary sessions to multiparty sessions that may consist of arbitrarily many roles, a significant contribution that earned the authors the Most Influential Paper POPL 2018 award [53].

The Two-Buyer-Protocol. We introduce multiparty session types with the classical example by Honda et al. known as the two buyers protocol. The protocol involves two buyers, B1 and B2, along with a seller S. The buyers want to purchase an expensive item together from S, and coordinate among themselves if they can split the price of the item. More precisely the scenario is: B1 requests the price of an item, and S responds with a price to both buyers; B1 then notifies B2 of how much they will contribute and B2 sends a label to S accepting or declining the offer. If the offer is accepted, B2 sends an address to S for shipping. If the offer is declined, communication halts. This protocol is seen below:

$$
\begin{aligned}
&\mathsf{B1} \to \mathsf{S} : \langle \mathsf{string} \rangle.\\
&\mathsf{S} \to \mathsf{B1} : \langle \mathsf{int} \rangle.\\
&\mathsf{S} \to \mathsf{B2} : \langle \mathsf{int} \rangle.\\
&\mathsf{B1} \to \mathsf{B2} : \langle \mathsf{int} \rangle.\\
&\mathsf{B2} \to \mathsf{S} : \left\{ \begin{array}{ll} \mathtt{Ok}: & \mathsf{B2} \to \mathsf{S} : \langle \mathsf{string} \rangle.\ \mathsf{S} \to \mathsf{B2} : \langle \mathsf{date} \rangle.\mathsf{end} \\ \mathtt{Quit}: & \mathsf{end} \end{array} \right\}
\end{aligned}
$$

The object above is a global type which is a declarative top-level protocol that specifies a multiparty session in terms of interactions. One type of interaction is the communication of a message, such as $\mathsf{B1} \to \mathsf{S} : \langle \mathtt{string} \rangle$ which denotes that a string is sent by B1 and received by S. The other kind of interaction is a branching, such as $\mathsf{B2} \to \mathsf{S} : \left\{ \begin{array}{l} \mathtt{Ok}: \ldots \\ \mathtt{Quit}: \ldots \end{array} \right\}$ which specifies that B2 will send either label `Ok` or `Quit` to S. Specifying a multiparty session in terms of these interactions allows a more precise description of

communication than is possible with binary session types. Consider how the behavior of B1 may be described with binary session types. This role interacts with B2 and S, and therefore needs to be part of two different sessions. The communication that B1 performs in the session with B2 (left) and the session with S (right) can be described by the types below:

$$!\langle \mathsf{int} \rangle; \mathsf{end} \qquad !\langle \mathsf{string} \rangle; ?\langle \mathsf{int} \rangle; \mathsf{end}$$

There are multiple ways B1 may alternate between its participation in the two sessions. The type system allows any interleaving of the actions in the two session types above. The global type on the other hand specifies a specific interleaving of the actions that B1 should perform in its interaction with B2 and S. Consequently, global types enforce that the first action of B1 is $!\langle \mathsf{string} \rangle$, while binary session types also allow $!\langle \mathsf{int} \rangle$ as the first action.

Global types. Global types are μ-types which allows specifications of message interactions and branching interactions to be tail-recursive. The full syntax of global types can be seen below:

Definition 2.1 (Global type syntax)

$$G \;::=\; \mathsf{p} \to \mathsf{q} : k\langle U \rangle.G \mid \mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j \in J} \mid \mu \mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end}$$

Here, $\mathsf{p} \to \mathsf{q} : k\langle U \rangle.G$ specifies the interaction of a message type U which p sends and q receives, after which the session continues as G. The branching interaction, $\mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j \in J}$, specifies that p will send a label $j \in J$ to q, after which the session continues as G_j. The terminated session is specified by end. Finally, the tail-recursive protocol $\mu \mathbf{t}.G$ expresses repeating behavior by occurrences of $\mathbf{t}$ in G. In the semantics of global types $\mu \mathbf{t}.G$ is unfolded to $G[\mu \mathbf{t}.G]$. The presence of the μ-binder gives rise to standard notions of wellformedness for μ-types. They are wellformed if they are contractive and closed. A contractive global type always separates the binder $\mu \mathbf{t}$ and the variable $\mathbf{t}$ by at least one interaction $\mathsf{p} \to \mathsf{q}$. For example the global type $\mu \mathbf{t}.\mathbf{t}$ is not contractive. A μ-type is closed if it has no free variables.

Linearity. The syntax of global types above uses explicit channels[1]. An explicit channel is the k in $\mathsf{p} \to \mathsf{q} : k$ and it specifies the queue that will be used for asynchronous communication. In Honda et al., a multiparty

[1]For simplicity the explicit channels in the two-buyer protocol are hidden.

session contains a collection of shared queues, that any role may send to and receive from with explicit channels. Because queues are shared, it is possible write global types that introduce races between roles. An example is:

$$\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int} \rangle.\ \mathsf{r} \to \mathsf{s} : k\langle \mathsf{int} \rangle.\mathsf{end}$$

Here, the message of p may accidentally be intercepted by s. These racy global types are avoided by checking that a global type has a property called *linearity*[2]. This intuitively asserts that any two interactions in a global type with the same explicit channel k, (e.g. $\mathsf{p} \to \mathsf{q} : k$ and $\mathsf{r} \to \mathsf{s} : k$), where one interaction occurs after the other, a causal dependency can be inferred that guarantees that $\mathsf{p} \to \mathsf{q} : k$ will complete before $\mathsf{r} \to \mathsf{s} : k$. Most later work use *implicit*, rather than explicit channels. Multiparty sessions with asynchronous communication based on implicit channels does not share queues between roles. Instead, each of the n roles in a multiparty session have $n-1$ queues used to receive messages from the remaining $n-1$ roles. This is a simpler setting because global types are not racy by construction, but it is also a more restrictive setting explicit channels can model implicit channels while the contrary is not true. Syntactically, global types with implicit channels are written as seen in the two buyers protocol above.

Local types. Multiparty session types uses global types to specify multiparty sessions declaratively from a top-level view, in a way that mentions all roles at once. On the level of the individual rule, multiparty session types use local types to specify the sequences of actions performed by one role that participates in the session. The syntax of local types can be seen below:

Definition 2.2 (Local Types)

$$T \ ::= \ !k\langle U \rangle.T \mid ?k\langle U \rangle.T \mid k \oplus \{l_i : T_i\}_{i \in I} \mid k \,\&\, \{l_i : T_i\}_{i \in I} \mid \mu \mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end}$$

Local types replace the message and branching interactions of global types with local sending and receiving actions on explicit channels. Here, $!k\langle U \rangle.T$ specifies the sending of a value of type U on channel k, while $?k\langle U \rangle.T$ specifies the reception of a value. Local types differ from binary session types because of the explicit channel. Had we used implicit channels, the explicit channel would be replaced by a role, highlighting that local types are more

[2]The exact definition can be found in Honda et al. [40] Section 3.5

expressive than binary session types because they specify where a message is sent. Sending a label $j \in J$ is denoted by $k \oplus \{l_i : T_i\}_{i \in I}$, and awaiting the reception of a label in J is denoted by $k \,\&\, \{l_i : T_i\}_{i \in I}$. Like global types, local types are μ-types with the same notions of contractiveness, closedness and unfolding.

Projection. Projection is an operation on global types, denoted by $G \upharpoonright_{\mathsf{p}}$, that is a partial function that attempts to capture the specification of p in G as a local type. Projection is undefined when the global type specifies behavior for p that is impossible to implement. The definition of projection by Honda et al. can be seen below:

Definition 2.3 (Projection) *The projection of G onto p, written $G \upharpoonright_{\mathsf{p}}$ [18] is defined such that:*

$$(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle.G) \upharpoonright_{\mathsf{p}} = \begin{cases} !k\langle U \rangle.(G \upharpoonright_{\mathsf{p}}) & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ ?k\langle U \rangle.(G \upharpoonright_{\mathsf{p}}) & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ G \upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

$$(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j \in J}) \upharpoonright_{\mathsf{p}} = \begin{cases} k \oplus \{l_j : (G_j \upharpoonright_{\mathsf{p}})\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ k \,\&\, \{l_j : (G_j \upharpoonright_{\mathsf{p}})\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ (G_1 \upharpoonright_{\mathsf{p}}) & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \text{ and } \forall i, j \in J.\ G_i \upharpoonright_{\mathsf{p}} = G_j \upharpoonright_{\mathsf{p}} \\ \bot & \text{otherwise} \end{cases}$$

$$(\mu \mathbf{t}.G) \upharpoonright_{\mathsf{p}} = \begin{cases} \mu \mathbf{t}.(G \upharpoonright_{\mathsf{p}}) & \text{if } G \upharpoonright_{\mathsf{p}} \neq \mathsf{end} \\ \mathsf{end}^{\mu} & \text{otherwise} \end{cases} \qquad \mathbf{t} \upharpoonright_{\mathsf{p}} = \mathbf{t} \qquad \mathsf{end}^{\mu} \upharpoonright_{\mathsf{p}} = \mathsf{end}^{\mu}.$$

Intuitively, the projection on p removes the parts of the specification that are about other roles than p, and captures everything specified about p as a local type. Consider the projection of a message interaction $\mathsf{p} \to \mathsf{q} : k\langle U \rangle.G$. If the projected role is the sender, then we produce $!k\langle U \rangle.T$; and we produce $?k\langle U \rangle.T$ if the projected role is the receiver. If it is neither, then $\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle$ is simply deleted and projection continues on G. The projection of the branching interaction is mostly similar. If the projected role is the sender, we produce $k \oplus \{l_i : T_i\}_{i \in I}$; if it is the sender then $k \,\&\, \{l_i : T_i\}_{i \in I}$ is produced. In the case that the projected role is neither, then it is checked that the projection on all branches are equal. This check is called the branching condition, and it checks whether it is possible to implement the

projected role. A choice made by p_1 is communicated only to p_2, thus, the specification for our projected role should not depend on this choice. This is ensured by the branching condition. There are two cases for the projection of $\mu\mathbf{t}.G$. If the projection of G is not equal to end, projection returns $\mu\mathbf{t}.(G\upharpoonright_{\mathsf{p}})$, and otherwise end is returned. The idea is that the side condition checks if the role occurs in G. If the role does not occur, then it makes sense to return end. The condition is however not sufficient because $(G\upharpoonright_{\mathsf{p}})$ might be a variable $\mathbf{t}$. Consider the example:

$$\mu\mathbf{t}.\mathsf{p} \to \mathsf{q} : \langle \mathsf{int} \rangle.\mathbf{t} \upharpoonright_{\mathsf{r}} = \mu\mathbf{t}.\mathbf{t}$$

The projected role does not occur in the global type, and unfortunately projection produces the local type $\mu\mathbf{t}.\mathbf{t}$ which is not contractive. This is incorrect, and the correct result should be end because the role does not occur in the global type. We go into greater details about incorrectness of recursion conditions in Chapter 4.

A corrected definition of projection is presented by Castro-Perez et al.[18]. They replace the faulty side condition with another condition denoted by $\mathsf{gVar}(\mathbf{t}, G)$ which checks if all occurences of $\mathbf{t}$ happen after an interaction, ensuring that non-contractive types are never produced. The operation is defined the following way:

$$\mathsf{gVar}(\mathbf{t}, G) = \begin{cases} \mathsf{gVar}(\mathbf{t}, G_1) & \text{if } G = \mu\mathbf{t}'.G_1 \\ \mathbf{t} \neq \mathbf{t}' & \text{if } G = \mathbf{t}' \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Projection of the μ-binder is then defined the following way:

$$(\mu\mathbf{t}.G)\upharpoonright_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G\upharpoonright_{\mathsf{p}}) & \text{if } \mathsf{gVar}(\mathbf{t}, G\upharpoonright_{\mathsf{p}}) \\ \mathsf{end} & \text{otherwise} \end{cases}$$

Semantics. The central property of projection stated in Honda et al. is their Lemma 5.11, showing a correspondence between the semantics of global types, and local types obtained through projection of global types. This property is known as Endpoint Projection Theorem, and they prove it using the definition of projection seen in Definition 2.3 above. Their semantics are given as labeled transition systems (LTS), denoted respectively by $G \xrightarrow{\ell} G'$ for global types and $T \xrightarrow{\zeta} T'$ for local types. Note that the labels

for global type reductions ℓ are different than for local type reductions ζ. This is because the former represents an interaction between two roles (e.g $\mathsf{p} \to \mathsf{q} : k\langle U \rangle$) and the latter represents an action by a single role (e.g. $k!\langle U \rangle$). A map from roles to local types is denoted by Δ, which we will also call a local type environment, and its LTS $\Delta \xrightarrow{\ell} \Delta'$ is defined by the single rule seen below, where $\mathcal{U}$ ranges over values U and labels l:

$$\frac{T_1 \xrightarrow{!k\langle \mathcal{U} \rangle} T_1' \qquad T_2 \xrightarrow{?k\langle \mathcal{U} \rangle} T_2'}{\Delta,\ \mathsf{p} : T_1,\ \mathsf{q} : T_2 \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle \mathcal{U} \rangle} \Delta,\ \mathsf{p} : T_1',\ \mathsf{q} : T_2'} \text{[LEnv]}$$

The rule states that the local type environment Δ can reduce if it contains two local types that can perform complimentary send and receive actions. Endpoint Projection Theorem shows the following equivalence between $G \xrightarrow{l} G'$ and $\Delta \xrightarrow{l} \Delta'$:

$$\text{If } G \text{ is coherent then } G \xrightarrow{l} G' \text{ iff } [\![G]\!] \xrightarrow{l} [\![G']\!]$$

Here, coherence of a global type means it is both linear and projectable, and the map denoted by $[\![G]\!]$ is defined as the following:

$$\{\mathsf{p} : G\upharpoonright_{\mathsf{p}} \mid \mathsf{p} \in G\}$$

That is $[\![G]\!]$ denotes a map from the roles of G to their projections. The theorem states that if a global type can reduce by label ℓ, then so can $[\![G]\!]$. Conversely, if the set $[\![G]\!]$ can reduce by ℓ, then so can G. To gain an intuition for the semantics we include some of the rules for global and local types. Consider these rules of the judgment $G \xrightarrow{\ell} G'$:

$$\frac{}{\mathsf{p} \to \mathsf{q} : k\langle U \rangle.G \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle U \rangle} G} \text{[GR1]} \qquad \frac{G[\mu \mathbf{t}.G/\mathbf{t}] \xrightarrow{\ell} G'}{\mu \mathbf{t}.G \xrightarrow{\ell} G'} \text{[GR5]}$$

Rule [GR1] states that a global type may reduce by its initial interaction and rule [GR5] unfolds the μ-binder when it occurs at top-level. Consider next the following rules for the semantics of local types $T \xrightarrow{\zeta} T'$:

$$\frac{}{!k\langle U \rangle.T \xrightarrow{!k\langle U \rangle} T} \text{[LR1]} \qquad \frac{}{?k\langle U \rangle.T \xrightarrow{?k\langle U \rangle} T} \text{[LR2]} \qquad \frac{T[\mu \mathbf{t}.T/\mathbf{t}] \xrightarrow{\zeta} T'}{\mu \mathbf{t}.T \xrightarrow{\zeta} T'} \text{[LR7]}$$

Like the semantics of global types, it states that a local type may reduce by its initial action, reducing with a send by rule [LR1] and a receive by rule [LR2]. Similarly, rule [LR7] unfolds the μ-binder. Only synchronous communication rules have been shown, but the semantics of global and local types are asynchronous and therefore, allowing for example a message of type U' in $!k\langle U\rangle.!k'\langle U'\rangle.\mathsf{end}$ to be received before U.

Remark 2.1 *We have stated the semantics of local types and local type environments in alignment with the definition we use in Appendix C. This differs from the definition of Honda et al. [39, 40] which is based on permutations of the communication actions in local types. Their permutation rules, denoted by the judgment $T \approx T'$, expressing that T may be permuted to T', are missing some cases. As an example, they are missing the case where a receive action may be permuted in front of a sending action on a distinct channel:*

$$!k\langle U\rangle;?k'\langle U'\rangle;T \approx ?k'\langle U'\rangle;!k\langle U\rangle;T \qquad (k \neq k')$$

This makes their Lemma 5.11 incorrect, but this seems to be simply addressed by adding the missing cases. In Chapter 4 we report on our result of proving a correspondence between $G \xrightarrow{\ell} G'$ and $\Delta \xrightarrow{\ell} \Delta'$ as they have been defined in this chapter.

Coinductive Projection. One can be confident in the soundness of the definition of Castro-Perez et al. because they prove it sound with respect to a specification of projection given by Ghilezan et al. [31] in the Coq proof assistant. This specification is defined on coinductive global and local types, which do not have μ-binders, and this gives a declarative definition of what projection of recursion should satisfy. Coinductive projection is denoted by $G^\nu \downharpoonright^\nu_{\mathsf{p}} T^\nu$, where G^ν and T^ν are coinductively generated by the grammars below:

$$G^\nu \quad ::= \quad \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.G^\nu \mid \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\{l_j : G_j^\nu\}_{j\in J} \mid \mathsf{end}^\nu$$

$$T^\nu \quad ::= \quad !^\nu k\langle U\rangle.T^\nu \mid ?^\nu k\langle U\rangle.T^\nu \mid k \oplus^\nu \{l_i : T_i^\nu\}_{i\in I} \mid k \,\&^\nu\, \{l_i : T_i^\nu\}_{i\in I} \mid \mathsf{end}^\nu$$

The constructs of coinductive global and local types closely match the corresponding inductive definitions, except for the omission of the μ-binder and

variable constructs. This is because coinductive definitions may be circular, allowing the inductive global type $\mu \mathbf{t}.\mathsf{p} \to \mathsf{q} : k\langle U \rangle.\mathbf{t}$ to be represented as:

$$G_1^\nu = \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U \rangle.G_1^\nu \tag{2.1}$$

Castro-Perez et al. [18] makes this notion precise, by defining a coinductive relation between inductive and coinductive types $G \mathcal{R} G^\nu$ which expresses that the repeated unfolding of inductive global type G corresponds to the coinductive type G^ν. As an example, consider the following rule of coinductive projection:

$$\frac{G^\nu \upharpoonright^\nu_{\mathsf{p}} \; T^\nu}{\mathsf{p} \xrightarrow{\nu} \mathsf{p}_2 : k\langle U \rangle.G^\nu \; \upharpoonright^\nu_{\mathsf{p}} \; !^\nu k\langle U \rangle.T^\nu} \; [\mathsf{M1}\upharpoonright^\nu]$$

When the projected role p matches the sender in the message interaction $\mathsf{p} \to \mathsf{p}_2 : k\langle U \rangle$ of the global type, then it must be related to a local type with a $!^\nu k\langle U \rangle$ prefix. The continuations G^ν and T^ν must also be related by coinductive projection. The premise is separated from the conclusion by a double line, which indicates that a derivation may be circular. This allows us to relate G_1^ν from Equation (2.1) with $T_1^\nu = !^\nu k\langle U \rangle.T_1^\nu$ in the following way:

$$\cfrac{\mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U \rangle.G_1^\nu \upharpoonright^\nu_{\mathsf{p}} !^\nu k\langle U \rangle.T_1^\nu}{G_1^\nu \upharpoonright^\nu_{\mathsf{p}} \; T_1^\nu} \; [\mathsf{M1}\upharpoonright^\nu] \tag{2.2}$$

For a full overview of coinductive projection, we refer to our papers in Appendix A and Appendix B.

Subject Reduction. Multiparty session types replace the duality of binary session types with *coherence*. The shape of the typing judgment stays the same $\Gamma \vdash P \triangleright \Delta$, but we call a wellformed Δ for coherent, rather than balanced, and this means that the local types in Δ are projections of global types that satisfy the linearity predicate. More precisely, we say that a set of local types $\{\mathsf{p}_i : T_{\mathsf{p}_i} \mid i \in I\}$ is coherent if there exists a global type G whose roles are $\{\mathsf{p}_i \mid i \in I\}$ and it holds for each role p_i that $T_{\mathsf{p}_i} = G \upharpoonright_{\mathsf{p}_i}$. This is generalised to an arbitrary Δ that possibly contains session channels from multiple sessions by requiring that the environment obtained by keeping only entries with the session channels of session s, denoted by $\Delta(s)$, is coherent. We use coherence in the typing rule for restriction:

$$\frac{\Gamma \vdash P \triangleright \Delta, \{\mathsf{s}^{\mathsf{p}_i} : T_{\mathsf{p}_i} \mid i \in I\} \quad \mathsf{coherent}(\{\mathsf{p}_i : T_{\mathsf{p}_i} \mid i \in I\})}{\Gamma \vdash (\boldsymbol{\nu} s) P \triangleright \Delta}$$

This rule is very similar to the restriction rule for binary session types, differing in two ways by using located session channels s^{p} instead of polarised channels k^+, and by replacing duality with coherence. Coherence is also used to state subject reduction. For multiparty session types this theorem states that that typing is preserved after reduction, and that the communication of processes is reflected by the LTS of local type environments. A simplified version of the Honda et al. theorem statement is:

$$\begin{array}{c}\text{If } \Gamma \vdash P \triangleright \Delta \text{ with } \Delta \text{ coherent and } P \longrightarrow P' \text{ then there exists } \Delta' \text{ s.t.} \\ \Gamma \vdash P' \triangleright \Delta' \text{ and } \Delta = \Delta' \text{ or there exists an } \ell \text{ such that } \Delta \xrightarrow{\ell} \Delta'\end{array}$$

This is a central property of multiparty session types. It is the foundation for proving three guarantees that are presented in Honda et al.:

1. Communication safety, which means that well typed processes "do not go wrong". This property ensures that the messages and labels received, will be of the correct type.

2. Session fidelity, meaning that a multiparty session will respect the protocol assigned to it. That is, the reductions that a process takes can be mapped back to the global type that specifies its session which will make corresponding reductions in terms of its LTS.

3. Progress (well typed processes do not get stuck). This ensures that a process does not deadlock. Honda et al. prove this property for processes communicating over a single session. Later work has developed techniques for generalising this to processes with multiple interleaved sessions [14].

Remark 2.2 *Our definition of the Δ environment differs from Honda et al.. In their work an entry in Δ is a vector of session channels $\tilde{s}$ which is assigned to a set of located local types. A located local type is a local type annotated by a role, denoted by $T@\mathsf{p}$. An entry in the original definition by Honda et al. therefore has the shape:*

$$\tilde{s} : \{T_{\mathsf{p_i}}@\mathsf{p_i} \mid i \in I\}$$

To the best of our knowledge all later work has moved away from this practice, in favor of entries of consisting of a single located session channel which is assigned to a local type:

$$\mathsf{s}^{\mathsf{p}} : T$$

We also follow this newer practice, which can be seen in the typing rule for restriction above. This is very similar to polarised channels of binary session type, with roles replacing polarities $\{+, i\}$.

Failures of Subject Reduction. Scalas and Yoshida [67] present a problem that occurs in many multiparty session type papers; in particular affecting their subject reduction proofs. The problem is rather technical, and it is caused by the interplay between a coherence definition for the Δ environment [77], and an extension to projection called *full merging* [17], two concepts that have been used in many later papers [67].

Chapter 3

Multiparty Session Types in Coq

Session types is part of the larger field called *formal methods*, where the behavior of programs are specified and proved correct by the formal means of mathematics and logic. Such results often rely on the error prone process of writing lengthy complicated proofs *by hand*, making mistakes likely and decreasing trust in the results. It has become more common to accompany research papers with *mechanised* proofs in an interactive proof assistant, verifying the claims of the results in these papers. These interactive proof assistants are tools that help the user to write proofs. The tool keeps track of what needs to be proven, what may be assumed and often supports some form of automation. A mechanised proof that verifies a formal result is an implementation of this proof in a proof assistant, and because the proof script has been checked by the proof assistant, one has the highest guarantees of the correctness.

The Coq Proof Assistant. Results in computer science and mathematics and are verified in many proof assistants, and these proof assistants are based on different theoretical foundations and offer different degrees of automation. In this project we choose the Coq proof assistant, and this choice is mostly due to prior experience with the tool. Coq is based on the Calculus of Inductive Constructions [13] which is a dependently typed lambda calculus with inductive and coinductive definitions. The functional programming language used for writing programs in Coq is called *gallina* and is a subset of the OCaml functional language [51], supporting a subset of OCaml's features.

A faithful representation Coq. The methodology taken in this thesis is to use the Coq proof assistant to verify results of Honda et al.. The aim is to give a faithful representation of their definitions in Coq, changing them only if necessary, particularly convenient or in an attempt to align with more recent developments in multiparty session types. As an example of alignment, see Remark 2.2 about the use of located session channels. A list of changes that we made to their type system can be found in Chapter 4.2, in the paragraph "A New Type System". Based on the presentation of multiparty session types we gave in Chapter 2, we now show global types have been represented in Coq.

3.1 Global types in Coq

We begin by showing how the two-buyer protocol may be represented in Coq[1]:

```
Definition two_buyer : gType :=
 GMSg (B1,S,k_1s) (VSort String).
 GMSg (S,B1,k_s1) (VSort Int).
 GMsg (S,B2,k_s2) (VSort Int).
 GMsg (B1,B2,k_12) (VSort Int).
 GBranch (B2,S,k_2s) [(0, GMsg (B2,S,k_2s) (VSort String).
                          GMsg (S,B2,k_s2) (VSort Date).
                          GEnd);
                      (1, GEnd)].
```

Here, we represent the message interaction $\mathtt{p} \to \mathtt{q} : k\langle U \rangle.G$ with the `GMsg a u g`, where `a` is an action whose type is a ternary tuple of two roles and an explicit channel `ptcp * ptcp * ch`, `u` is a value which may either be a sort, as in `VSort String` or a local type. The branching interaction $\mathtt{p} \to \mathtt{q} : k\{l_j : G_j\}_{j \in J}$ is represented as `GBranch a gs`, where `a` is an action and `gs` is a list of global types paired with a natural number labeling each branch. We represent `end` with `GEnd`. Note that we have included five distinct explicit channels for ordering of roles in an interaction to mimic the implicit channels used to write the protocol in Chapter 2.

[1]The actual implementation keeps the expression language minimal by only supporting boolean expressions. The extension to other simple types is trivial and orthogonal to our goals.

Global type syntax and Autosubst2. The full syntax of global types is inductively defined by the type `gType` seen below:

```
Inductive gType : Set :=
| GMsg : action -> value -> gType -> gType
| GBranch : action -> list (nat * gType) -> gType
| GEnd : gType
| GRec : gType -> gType
| GVar : nat -> gType.
```

The construct `GRec G` represents $\mu\mathbf{t}.G$, making this constructor a binder for de Bruijn indices `GVar n`, which indicate by their natural number `n` the distance to their binder. This syntax definition has been generated by the Autosubst2 library [68], which it generated from a signature file an excerpt is shown of below:

```
GMsg : action -> value -> gType -> gType
GBranch : action -> "list" ("prod" (nat,gType)) -> gType
GEnd : gType
GRec : (gType -> gType) -> gType
```

The first three lines are responsible for generating the constructors for message interaction, branching interaction and `end`. We indicate that `GRec g` is a binder with a higher-order abstract syntax notation, seen in `GRec : (gType -> gType) -> gType`, which Autosubst2 translates to a de Bruijn representation. The constructor `GVar : nat -> gType` is thus constructed without being mentioned in the signature file. As an example of how these de Bruijn Indices work, consider the variables $\mathbf{t}$ and $\mathbf{t}'$ which bind respectively to the outer and inner binder in the global type below:

$$G_0 = \mu\mathbf{t}.\mu\mathbf{t}'.\mathtt{p} \to \mathtt{q}\colon k\left\{\begin{array}{l}\mathtt{Left}\colon \mathbf{t}\\ \mathtt{Right}\colon \mathbf{t}'\end{array}\right\} \tag{3.1}$$

We represent these variables as `GVar 1` and `GVar 0` in `G_0` seen below, where labels `Left` and `Right` are represented with `0` and `1` respectively:

```
Definition G_0 :=
 GRec (GRec (GBranch (p,q,k) [(0, GVar 1); (1, GVar 0)]))
```

Unfolding in Coq. The μ-binder in $\mu\mathbf{t}.G$ is used to define tail-recursive protocols by unfolding them when they occur at top level to $G[\mu\mathbf{t}.G/\mathbf{t}]$, as we saw in rule [GR5] of the LTS for global types. Performing this operation on the outer binder of G_0 in Equation (3.1) looks like this:

$$G_1 = \mu\mathbf{t}'.\mathtt{p} \to \mathtt{q}\colon k\left\{\begin{array}{l}\mathtt{Left}\colon \mathbf{t}\\ \mathtt{Right}\colon \mathbf{t}'\end{array}\right\}[G_0/\mathbf{t}] \tag{3.2}$$

$$= \mu\mathbf{t}'.\mathtt{p} \to \mathtt{q}\colon k\left\{\begin{array}{l}\mathtt{Left}\colon G_0\\ \mathtt{Right}\colon \mathbf{t}'\end{array}\right\} \tag{3.3}$$

The corresponding operation on `G_0` requires a substitution function on global types. The substitution function seen below was automatically generated by Autosubst2 from the signature file, and has the following type:

```
Fixpoint subst_gType (f : nat -> gType) (g : gType) : gType :=
 ...
```

We omit its definition because it is verbose. The function takes as arguments a substitution function `f: nat -> gType` that maps all indices to global types, and a global type `G : gType` on which the substitution is performed. This substitution touches all the free variables of the global type and is called a parallel substitution, in contrast to the single-point substitution $G[\mu\mathbf{t}.G/\mathbf{t}]$ we performed above. Autosubst2 implements parallel substitution and provides a lot of tactical support for simplifying composed parallel substitution expressions. Using the substitution function generated by Autosubst2, we perform the unfolding that was done in Equation (3.2) the following way:

```
Definition G_1 :=
subst_gType (scons G_0 GVar)
            (GRec (GBranch (p,q,k) [(0, GVar 1); (1, GVar 0)]))
```

Here `(scons x f) : nat -> gType` is a substitution that replaces `GVar 0` with `x`, and decrements all other variables. Autosubst2 defines `scons` the following way[2]:

```
Definition scons (G : gType) (f : nat -> gType) n : gType :=
  match n with | 0 => x | S n' => f n'
  end.
```

[2]The actual definition used by Autosubst2 is more general and makes use of typing features of Coq we do not explain.

One can state and prove the equality in Equation (3.3) like this:

```
Lemma G_eq :
  G_1 = (GRec (GBranch (p,q,k) [(0, G_0); (1, GVar 0)]))
Proof. rewrite /G_1 //=. Qed.
```

Here `Proof` and `Qed` mark respectively the beginning and end of a proof script. The tactic `/G_1 //=` unfolds the definition of `G_1`, simplifies the substitution expression and finishes the proof by reflexivity.

Remark 3.1 *Mechanising inductive μ-types is non-trivial. In Chapter 2.2 we saw that the structurally recursive definition of projection by Honda et al. can produce non-contractive types. The challenge of inductive μ-types is due to the interplay between definitions that unfold the μ-binder, and those that structurally recurse under it. This challenge is addressed in many mechanisations of binary and multiparty session types by avoiding the inductive μ-binder in favor of other choices of representation. Examples include:*

- *Castro-Perez et al. [19] who mechanised a subject reduction result for inductive binary session types without recursion.*
- *Jacobs et al. [44] who mechanised multiparty session types in a novel setting, representing global and local types coinductively and avoiding the need for a μ-binder because coinductive definitions can be circular.*
- *Castro-Perez et al. [18] who rely partially on coinductive types for the formal guarantees of a multiparty session type based tool they develop.*
- *Hinrichsen et al. [37] who mechanised binary session types using a program logic, and represented recursive session types via the general fixpoint operator of the program logic.*

Staying faithful to the original formulation by Honda et al. we represent global and local types inductively with the μ-binder. Equality is also undecidable for coinductive types [12], which makes inductive types superior in our setting where the desire is to support the future development of tools.

Chapter 4

Results

This PhD project makes the following contributions to the field of session types

- **Projection** We identify multiple either incorrect or severely restrictive definitions of projection; we develop a new projection procedure, proven sound and complete in Coq with respect to the coinductive specification by Ghilezan et al. [31]. This affords inductive types the expressivity of coinductive types in a mechanised computable projection, for which we also mechanise a proof of Endpoint Projection Theorem.

- **Subject Reduction** We present a counterexample to the Subject Reduction result of Honda et al.. A new type system that makes use of our new projection definition is presented, along with a mechanised proof of subject reduction for this type system.

We now give a detailed explanation of these contributions.

4.1 Projection

Incorrect and Restrictive Projections. Glabbeek et al. [74] identified that the recursion condition in Bejleri and Yoshida [11] results in an incorrect projection, returning end when a recursion variable should have been returned. We extend on their contribution by identifying three additional problematic recursion conditions. One of these conditions similarly returns

Figure 4.1: Projection of the μ-binder.

Paper	Condition on $\mu\mathbf{t}.G\upharpoonright_{\mathsf{p}}$	Problem
Honda et al. [39]	$G\upharpoonright_{\mathsf{p}} \neq \mathsf{end}$	Restrictive (4.1)
Bettini et al. [14]	None	Restrictive (4.1)
Yoshida et al. [77]	None ($\mu\mathbf{t}.\mathbf{t}$ identified with end)	-
Bejleri and Yoshida [11]	$\mathsf{p} \in G$	Incorrect (4.2)
Demangeon and Yoshida [25] Coppo et al. [22]	$G\upharpoonright_{\mathsf{p}} \neq \mathbf{t}$	Restrictive (4.3)
Scalas and Yoshida [67]	$\forall \mathbf{t}, G\upharpoonright_{\mathsf{p}} \neq \mathbf{t}$	Incorrect (4.2)
Castro-Perez [18]	$\mathsf{gVar}(\mathbf{t}, G)$	-
Glabbeek et al. [74] Yoshida and Hou [78]	$\mathsf{p} \in G \vee \neg\mathsf{closed}(\mu\mathbf{t}.G)$	-

end when a variable should have been returned and is thus incorrect. The two other conditions, when considered informally, appear correct but they severely restrict the set of projectable global types. Collectively these incorrect and restrictive conditions span four well-cited papers, and to provide a comparison we list them along with four other papers that use three distinct recursion conditions that are neither incorrect nor severely restrictive. The conditions are summarised in Figure 4.1. For each publication, or pair of publications, the recursion condition that has been used is shown, indicating the problem that occurs, if at all, and a global type that causes it. We use the three global types below:

$$\mathsf{p} \to \mathsf{s} : k\langle U\rangle.\mu\mathbf{t}.\mathsf{q} \to \mathsf{r} : k'\langle U'\rangle.\mathbf{t} \tag{4.1}$$

$$\mu\mathbf{t}.\mathsf{p} \to \mathsf{r} : k\langle U\rangle.\mu\mathbf{t}'.\mathbf{t} \tag{4.2}$$

$$\mathsf{p} \to \mathsf{s} : k\langle U\rangle.\mu\mathbf{t}.\mathsf{q} \to \mathsf{r} : k\langle U\rangle.\mu\mathbf{t}'.\mathbf{t} \tag{4.3}$$

Starting with the Honda et al. condition, we recall that Chapter 2 shows that this definition can produce $\mu\mathbf{t}.\mathbf{t}$. In particular (4.1) projects on p to $!k\langle U\rangle.\mu\mathbf{t}.\mathbf{t}$. Because Honda et al. explicitly state that they assume μ-types to always be closed and contractive, an argument can be made that because the produced local type is an exotic term, then the projection of p on (4.1) should be undefined. This would also mean that (4.1) is not projectable,

since p is a role in the global type. Moreover it would mean that any global type $\mathsf{p} \to \mathsf{s} : k\langle U\rangle.G$ with a recursive definition in G that does not involve p would be undefined. This rules out many recursive global types. Bettini et al. suffer the same problem since they have no condition from which it is straightforward to produce $\mu\mathbf{t}.\mathbf{t}$.

Yoshida et al. does not have this issue, in spite of having no condition, and this is because they explicitly make the additional assumption that $\mu\mathbf{t}.\mathbf{t}$ is equated with end.

Bejleri and Yoshida use the condition $\mathsf{p} \in G$ which checks for the presence of the role in G. The incorrectness of using this as the condition for projecting recursion was spotted by Glabbeek et al. [74]. A variation on their example is (4.2). The incorrectness is then seen by comparing the specification for p in the global and local type:

$$\mu\mathbf{t}.\mathsf{p} \to \mathsf{r} : k\langle U\rangle.\mu\mathbf{t}'.\mathbf{t} \upharpoonright_{\mathsf{p}} = \mu\mathbf{t}.!k\langle U\rangle.\mathsf{end}$$

The global type specifies the repeated sending on channel k, which the local type has replaced the recursion variable $\mathbf{t}$ with end, incorrectly specifying the communication to take place only once.

Demangeon and Yoshida along with Coppo et al. use a condition that with (4.3) projected on p produces $!k\langle U\rangle.\mu\mathbf{t}.\mu\mathbf{t}'.\mathbf{t}$, suffering similar restrictiveness to Honda et al..

Scalas and Yoshida use a more general condition that checks whether the projection of the body is any variable, not just a specific variable bound to the binder being projected. This results in the same incorrect behavior as in Bejleri and Yoshida, (4.2) specifies recursive behavior for p which projection translates to finite behavior.

We covered in Chapter 2 the correctness of the condition used in Castro-Perez et al.. A more expressive and correct condition was presented by Glabbeek et al. who are able to project the following global type on p:

$$\mathsf{p} \to \mathsf{s} : k_0\langle U\rangle.\mu\mathbf{t}.\mathsf{q} \to \mathsf{r} : k_1 \left\{ \begin{array}{l} l_1 : \mathsf{q} \to \mathsf{r} : k_2\langle U'\rangle.\mathbf{t} \\ l_2 : \mathsf{end} \end{array} \right\} \upharpoonright_{\mathsf{p}} = !k_0\langle U\rangle.\mathsf{end} \quad (4.4)$$

The condition they use is the only one known in the literature that can correctly project such a global type, where the projection on the branches are distinct (e.g $\mathbf{t}$ and end), yet due to checking for the presence of the role $\mathsf{p} \in G$ the procedure may return end before the failing branching check is performed. While Bejleri and Yoshida achieve the same, their omission of the disjunct $\neg\mathsf{closed}(\mu\mathbf{t}.G)$ invites (4.2) as a counterexample.

A Sound and Complete Projection. We present a projection procedure that unlike prior work is not only proved sound with respect to the coinductive specification $G^\nu \downarrow^\nu_{\mathsf{p}} T^\nu$ presented in Chapter 2, but also complete. We denote our projection function by $\mathsf{proj}_{\mathsf{p}}(G)$ and its definition can be seen below:

$$\mathsf{proj}_{\mathsf{p}}(G) = \begin{cases} \mathsf{trans}_{\mathsf{p}}(G) & \text{if } \mathsf{projectable}_{\mathsf{p}}(G) \\ \textit{undefined} & \text{otherwise} \end{cases} \tag{4.5}$$

Here $\mathsf{trans}_{\mathsf{p}}(G)$ is the total function that is derived by removing the branching check from the Castro-Perez et al. projection. We use a test $\mathsf{projectable}_{\mathsf{p}}(G)$ to check whether the global type is projectable on p. We use unravelling relations $G \mathrel{\mathcal{R}} G^\nu$ and $T \mathrel{\mathcal{R}} T^\nu$ to state soundness and completeness properties about this projection function. For our present purposes it is sufficient to rely on the intuitive description of unravelling given in Chapter 2, namely that unravelling asserts that the coinductive type corresponds to the inductive type unfolded to the limit. We explain the unravelling relations in detail in Appendix B.

The soundness and completeness properties for our projection function are the following:

Theorem 4.1 (Soundness of Projection) *If* $\mathsf{proj}_{\mathsf{p}}(G)$ *is defined then there exist coinductive types* G^ν *and* T^ν *such that* $G \mathrel{\mathcal{R}} G^\nu$, $\mathsf{proj}_{\mathsf{p}}(G) \mathrel{\mathcal{R}} T^\nu$ *and* $G^\nu \downarrow_p T^\nu$.

Theorem 4.2 (Completeness of Projection) *If* $G^\nu \downarrow^{co}_p T^\nu$ *and* $G \mathrel{\mathcal{R}} G^\nu$ *then* $\mathsf{proj}_{\mathsf{p}}(G)$ *is defined and* $\mathsf{proj}_{\mathsf{p}}(G) \mathrel{\mathcal{R}} T^\nu$.

Informally, soundness states that when $\mathsf{proj}_{\mathsf{p}}(G)$ is defined, then the unravelling of G and $\mathsf{proj}_{\mathsf{p}}(G)$ are related by coinductive projection. This property follows by construction because it is exactly what $\mathsf{projectable}_{\mathsf{p}}(G)$ checks. Completeness on the other hand states that if $G^\nu \downarrow^\nu_{\mathsf{p}} T^\nu$, then $\mathsf{proj}_{\mathsf{p}}(G)$ is defined for G if it unravels to G^ν, and $\mathsf{proj}_{\mathsf{p}}(G)$ unravels to T^ν. To illustrate what it means for our projection to be complete, consider the global type below which is projectable by both Castro-Perez et al. and us:

$$\mu \mathbf{t}.\ \mathsf{p} \to \mathsf{q} : k\langle \mathtt{String} \rangle.\ \mathsf{r} \to \mathsf{s} : k'\{\mathtt{Left} : \mathbf{t},\quad \mathtt{Right} : \mathbf{t}\} \tag{4.6}$$

This global type specifies that the first interaction that will take place is $\mathsf{p} \to \mathsf{q} : k$, afterwards $\mathsf{r} \to \mathsf{s} : k'$ will occur, and then the protocol repeats.

This can also be expressed with two binders the following way:

$$\mu \mathbf{t}.\ \mathsf{p} \to \mathsf{q} : k\langle \mathtt{String} \rangle.\ \mu \mathbf{t}'.\ \mathsf{r} \to \mathsf{s} : k' \left\{ \begin{array}{l} \mathtt{Left} : \ \mathbf{t} \\ \mathtt{Right} : \ \mathsf{p} \to \mathsf{q} : k\langle \mathtt{String} \rangle.\mathbf{t}' \end{array} \right\} \quad (4.7)$$

Because the two branches in Equation (4.6) are the same, the Castro-Perez et al. projection is defined for this global type. This is not the case for the global type in Equation (4.7) where the `Left` branch projects to $\mathbf{t}$ and the `Right` branch projects to $!k\langle \mathtt{String} \rangle.\mathbf{t}'$. For this reason, the projection will be undefined for all projections that rely on the branches projecting to the same thing. This includes the Castro-Perez et al. projection and all other projections on inductive types in the literature. On the other hand, our projection relies on the $\mathsf{projectable}_{\mathsf{p}}(G)$ which intuitively decides if the unravelling of G and $\mathsf{trans}_{\mathsf{p}}(G)$ into the coinductive types we denote as G^ν and T^ν are related by coinductive projection $G^\nu \downarrow^\nu_{\mathsf{p}} T^\nu$.

Endpoint Projection Theorem. The Endpoint Projection Theorem relates the LTS of global and local types. If a global type can reduce, then the environment of local types that is derived from the global type through projection can reduce as well. Likewise, if the environment of local types derived through projection of a global type can reduce, then the global type in question can as well. Both directions of the theorem assumes that this initial global type is coherent, which means it is linear and projectable. Crucial for proving the theorem, is the fact that projectability must be preserved by global type reductions. That is, whenever $G \xrightarrow{l} G'$ and G is projectable, then also G' must be projectable. Recall that a global type is projectable if its projection is defined on all roles. One of the rules for the LTS of global types is the one seen below:

$$\frac{\forall i \in I.\ G_i \xrightarrow{\ell} G_i' \qquad \mathsf{q} \notin \ell}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i \in I} \xrightarrow{\ell} \mathsf{p} \to \mathsf{q} : k\{l_i : G_i'\}_{i \in I}} \ [\text{GR4}]$$

Honda et al. prove preservation of projectability by induction on $G \xrightarrow{l} G'$ (proposition 4.4) but the case for [GR4] is incorrect. They incorrectly state that the proof is immediate by induction hypothesis. This is not the case, and the reason for this illustrates the challenge in proving this property.

In showing preservation of projectability we may assume it to hold for $\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i \in I}$, meaning the projection onto all roles is defined, which means the branching condition is satisfied for all roles that are not p nor q:

$$\forall \mathsf{c} \notin \{\mathsf{p}, \mathsf{q}\}, \forall i, j \in I,\ G_i \upharpoonright_{\mathsf{c}} = G_j \upharpoonright_{\mathsf{c}}$$

The induction hypothesis tells us that each $G'_i, \forall i \in I$ are projectable and from this we must show that also $\mathsf{p} \to \mathsf{q} : k\{l_i : G'_i\}_{i \in I}$ is projectable, which reduces to showing the following branching condition is satisfied:

$$\forall \mathsf{c} \notin \{\mathsf{p}, \mathsf{q}\}, \forall i, j \in I,\ G'_i \upharpoonright_{\mathsf{c}} = G'_j \upharpoonright_{\mathsf{c}}$$

This property must somehow be derived from the assertions about the initial global type. A natural way to proceed would be to fix c, and in the case where $J = \{1, 2\}$ we would have to show the following:

$$(\forall \mathsf{d} \notin \{\mathsf{p}, \mathsf{q}\},\ G_0 \upharpoonright_{\mathsf{d}} = G_1 \upharpoonright_{\mathsf{d}}) \implies G'_0 \upharpoonright_{\mathsf{c}} = G'_1 \upharpoonright_{\mathsf{c}} \tag{4.8}$$

One would expect the only relevant part of the premise above is the projection on $\mathsf{c} \notin \{\mathsf{p}, \mathsf{q}\}$, and thus that it would suffice to show the stronger property seen below:

$$G_0 \upharpoonright_{\mathsf{c}} = G_1 \upharpoonright_{\mathsf{c}} \implies G'_0 \upharpoonright_{\mathsf{c}} = G'_1 \upharpoonright_{\mathsf{c}} \tag{4.9}$$

While the weaker statement in Equation (4.8) holds, the stronger statement in Equation (4.9) does not. A counterexample is a reduction by label $\mathsf{a} \to \mathsf{d} : k\langle \mathtt{int} \rangle$ on the global types below:

$$G_0 = \mathsf{a} \to \mathsf{d} : k\langle \mathtt{int} \rangle.\mu \mathbf{t}.\mathsf{a} \to \mathsf{c} : k'\langle \mathtt{int} \rangle.\mathsf{a} \to \mathsf{d} : k'\langle \mathtt{int} \rangle.\mathbf{t} \tag{4.10}$$

$$G_1 = \mu \mathbf{t}.\mathsf{a} \to \mathsf{c} : k\langle \mathtt{int} \rangle.\mathsf{a} \to \mathsf{d} : k'\langle \mathtt{int} \rangle.\mathbf{t} \tag{4.11}$$

Both G_0 and G_1 have the same projection on c

$$\mu \mathbf{t}.?k\langle \mathsf{int} \rangle.\mathbf{t}$$

After reduction, the global types become

$$G'_0 = \mu \mathbf{t}.\mathsf{a} \to \mathsf{c} : k'\langle \mathtt{int} \rangle.\mathsf{a} \to \mathsf{d} : k\langle \mathtt{int} \rangle.\mathbf{t} \tag{4.12}$$

$$G'_1 = \mathsf{a} \to \mathsf{c} : k'\langle \mathtt{int} \rangle.\mu \mathbf{t}.\mathsf{a} \to \mathsf{c} : k'\langle \mathtt{int} \rangle.\mathsf{a} \to \mathsf{d} : k\langle \mathtt{int} \rangle.\mathbf{t} \tag{4.13}$$

Now the projection on c is not the same

$$G'_0 \upharpoonright_{\mathsf{c}} = \mu \mathbf{t}.?k\langle \mathtt{int} \rangle.\mathbf{t} \qquad G'_1 \upharpoonright_{\mathsf{c}} = ?k\langle \mathtt{int} \rangle.\mu \mathbf{t}.?k\langle \mathtt{int} \rangle.\mathbf{t}$$

They do however correspond to the same protocol, one of them is just unfolded (e.g $\mathsf{unf}(G'_0 \upharpoonright_{\mathsf{p}}) = G'_1 \upharpoonright_{\mathsf{c}}$). The syntactic equality assertion does

not recognise types up to unfolding, and this demonstrates why we cannot reduce preservation of projectability to the strengthened statement in Equation (4.9). The reason for this is that the universal quantification in the premise of the weaker statement in Equation (4.8) includes an equality assertion on the projection of d, from which we infer that the two global types have unfolded the same number of times

This highlights the counterintuitive idea that the simple syntactic branching condition can, due to its restrictiveness, make proofs more difficult. In this case, we had to use an additional role (d) to ensure both branches had unfolded the same number of times.

We would completely avoid this problem if the branching condition used a coinductive equality up to unfolding. It is a strong benefit that our sound and complete projection satisfies exactly this property. To be precise, our branching condition is the side condition $\mathsf{projectable}_{\mathsf{p}}(G)$ in Equation (4.5) which satisfies that if the following holds

$$\mathsf{projectable}_{\mathsf{c}}\left(\mathsf{p} \to \mathsf{q}\colon k\left\{\begin{array}{l}\texttt{Left}\colon G_0\\ \texttt{Right}\colon G_1\end{array}\right\}\right)$$

Then it also holds that

$$\mathsf{proj}_{\mathsf{c}}(G_0) \approx \mathsf{proj}_{\mathsf{c}}(G_1)$$

This makes the technique which previously failed for the syntactic assertion applicable with our more flexible branching check by satisfying the implication:

$$\mathsf{proj}_{\mathsf{c}}(G_0) \approx \mathsf{proj}_{\mathsf{c}}(G_1) \implies \mathsf{proj}_{\mathsf{c}}(G_0') \approx \mathsf{proj}_{\mathsf{p}}(G_1') \tag{4.14}$$

This allows us to reason about the projection of c alone, avoiding the counterexample because while it is not the case that $\mathsf{proj}_{\mathsf{c}}(G_0') = \mathsf{proj}_{\mathsf{c}}(G_1')$, it is the case that $\mathsf{proj}_{\mathsf{c}}(G_0') \approx \mathsf{proj}_{\mathsf{c}}(G_1')$.

4.2 Subject Reduction

A Counterexample. We have identified a counterexample to the Subject Reduction result of Honda et al.. Recall the shape of the subject reduction theorem we gave in Chapter 2:

If $\Gamma \vdash P \triangleright \Delta$ with Δ coherent and $P \longrightarrow P'$ then there exists Δ' s.t. $\Gamma \vdash P' \triangleright \Delta'$ and either $\Delta = \Delta'$ or there exists an ℓ such that $\Delta \xrightarrow{\ell} \Delta'$

The counterexample shows an initial process P_0 that performs two sending actions, thereby becoming P in the theorem above, and then performing a final receiving action and becoming P'. The typing environment Δ is used to type both P_0 and P. There does however not exist a Δ' that can type P' where either $\Delta = \Delta'$, or there exists a label ℓ such that $\Delta \xrightarrow{\ell} \Delta'$ is derivable. The counterexample is presented in detail in Appendix C and we now summarise it, starting with the process P_0 seen below:

$$\overbrace{\mathsf{s}^{\mathsf{p}}[1] \lhd l_2; \mathsf{s}^{\mathsf{p}}[2]!\langle \mathsf{true} \rangle; \mathbf{0}}^{\mathsf{p}} \quad | \quad \overbrace{\mathsf{s}^{\mathsf{q}}[1] \rhd \{l_1 : \mathsf{s}^{\mathsf{q}}[2]!\langle \mathsf{false} \rangle; \mathbf{0}, l_2 : \mathbf{0}\}}^{\mathsf{q}} \quad | \quad \underbrace{\mathsf{s}^{\mathsf{r}}[2]?(x); \mathbf{0}}_{\mathsf{r}} \quad | \quad \mathsf{s}[1] :: \emptyset \quad | \quad \mathsf{s}[2] :: \emptyset \tag{4.15}$$

The P_0 process seen in Equation (4.15) is a multiparty session that consists of two queues addressed at $s[1]$ and $s[2]$ and three roles: p is about to send label l_2 to $s[1]$, and then true to $s[2]$; q is awaiting a label from $s[1]$; r is awaiting a value from $s[2]$. We can type this session with an environment of three local types:

$$\mathsf{p} : 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : !2\langle \mathsf{bool} \rangle; \mathsf{end} \end{array} \right\}, \qquad \mathsf{q} : 1 \& \left\{ \begin{array}{l} l_1 : !2\langle \mathsf{bool} \rangle; \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}, \qquad \mathsf{r} : ?2\langle \mathsf{bool} \rangle; \mathsf{end} \tag{4.16}$$

We will refer to the environment in Equation (4.16) as Δ. In this environment, the type for p specifies that it will send either label l_1 or l_2, and continue according to the branch associated with the chosen label. In the implementation of p we see that it will send l_2. The remaining communication of p follows the specification of the l_2 branch by sending a bool. The explicit channels 1 and 2 in the type refer to the two queues at $s[1]$ and $s[2]$ in Equation (4.15). The type for q corresponds exactly to the shape of the q process and the same holds for r.

After p has sent l_2 and true the process becomes P seen below:

$$\overbrace{\mathbf{0}}^{\mathsf{p}} \quad | \quad \overbrace{\mathsf{s}^{\mathsf{q}}[1] \rhd \{l_1 : \mathsf{s}^{\mathsf{q}}[2]!\langle \mathsf{false} \rangle; \mathbf{0}, l_2 : \mathbf{0}\}}^{\mathsf{q}} \quad | \quad \underbrace{\mathsf{s}^{\mathsf{r}}[2]?(x); \mathbf{0}}_{\mathsf{r}} \quad | \quad \mathsf{s}[1] :: l_2{}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2] :: \mathsf{true}^{\mathsf{p}} \cdot \emptyset \tag{4.17}$$

The P process seen in Equation (4.17) shows p completed as $\mathbf{0}$; $s[1]$ contains $l_2{}^{\mathsf{p}}$ with the role annotation indicating that p sent this message; $s[2]$ contains $\mathsf{true}^{\mathsf{p}}$; q forms a redex with $s[1]$ and r forms a redex with $s[2]$. While the process has changed, its typing (e.g. Δ) stays the same. This is because Δ tracks interactions, and neither of the messages sent by p have been received yet. This means that both of the processes in Equation (4.15) and Equation (4.17) are typed by Δ. We now consider what happens after r reads from $\mathsf{s}^{\mathsf{r}}[2]$ and omitting P':

$$\begin{array}{l} \overbrace{\mathbf{0}}^{\mathsf{p}} \quad | \quad \overbrace{\mathsf{s}^{\mathsf{q}}[1] \rhd \{l_1 : \mathsf{s}^{\mathsf{q}}[2]!\langle \mathsf{false}\rangle; \mathbf{0}, l_2 : \mathbf{0}\}}^{\mathsf{q}} \quad | \\ \underbrace{\mathbf{0}}_{\mathsf{r}} \quad | \quad \mathsf{s}[1] :: l_2{}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2] :: \emptyset \end{array} \tag{4.18}$$

The P' process in Equation (4.18) shows the state of the session after the completion the interaction consisting of p sending to r a $\mathtt{bool}$ over the $s[2]$. This corresponds to the label $\mathsf{p} \to \mathsf{r} : 2\langle \mathtt{bool}\rangle$. Since this interaction has taken place in the session, we would expect the Δ environment that types the session to be able to reduce by this label, but that is not the case. That is, there exists no Δ' such that $\Delta \xrightarrow{\mathsf{p}\to\mathsf{r}:2\langle \mathtt{bool}\rangle} \Delta'$ is derivable. To see why, we must consider the Honda et al. environment semantics which are based on permutations of the communication actions in local types. As an example, if we wanted to perform a communication on explicit channel 2 involving a local type $!1\langle \mathsf{bool}\rangle.!2\langle \mathsf{bool}\rangle.\mathsf{end}$, we can permute its actions to $!2\langle \mathsf{bool}\rangle.!1\langle \mathsf{bool}\rangle.\mathsf{end}$. In our case, the issue is that our interaction $\Delta \xrightarrow{\mathsf{p}\to\mathsf{r}:2\langle \mathtt{bool}\rangle} \Delta'$ involves p as a sender of bool, and we would like to permute its type to the following type:

$$!2\langle \mathsf{bool}\rangle; 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}$$

This would allow Δ to reduce by $\mathsf{p} \to \mathsf{r} : 2\langle \mathtt{bool}\rangle$ to the following environment that types P':

$$\mathsf{p} : 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}, \qquad \mathsf{q} : 1\& \left\{ \begin{array}{l} l_1 : !2\langle \mathsf{bool}\rangle; \mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\}, \qquad \mathsf{r} : \mathsf{end} \tag{4.19}$$

It is however not possible to make the necessary permutation for p, and that is because the permutation rule for external choice follows the same idea as

the [GR4]-rule of global types. Intuitively, reordering a communication inside a branch to become the prefix of the local type, requires this reordering to take place in all branches. As an example, the necessary permutation would have been possible if Δ assigned p to the following type:

$$1 \oplus \left\{ \begin{array}{l} l_1 : !2\langle \mathsf{bool} \rangle; \mathsf{end} \\ l_2 : !2\langle \mathsf{bool} \rangle; \mathsf{end} \end{array} \right\}$$

We have established that Δ is stuck in spite of being used to type a process that completed the interaction $\mathsf{p} \to \mathsf{r} : 2\langle \mathtt{bool} \rangle$. Moreover, the process P' that completed the interaction is typable by the environment seen in Equation (4.19). The problem is therefore caused by the restrictiveness of the environment semantics $\Delta \xrightarrow{\ell} \Delta'$, and because of Endpoint Projection Theorem, this issue is also present in the global type semantics $G \xrightarrow{\ell} G'$. To see how this problem manifests in global types, consider if we wrap P in a session restriction. We can type $(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P$ with the restriction rule we saw in Chapter 2.2. We use the following coherent global type:

$$G = \mathsf{p} \to \mathsf{q} : 1 \left\{ \begin{array}{l} l_1 : \mathsf{q} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2 : \mathsf{p} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \end{array} \right\} \tag{4.20}$$

We have the identity below that makes Δ coherent:

$$\Delta = \{\mathsf{s}^{\mathsf{p}} : G\!\upharpoonright_{\mathsf{p}}, \mathsf{s}^{\mathsf{q}} : G\!\upharpoonright_{\mathsf{q}}, \mathsf{s}^{\mathsf{r}} : G\!\upharpoonright_{\mathsf{r}}\}$$

A typing derivation with the restriction rule would then look like this:

$$\frac{\overset{\cdots}{\Gamma \vdash P \triangleright \Delta} \quad \mathsf{coherent}(\Delta)}{\Gamma \vdash (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \triangleright \emptyset}$$

The problem of Δ not being able to reduce has now manifested itself on the global type since neither G can reduce by $\mathsf{p} \to \mathsf{r} : 2\langle \mathtt{bool} \rangle$ because the relevant rule [GR4] is not applicable. The global type that corresponds to Δ' is

$$G' = \mathsf{p} \to \mathsf{q} : 1 \left\{ \begin{array}{l} l_1 : \mathsf{q} \to \mathsf{r} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\} \tag{4.21}$$

The global type semantics does however not relate G and G' and this highlights a limitation of the global type semantics of Honda et al. which is discussed in Chapter 6.

Subject Reduction for a new Type System. We present a new type system, addressing the counterexample by imposing a new constraint on global types called *unstuck* that rules out the global type in Equation (4.20). Intuitively it checks that if a single branch of a global type can perform a reduction, denoted by $G \downarrow^1 \ell$, then a reduction by the global type LTS must be possible, denoted by the existence of a G' such that, $G \xrightarrow{\ell} G'$ is derivable. We say a global type is *unstuck* if it satisfies the predicate $\mathsf{unstuck}(G)$ coinductively defined as:

$$\frac{\forall \ell.\ G \downarrow^1 \ell \implies \exists G'. G \xrightarrow{\ell} G' \wedge \mathsf{unstuck}(G')}{\mathsf{unstuck}(G)}$$

We include unstuckness in the definition of coherence for global types, thus restricting the global types that may be used to introduce local type environments in the typing rule for restriction.

Besides the addition of unstuckness, other changes were also made from the original formulation by Honda et al.. We list the changes to the theory that we made, either of (N)ecessity (e.g *unstuck*), (C)onvenience, or to (A)lign with more recent developments in multiparty session types.

- (N) Honda et al. use typing contexts and subtyping to type queues. We replace this with the notion of *decomposition* denoted by $\mathsf{decomp}_{\vec{p}}\ T = (T_0, Q)$, which uses a path $\vec{p}$ that splits up T into the sending actions that have already been executed Q used for typing queues, and the remaining actions yet to be performed T_0. In this setting we call Q for a queue type, and T_0 for a residual local type. The use of decomposition instead of subtyping was necessary in the mechanisation because of the non-structural subsumption rule that implements subtyping. Typing systems with a subsumption rule require requires intensional reasoning about the shape of typing derivations. While such intensional reasoning is possible, it is a significant overhead which we avoided by relying instead on decomposition

- (C) We replace the single linear environment Δ with two environments $\Delta; \mathcal{Q}$. This conveniently maintains the executed part of a local type in $\mathcal{Q}$, and the remaining part in Δ, in the sense of the decomposition operation explained in the last point. This is in contrast to the approach by Honda et al. to use typing contexts $\mathcal{T}$ in the typing rule

for parallel composition, splitting the local types in the environment T into a context $\mathcal{T}$ and remaining local type T', such that the application $\mathcal{T}[T']$ is a subtype of T. As a consequence of us now having two environments $\Delta; \mathcal{Q}$ We also extend the definition of coherence of a local type environment to $\Delta; \mathcal{Q}$ by requiring that $\Delta; \mathcal{Q}$ be the point-wise decomposition of some coherent Δ_0. More precisely, each local type T in Δ_0 is decomposed according to some path $\vec{p}$, storing the queue type in $\mathcal{Q}$ and the residual local type in Δ. We then say that $\Delta; \mathcal{Q}$ is coherent *as* Δ_0 to mean that Δ_0 is coherent and its decomposition yields $\Delta; \mathcal{Q}$.

- (N) We replace the Honda et al. projection with our sound and complete projection. The necessity of this change is seen in the challenges of proving preservation of projectability in the presence of the restrictive branching check that we highlight above in Chapter (4.1).

- (N) Unlike Honda et al. we do not take the equi-recursive view. More precisely we do not make the assumption that T may freely be substituted for its unfolding $\mathsf{unf}(T)$ whenever necessary. Instead, local types are explicitly unfolded when necessary to do so. For example, the typing rule of $\mathbf{0}$ includes the premise $\mathsf{unf}(T) = \mathsf{end}$.

- (N) Related to the previous point, we explicitly use a coinductive equality in exactly three rules of the type system which are the rules of typing delegation [T-Deleg], process call [T-Var] and queues containing session channels [T-Qsess]. This is sufficient to prove that the type system is invariant to replacing the local types in Δ with coinductively equal local types. ($\approx$). This is necessary to prove the session restriction case of subject reduction.

- (A) We replace the use a vector of session channels (e.g. $\tilde{s}$), with the standard convention of located session channels (e.g. s^{p}) seen in most multiparty session type papers. Like Coppo et al. [22] and the technical report of Scalas and Yoshida [67], we annotate the messages in queues with their sender. Unlike Coppo et al. and Scalas and Yoshida we allow for more than one queue, staying in line with the presentation by Honda et al.

- (A) We remove the parallel construct $G_0 \mid\mid G_1$ of global types because it has been dropped in most later multiparty session type papers.

- (C) Our Coq mechanisation represents linear environments as association lists of key value pairs where the keys are unique. We use the same representation in our paper formalisation to stay as close as possible to the mechanisation. This would usually require a non-structural exchange rule for repositioning entries in the environment. We avoid the need for this by splitting environments, in the rule for parallel composition, as partitions according to boolean predicates about the keys.

- (C) We define the reduction semantics of local types differently than Honda et al.. Rather than using permutations, as was illustrated in the presentation of the counterexample, we use an inductive definition in the style of the global type LTS. This simplifies the proof of Endpoint Projection Theorem because there is a clear correspondence between the LTS of global and local types.

- (C) The standard way to write recursive processes in session types is with process definitions $\mathsf{def}\ D\ \mathsf{in}\ P$, where D is a list if definitions that may be invoked in P. The process definition construct makes the binding discipline of processes much harder to represent. To avoid this overhead we replaced this construct with a parameterised process reduction judgment $P \ \rightarrow_D \ P'$, where D is a list of definitions that may be used to define recursive behavior.

We now present the precise statement of Subject Reduction that has been mechanised.

Theorem 4.3 (Subject congruence and reduction)

1. *If* $P \equiv Q$ *and* $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \mathcal{Q};\ \Delta$ *then* $\Gamma \vdash_{\mathcal{D}} Q \triangleright_{\mathcal{C}} \mathcal{Q};\ \Delta$

2. *If* $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \mathcal{Q};\ \Delta$ *and* $\mathsf{coherent}(\Delta; \mathcal{Q})$ *as* Δ_0 *and* $P \ \rightarrow_D \ P'$ *then there exists* $\Delta_1, \Delta', \mathcal{Q}'$ *s.t* $\mathsf{coherent}(\Delta'; \mathcal{Q}')$ *as* Δ_1 *and* $\Gamma \vdash_{\mathcal{D}} P' \triangleright_{\mathcal{C}} \mathcal{Q}';\ \Delta'$ *and* $\Delta_0 = \Delta_1$ *or there exists* ℓ *s.t.* $\Delta_0 \xrightarrow{\ell} \Delta_1$

3. *If* $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\emptyset} \emptyset;\ \emptyset$ *and* $P \ \rightarrow_D \ P'$ *then* $\Gamma \vdash_{\mathcal{D}} P' \triangleright_{\emptyset} \emptyset;\ \emptyset$

The theorem states in three parts that: (1) A process remains well-typed by congruence rules; (2) A well-typed process remains well typed after reduction, possibly by a new environment whose coherence is given by a Δ_1

which can be reached by reduction from the Δ_0 that makes the initial environment coherent; (3) A process typed by the empty environment remains typable by the empty environment after reduction.

The remaining two paragraphs summarise decidability aspects of the linearity assumption for global types, and the coinductive equality relation used in three of the rules of the type system. The intention is that the assumptions the type system depends on for subject reduction, along with the definitions that occur in the typing rules, should all be decidable. We believe this to be the case for all definitions except for unstuckness. We discuss why and how this may be addressed in Chapter 6.

Linearity. Nearly all later work on multiparty session types has departed from explicit channels due to the complexity of linearity. We represent the Honda et al. definition of linearity in a faithful way in terms of interaction orderings and prove this equivalent to a coinductive definition that relies on observations that are characterised by simple inductive predicates. The coinductive definition consists of this single rule:

$$\frac{\mathsf{linearHead}(G) \quad \forall G' \in \mathsf{next}(G).\ \mathsf{linear}(G')}{\mathsf{linear}(G)}$$

The first premise $\mathsf{linearHead}(G)$ checks that the first interaction will occur before any other interactions sharing the same explicit channel. The second premise checks that the property is preserved by all continuations of the global type, unfolding $\mu \mathbf{t}.G$ when necessary.

Coinductive Equality. A standard coinductive presentation of subtyping for equi-recursive types can be found in Pierce [64]. Yoshida and Vasconcelos [80] adapted this definition to binary session types, using it as the basis for what it means to take the equi-recursive view: Allowing at any point the replacement of T with T', so long that $T \approx T'$. While mechanised decidability results exists for a coinductive declarative subtyping of equi-recursive simple types by Danielsson and Altenkirch in Agda [23] (based on Brandt and Henglein [15]), to the best of my knowledge, no such mechanised result exists for the coinductive equality of local types. We provide one and it is surprisingly short, taking up approximately 200 loc. A caveat is that Yoshida and Vasconcelos [80] also treat higher-order session types coinductively, as in $!k\langle T\rangle.T_0 \approx !k\langle T'\rangle.T_1$ implies $T_0 \approx T_1$ and $T \approx T'$. We are more restrictive here, imposing that equality on the higher-order types $T = T'$.

4.3 Overview of Papers

The results presented in this chapter, and part of this PhD Project, are comprised of the following three papers.

1. A published conference paper titled: "A Sound and Complete Projection for Global Types" [72], published in the 14th International Conference on Interactive Theorem Proving.

   This paper identifies a shortcoming regarding the projection of the μ-binder which is shared by all computable definitions of projection in the literature. We show that many sensible global types are ruled out as a consequence of defining projection by structural recursion. We give an overview of structurally recursive definitions of projections. Chapter 4.1 is an extended version of this presentation. In the paper, we address the limitation by presenting a sound and complete projection $\mathsf{proj}_{\mathsf{p}}(G)$. Based on the work of Castro-Perez et al., we specify the projection with the coinductive projection by Ghilezan et al., and relate it to the computable projection on inductive global types with the unravelling relation introduced by Castro-Perez et al..

   The paper can be found in Appendix A and the code is available at the following link.

2. A journal version of [72] with the same title, submitted to a special issue of the Journal of Automated Reasoning (JAR) for extended papers of ITP 2023. The paper is now in the second phase (reviews have been received).

   This journal version of [72] elaborates and revises the presentation of our projection function $\mathsf{proj}_{\mathsf{p}}(G)$. The presentation of inductive and coinductive types is extended. The definition of $\mathsf{proj}_{\mathsf{p}}(G)$ is simplified. More details about the proofs are presented, here including proof of termination of the decision procedure $\mathsf{projectable}_{\mathsf{p}}(G)$, soundness proof and completeness proof.

   The paper can be found in Appendix B and the code is available at the following link.

3. A conference paper, submitted to ESOP 2025, titled: "Multiparty Asynchronous Session Types: A Mechanised proof of Subject Reduction".

The paper presents a counterexample to the Subject Reduction Result of Honda et al.. We identify that the combination of using explicit channels along with the restrictive LTS semantics for global and local types types that Honda et al. introduced, breaks subject reduction. We address this with a new type system, faithful in its definition to the original system by Honda et al.. Changes made to the type system are detailed in Chapter 4.2. One change is replacing the projection function of Honda et al. with the projection function we introduce in Tirore et al. [72]. We mechanise a proof of subject reduction in Coq for this type system.

The paper can be found in Appendix C and the code is available at the following link.

Chapter 5

Related Work

Multiparty session types is a vast field, and an overview of it can be found in Coppo et al. [22]. Most results in this field have however not been mechanised. In this chapter we cover related work that satisfies the inclusion criterion of being mechanised results about session types. We do this by giving a detailed account of the two mechanised results of multiparty session types by Castro-Perez et al. [18] and Jacobs et al. [44]. We then touch lightly on mechanised results of binary session types.

5.1 Zooid

Most closely related to this PhD project is the work of Castro-Perez et al. [18]. They present Zooid, a domain-specific-language for writing correct-by-construction communicating processes. More precisely they define smart-constructors which allow one, in one go, to define an endpoint process along with a proof that it is well-typed, which verifies that the endpoint implements a local type. This coupling of a process, and a proof of well-typedness for this processes, is collectively called a Zooid term, which one writes in their domain-specific-language. Endpoint processes written in Zooid are Coq terms, and the code extraction features of Coq can be used to extract Ocaml code from these endpoint definitions that can then be executed. Their processes language is intended to define a single endpoint, and thus contains no parallel composition construct. We, on the other hand, include parallel composition in our process language along with delegation and session reception. They achieve parallelism by extracting the individ-

ual endpoints and executing the resulting OCaml code in parallel, which corresponds to the execution of a single session. Our process language, on the other hand, allows the execution of multiple ongoing sessions.

Their main result is a proof that the trace of well-typed processes conform to the trace of the local types they are typed against, and the global types the local types have been derived from by projection. This trace equivalence relies on a Projection theorem they prove for a semantics of global and local types by Deniélou and Yoshida [26]. This semantics is more flexible than ours, due to reducing not only local types by the local labels ζ defined in Appendix A and Appendix B, but also global types. That is, their transition relation on global types is a judgment of shape $G \xrightarrow{\zeta} G'$ which tracks actions, and not interactions.

They prove their trace result using coinductive global and local types, and they introduce a technique for reasoning about inductive global and local types in terms of these coinductive types. They do this by defining a coinductive relation they call unravelling. The definition of unravelling that we use is based on theirs, which we tighten to disallow non-contractive types as explained in Tirore et al. [72]. Using the idea of unravelling, they prove a computable definition of projection sound with respect to coinductive projection. We extend on their soundness result by giving a computable definition of projection which is not only sound, but also complete with respect to coinductive projection.

5.2 Multiparty GV

The work of Jacobs et al. [44] mechanise in Coq a deadlock freedom property for a linearly typed functional language with inter-thread communication specified by multiparty session types. Their result extends the binary session typed language GV [29, 76]. They initiate multiparty sessions with a n-ary fork operation, which spawns n new threads that together with the main thread forms a session of $n+1$ threads that interact by asynchronous communication. They mechanised their result in Coq using Iris [47].

The semantics of their language is given in terms of a small-step thread pool semantics, relating configurations of shape $(\vec{e}, h)$, where $\vec{e}$ is a vector of expressions corresponding to threads and h is a shared heap containing queues.

In their type system, they take inspiration from Scalas and Yoshida [67]

in the typing rule for the fork operation. Rather than using global types to specify the behavior of a session, as we do, they define a coinductive predicate on the typing environment, called consistency. This idea was introduced by Scalas and Yoshida [67] as a more general, as well as convenient, definition of well-formedness of typing environments. They relate this coinductive well-formedness condition to global types by proving that well-formedness, defined in terms of the projections of global types, is subsumed by consistency. We followed the approach of Honda et al. defining it in terms global type projections. Subject reduction requires this property to be re-established after an environment has reduced, and this preservation property was challenging to prove with our coherence definition. It required the construction of a global type derivation, based on the state of a process and the contents of a queue, and it further required reasoning about coinductive equality of local types, since the global type semantics can perform unfoldings. For future mechanised proofs of subject reduction, we therefore recommend the use of the consistency predicate as the well-formedness condition for typing environments.

The overall methodology of Jacobs et al. is based on Jacobs et al. [42], Rouvoet et al. [65] and separation logic [62, 63]. Like is standard in session types, they define two type systems. They introduce one for programs that have not yet performed any computation steps (static programs), and another type system for programs that are mid execution (runtime programs), and prove that the latter type systems subsumes the former. Their type system for static programs is defined in the standard way as an inductive judgment $\Gamma \vdash e : \tau$. Their type system for runtime programs is defined by structural recursion on the typed expression, generating a separation logic predicate $P(\Sigma)$. Here Σ is an environment containing local types, used in the typing of an expression. A separation logic predicate is defined using the connectives of separation logic, and this includes separating conjunction $P * Q$, which means that the predicate P holds for a set of resources, disjoint from the set of resources that satisfy Q. Using this connective, they conveniently ensure that session channels are used linearly.

5.3 Mechanisations of Binary Session Types

Jacobs et al. [43] study the topology of overlapping sessions and introduce a library for the connectivity graph. They use the library to prove

deadlock freedom for a binary session-typed functional language. Like the work for Jacobs et al. [44], their technique is based on separation logic. Thiemann [71] gave the first mechanised result of a functional language with binary session types, mechanising in Agda the language MicroSession which is a simplified subset of the language by Gay and Vasconcelos [29]. Rouvoet et al. [65] adapted the approach of Thiemann via separation logic, and proved type safety for a language inspired by GV. Also using separation logic, Hinrichsen et al. [37] mechanised results for a session typed language using semantic typing with binary session types with polymorphism and subtyping. Tassaroti et al. [70] use Iris in Coq to prove compiler correctness of a binary session-typed language. Sano et al [66] develop a technique to localise the reasoning about linear channel as a predicate $\mathsf{lin}(x, P)$, asserting that channel x is used linearly in P. Using this technique, they can ensure linear channel use with a structural (rather than linear) typing environment. In comparison, our type system is linear and splits the environment. They demonstrate in the Beluga proof assistant their technique on a subset of Classical Processes (CP) by Wadler [76]. Castro-Perez et al. [19] mechanise subject reduction for a variant of binary session types [80], using process replication $!P$ rather than recursive process definitions. They support delegation, but not recursive session types. They use a locally nameless representation for binders. They define a library for resource aware environments, which they use for their proofs. Coccone and Padovani [21] embed dependent binary session types into the linear π-calculus and mechanise in Agda a subject reduction proof for their process language. Gay et al. [30] investigate various definitions of duality for binary session types and mechanise some of their results in Agda. Ekici and Yoshida [27] study asynchronous session subtyping, proving in Coq the completeness of of a coinductive subtyping relation with respect to a streamlined version of the inductive negation of refinement by Ghilezan et al. [32]. Goto et al. [34] mechanise in Coq subject reduction and safety properties for a polymorphic session typing system for the π-calculus. They do not prove deadlock freedom. Inspired by the POPLmark challenge [10], a new benchmark has recently been proposed by Carbone et al. [16], of which I am co-author, targeting the mechanisation of concurrent systems in proof assistants.

Chapter 6

Discussion

6.1 Counterexample

We saw in Chapter 4.2 that the subject reduction result of Honda et al. did in fact not hold, and this was because of a counterexample that relied on explicit channels. We address this counterexample by introducing the coinductive predicate $\mathsf{unstuck}(G)$. This solution can be improved upon. Firstly, an inconvenience of unstuckness is that it is not clear how to decide this property. Unlike our decision procedures for linearity, coinductive equality and projectability, the termination of a decision procedure for unstuckness cannot be given in terms of enumerations. This is because the premise of unstuckness asserts that G' in $G \xrightarrow{l} G'$ is unstuck, and G' need not be in the enumeration of G. Secondly, the motivation of using explicit channels in global types is to allow more expressive specifications than implicit channels allow. The global types prohibited by unstuckness are sensible, yet not allowed, and this foes against the purpose of increasing expressivity using explicit channels.

We disallow these sensible global types because the global type semantics is restrictive. A more liberal semantics of global and local types, such as the one by Deniélou and Yoshida [26], and used by Castro-Perez et al. [18], reduces global types by the action ζ, rather than by an interaction ℓ. With this semantics, the environment in the counterexample would not be stuck. In future work, the counterexample may be addressed by adopting this semantics, or similar, thus eliminating the need for unstuckness.

6.2 Subtyping

The decomposition of a local type to its residual local type and queue types makes it possible to omit a subtyping rule for our type system. Our omission of a subtyping rule comes at the expense of reduced expressiveness: we cannot type a process offering more external choices than mentioned in its local type. This could pose a problem for an extension of this type system with merging projection, which relies on subtyping. Its omission however shows that subject reduction can be proved without the presence of a subtyping rule. Its omission is also very convenient when working in a proof assistant. This is because a subtyping rule is sub-structural, and thus always a case that must be considered in a proof by inversion. On paper, it is common practice to prove lemmas about the intensional shape of type derivations. An example from Honda et al. [40] is their Lemma 5.16, which shows that consecutive applications of the subtyping rule, due to transitivity of subtyping, can be truncated into a single application. The conclusion of this lemma could be represented in Coq by a predicate that inspects the shape of a typing derivation, ensuring that the produced proof contains no consecutive use of the subtyping rule. The intensional treatment of such proofs is nontrivial, and requires proofs about the shape of proofs. The standard induction tactic in Coq is insufficient for this purpose, therefore requiring one to use the dependent induction tactic, which is more complicated to use. It is not clear how such intensional reasoning about typing derivations should be mechanised effectively.

Hinrichsen et al. [37] mechanised subtyping for binary session types. They represent a subtyping relation using separation logic, and the absence of an inductive typing judgment means they do not run into the issues described above about intensional proofs.

6.3 Projection

The definition of our projection function was found by trial-and-error, using the properties of soundness and completeness to guide the definition, essentially aiming at mimicking the definition of coinductive projection in a decidable way. A consequence of this process is that the resulting definition is significantly different from, as well as more complicated, than the standard projections in the literature. This is solely due to the decision pro-

cedure $\mathsf{projectable}_{\mathsf{p}}(G)$ defined by well-founded recursion. The fact that our definition deviates from standard projection is not necessarily a detriment, but its complexity and ease of use, is. The complexity of the definition poses problems for mechanised and non-mechanised use of the definition, the former being forced to reason about mixed use of coinduction and induction to prove properties about projection, and the latter working in an unfamiliar setting without the safety rails of a proof assistant, which invites mistakes. Compared to the original presentation in Tirore et al. [72] which can be found in Appendix A, the definition has been simplified in the journal version of that paper which appears in Appendix B. Even still, there is room for improvement.

A promising direction of future work is to simplify our definition of projection in a way that relies solely on the familiar definition of projection by structural recursion, and coinductive equality of local types. Consider a set-based definition of projection, denoted by $G \upharpoonright\!\!\upharpoonright_{\mathsf{p}}$, where the equality check on projected branches has been removed, similar to translation, but unlike returning the projection of the first branch, as translation does, we return a set consisting of the projection on all branches:

$$(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j\in J}) \upharpoonright\!\!\upharpoonright_{\mathsf{p}} = \begin{cases} \ldots & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \ldots & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \bigcup_{i\in I} G_i \upharpoonright\!\!\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

Each local type, in the set produced by $G \upharpoonright\!\!\upharpoonright_{\mathsf{p}}$, corresponds to a specific sequence of branch choices through the global type, and we conjecture that testing whether they are all coinductively equal, corresponds to our projectability predicate $\mathsf{projectable}_{\mathsf{p}}(G)$. The projection that produces a single local type, would then be defined as:

$$G \downharpoonright_{\mathsf{p}} = \begin{cases} T_0 & \text{if } G \upharpoonright\!\!\upharpoonright_{\mathsf{p}} = \{T_0, .., T_n\} \text{ and } \forall i \in \{1, \ldots, n\}.\ T_0 \approx T_i \\ \text{undefined} & \text{otherwise} \end{cases} \tag{6.1}$$

This definition has the benefit of being defined in terms of $G \upharpoonright\!\!\upharpoonright_{\mathsf{p}}$ and $T \approx T'$, which use familiar definitions that may be more convenient to use in both mechanised and non-mechanised settings.

6.4 Merging

The projection function we introduce in Chapter 4.1 has been defined such that it corresponds to coinductive projection without merging. We opted for this simpler setting because this was in itself a nontrivial problem, and because the projection of Honda et al. did not use merging either. Most multiparty session type papers do however use merging in their definition of projection, and since the merging operation makes projection more expressive, it is a natural line of future work to incorporate this operation into our projection.

Merging, denoted by $T_0 \sqcup T_1$, is a partial operation which unions the branches of external choices, and recursively merges the continuations when labels coincide, as seen in the example below:

$$k\&\left\{\begin{array}{l}\texttt{Left} : T_0 \\ \texttt{Mid}\colon T_1\end{array}\right\} \sqcup k\&\left\{\begin{array}{l}\texttt{Mid} : T_2 \\ \texttt{Right}\colon T_3\end{array}\right\} = k\&\left\{\begin{array}{l}\texttt{Left} : T_0 \\ \texttt{Mid}\colon T_1 \sqcup T_2 \\ \texttt{Right}\colon T_3\end{array}\right\}$$

If one attempts to merge local types that differ beyond the labels in their external choices, the operation is undefined.

One way we could incorporate merging is to replace the coinductive equality check in Equation (6.1) with the merge operation, yielding the following definition of projection:

$$G\upharpoonright_{\mathsf{p}} = \begin{cases} \bigsqcup_{i=1}^{n} T_i & \text{if } G \Downarrow_{\mathsf{p}} = \{T_0, .., T_n\} \text{ and } \bigsqcup_{i=1}^{n} T_i \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases} \tag{6.2}$$

Note that merging has replaced our coinductive equality check, and not a syntactic equality check, it is therefore fair to expect the merge operation to satisfy properties such as being invariant to unfolding. This raises an interesting question about how the merge operation handles the μ-binder and recursion variable $\mathbf{t}$. We

Ghilezan et al., who introduce coinductive projection, define a merge operation on coinductive local types, which we will call coinductive merge. Given the properties we have proved for our projection function, it is natural to ask what the soundness and completeness properties are between coinductive merge and merging on inductive types, which we will call inductive merge. Answering this question in full is future work. We have however identified that the standard way inductive merge is defined is unsound.

The standard approach to merge binders and variables (See Remark 3.14 [31]), is done the following way:

$$\mu\mathbf{t}.T_0 \sqcup \mu\mathbf{t}.T_1 = \mu\mathbf{t}.(T_0 \sqcup T_1) \qquad \mathbf{t} \sqcup \mathbf{t} = \mathbf{t}$$

Using this definition, we can compute the merge of T_0 and T_1 below:

$$T_0 = \mu\mathbf{t}.k\& \left\{ \begin{array}{l} \texttt{Left} : \mathbf{t} \\ \texttt{Mid}\colon \mathbf{t} \end{array} \right\} \tag{6.3}$$

$$T_1 = \mu\mathbf{t}.k\& \left\{ \begin{array}{l} \texttt{Mid} : \mathbf{t} \\ \texttt{Right}\colon \mathbf{t} \end{array} \right\} \tag{6.4}$$

$$T_0 \sqcup T_1 = \mu\mathbf{t}.k\& \left\{ \begin{array}{l} \texttt{Left} : \mathbf{t} \\ \texttt{Mid}\colon \mathbf{t} \\ \texttt{Right}\colon \mathbf{t} \end{array} \right\} \tag{6.5}$$

To see why the local type in Equation (6.5) is an incorrect merging of the local types in Equation (6.3) and Equation (6.4), consider the branches of (6.5), which are `Left`, `Mid` and `Right`. Branch `Mid` correctly consists of $\mathbf{t}$ that points back to an external choice with labels `Left` from (6.3), `Right` from (6.4), and `Mid`. The incorrect `Left` branch does however also consist of $\mathbf{t}$. The label `Left` only occurs in (6.3), therefore once the `Left` branch is chosen, one should return to a state where it is not possible to choose the `Right` branch. We conjecture that soundness and completeness properties can be shown for a formulation of the inductive merge operation that merges (6.3) and (6.4) in the following way:

$$T_0 \sqcup T_1 = \mu\mathbf{t}.k\& \left\{ \begin{array}{l} \texttt{Left} : T_0 \\ \texttt{Mid}\colon \mathbf{t} \\ \texttt{Right}\colon T_1 \end{array} \right\}$$

6.5 Binders and Environments

In our mechanisation, binders in types and processes are represented with de Bruijn indices [24]. We generate the inductive definitions using Autosubst 2 [68]. On a subjective note, using this approach to represent and reason first about μ-types, and later processes, gave a continuity to the use of parallel substitutions that made parallel substitutions more intuitive to reason about over time.

Environment representation is affected by variable representation, and because de Bruijn indices are natural numbers, one may be tempted to represent the environment that types theses variables as an inductive list, using the variable as an index into the list. This however requires cumbersome reasoning about padding if environment entries are not contiguous. Instead, we opt for representing typing environments as association lists. This approach complements the use of parallel substitutions. The natural substitution lemma for a typing simply maps the parallel substitution function across the domain of the environment. By implementing a library for association lists based on the list comprehension functions map and filter, the proofs of commuting properties about environment operations reduce to commuting properties about comprehensions, for which the Coq library already has many theorems. To represent their environment, Jacobs et al. use in their mechanisation of multiparty GV the gmap library of the stdpp Coq library [2], which has the benefits of being both efficient and a syntactic equality that coincides with extensional equality. It would have been inconvenient for us to use gmap in this project because the association list representation is routinely exposed in order to prove properties about inserting entries and incrementing de Bruijn indices.

6.6 Libraries and Reusability

The Coq mechanisation of the results presented in this thesis have been developed over the span of three years. An effort of this magnitude is realistic for a PhD project focused on mechanisation. The large amount of boilerplate code required, and the many pitfalls one must avoid in writing this boilerplate, makes the barrier of entry high, and halts the progress of mechanised results in multiparty session types. The Concurrent Calculi Formalisation Benchmark [16] is a step in the right direction, which will help the community converge towards effective solutions to the frequently occurring challenges that are linear resource handling, coinduction, and scope extrusion of binders.

In the future, it would be interesting to extend on our subject reduction result by proving the formal guarantees that Honda et al. show: session fidelity, in-session deadlock freedom and type safety. It is however a challenge that the mechanisation is not easily reusable, and it would be interesting future work to factor parts of this mechanisation into generic reusable li-

braries, with the aim of reducing boilerplate code. Here we consider specifically our definitions and proofs regarding μ-binders and environments.

Papers of multiparty session types may use different formulations of global and local types, but by definition, if they are μ-types they will include $\mu\mathbf{t}$ and $\mathbf{t}$. In our mechanisation, we have shown many properties about the unfolding of μ-types, which makes them nearly as convenient to work with as coinductive types. It would be interesting to establish these properties for a library that is parametric on the formulation of global and local types.

Ensuring linear use of typing environments was also a challenge in the mechanisation, requiring reasoning about disjoint domains and proofs commutativity properties about environment operations such as insertion and removal. We represent environments as association lists and our implementation is parametric on the keys and values. Many of the commutativity properties are however stated for our specific instantiation of keys as de Bruijn indices. It would be interesting to develop a library that is compatible with de Bruijn indices and provides means of automation for reasoning about disjointness and commutativity of environment operations. This could possibly be part of a tool-chain based on Autosubst 2. It should in this context be mentioned that Castro-Perez et al. [19] introduce a library for reasoning about linear environments. We did not use this library because of our need to routinely expose our environment representation, the association list, to map over the keys, which was necessary to type a term on which a parallel substitution was applied.

6.7 Tools

The broader goal of this PhD project is to support the development of safe distributed systems by using the Coq proof assistant to verify formal results that underpin programming language features and external tools based on multiparty session types. One of these external tools is the Scribble language [79], in which one can write global protocols and project them to local protocols. An example of how the results of this PhD may support such tool development in the future, is an idea by Nobuko Yoshida, who suggested the future integration of our projection function into this tool.

The Scribble language has close ties to multiparty session types. A correspondence has been proved for a subset of the language called Featherweight Scribble [59], between global and local protocols in Featherweight

Scribble, and global and local types in multiparty session types. It would be interesting to relate the definitions in Featherweight Scribble with our mechanisation of multiparty session types, and mechanise such a correspondence theorem in Coq, possibly allowing parts of the tool implementation to be replaced with verified code extracted from Coq.

Bibliography

[1] Amazon Web Services. `https://aws.amazon.com`, accessed September 2024.

[2] Extended Standard Library for Coq, coq-stdpp. `https://gitlab.mpi-sws.org/iris/stdpp`, accessed September 2024.

[3] Google Cloud Platform. `https://cloud.google.com`, accessed September 2024.

[4] Impact of cloud outage. `https://www.lloyds.com/clouddown`. Accessed: Septemember 2024.

[5] MongoDB. `https://www.mongodb.com`, accessed September 2024.

[6] Mozilla Web Browser. `https://www.mozilla.org`, accessed September 2024.

[7] MySQL. `https://www.mysql.com`, accessed September 2024.

[8] Session Types in Programming Languages: A Collection of Implementations. `https://groups.inf.ed.ac.uk/abcd/session-implementations.html`, accessed September 2024.

[9] Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S. J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E. B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V. T., and Yoshida, N. Behavioral types in programming languages. *Found. Trends Program. Lang. 3*, 2-3 (2016), 95–230.

[10] Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G.,

Weirich, S., and Zdancewic, S. Mechanized metatheory for the masses: The poplmark challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings* (2005), J. Hurd and T. F. Melham, Eds., vol. 3603 of *Lecture Notes in Computer Science*, Springer, pp. 50–65.

[11] Bejleri, A., and Yoshida, N. Synchronous multiparty session types. In *Proceedings of PLACES* (2008), vol. 241 of *ENTCS*, Elsevier, pp. 3–33.

[12] Berger, U., and Setzer, A. Undecidability of equality for codata types. In *Coalgebraic Methods in Computer Science - 14th IFIP WG 1.3 International Workshop, CMCS 2018, Colocated with ETAPS 2018, Thessaloniki, Greece, April 14-15, 2018, Revised Selected Papers* (2018), C. Cîrstea, Ed., vol. 11202 of *Lecture Notes in Computer Science*, Springer, pp. 34–55.

[13] Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[14] Bettini, L., Coppo, M., D'Antoni, L., Luca, M. D., Dezani-Ciancaglini, M., and Yoshida, N. Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings* (2008), F. van Breugel and M. Chechik, Eds., vol. 5201 of *Lecture Notes in Computer Science*, Springer, pp. 418–433.

[15] Brandt, M., and Henglein, F. Coinductive axiomatization of recursive type equality and subtyping. vol. 33, pp. 63–81.

[16] Carbone, M., Castro-Perez, D., Ferreira, F., Gheri, L., Jacobsen, F. K., Momigliano, A., Padovani, L., Scalas, A., Tirore, D. L., Vassor, M., Yoshida, N., and Zackon, D. The concurrent calculi formalisation benchmark. In *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated*

Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings (2024), I. Castellani and F. Tiezzi, Eds., vol. 14676 of *Lecture Notes in Computer Science*, Springer, pp. 149–158.

[17] Carbone, M., Honda, K., and Yoshida, N. Structured communication-centred programming for web services. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings* (2007), R. D. Nicola, Ed., vol. 4421 of *Lecture Notes in Computer Science*, Springer, pp. 2–17.

[18] Castro-Perez, D., Ferreira, F., Gheri, L., and Yoshida, N. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In *Proceedings of PLDI* (2021), ACM, pp. 237–251.

[19] Castro-Perez, D., Ferreira, F., and Yoshida, N. EMTST: engineering the meta-theory of session types. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II* (2020), A. Biere and D. Parker, Eds., vol. 12079 of *Lecture Notes in Computer Science*, Springer, pp. 278–285.

[20] Church, A. A formulation of the simple theory of types. *J. Symb. Log. 5*, 2 (1940), 56–68.

[21] Ciccone, L., and Padovani, L. A dependently typed linear π-calculus in agda. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020* (2020), ACM, pp. 8:1–8:14.

[22] Coppo, M., Dezani-Ciancaglini, M., Padovani, L., and Yoshida, N. A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro,*

Italy, June 15-19, 2015, Advanced Lectures (2015), M. Bernardo and E. B. Johnsen, Eds., vol. 9104 of *Lecture Notes in Computer Science*, Springer, pp. 146–178.

[23] Danielsson, N. A., and Altenkirch, T. Subtyping, declaratively. In *Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings* (2010), C. Bolduc, J. Desharnais, and B. Ktari, Eds., vol. 6120 of *Lecture Notes in Computer Science*, Springer, pp. 100–118.

[24] de Bruijn, N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings) 75*, 5 (1972), 381–392.

[25] Demangeon, R., and Yoshida, N. On the expressiveness of multiparty sessions. In *Proceedings of FSTTCS* (2015), vol. 45 of *LIPIcs*, pp. 560–574.

[26] Deniélou, P., and Yoshida, N. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II* (2013), F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, Eds., vol. 7966 of *Lecture Notes in Computer Science*, Springer, pp. 174–186.

[27] Ekici, B., and Yoshida, N. Completeness of asynchronous session tree subtyping in coq. In *15th International Conference on Interactive Theorem Proving, ITP 2024, September 9-14, 2024, Tbilisi, Georgia* (2024), Y. Bertot, T. Kutsia, and M. Norrish, Eds., vol. 309 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 13:1–13:20.

[28] Fowler, S. An erlang implementation of multiparty session actors. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016* (2016), M. Bartoletti, L. Henrio, S. Knight, and H. T. Vieira, Eds., vol. 223 of *EPTCS*, pp. 36–50.

[29] Gay, S., and Vasconcelos, V. Linear type theory for asynchronous session types. *Journal of Functional Programming 20*, 1 (2010), 19–50.

[30] Gay, S. J., Thiemann, P., and Vasconcelos, V. T. Duality of session types: The final cut. In *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020* (2020), S. Balzer and L. Padovani, Eds., vol. 314 of *EPTCS*, pp. 23–33.

[31] Ghilezan, S., Jaksic, S., Pantovic, J., Scalas, A., and Yoshida, N. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Albegraic Methods in Programming 104* (2019), 127–173.

[32] Ghilezan, S., Pantovic, J., Prokic, I., Scalas, A., and Yoshida, N. Precise subtyping for asynchronous multiparty sessions. *ACM Trans. Comput. Log. 24*, 2 (2023), 14:1–14:73.

[33] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S. L., Mahboubi, A., O'Connor, R., Biha, S. O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings* (2013), S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., vol. 7998 of *Lecture Notes in Computer Science*, Springer, pp. 163–179.

[34] Goto, M. A., Jagadeesan, R., Jeffrey, A., Pitcher, C., and Riely, J. An extensible approach to session polymorphism. *Math. Struct. Comput. Sci. 26*, 3 (2016), 465–509.

[35] Hales, T. C., Adams, M., Bauer, G., Dang, D. T., Harrison, J., Hoang, T. L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T. T., Nguyen, T. Q., Nipkow, T., Obua, S., Pleso, J., Rute, J. M., Solovyev, A., Ta, A. H. T., Tran, T. N., Trieu, D. T., Urban, J., Vu, K. K., and Zumkeller, R. A formal proof of the kepler conjecture. *CoRR abs/1501.02155* (2015).

[36] Harrison, J. HOL light: An overview. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings* (2009), S. Berghofer,

T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674 of *Lecture Notes in Computer Science*, Springer, pp. 60–66.

[37] Hinrichsen, J. K., Louwrink, D., Krebbers, R., and Bengtson, J. Machine-checked semantic session typing. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021* (2021), C. Hritcu and A. Popescu, Eds., ACM, pp. 178–198.

[38] Honda, K., Vasconcelos, V. T., and Kubo, M. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP* (1998), vol. 1381 of *LNCS*, Springer, pp. 122–138.

[39] Honda, K., Yoshida, N., and Carbone, M. Multiparty asynchronous session types. In *Proceedings of POPL* (2008), ACM, pp. 273–284.

[40] Honda, K., Yoshida, N., and Carbone, M. Multiparty asynchronous session types. *Journal of the ACM 63*, 1 (2016), 9:1–9:67.

[41] Hunt, N., Bergan, T., Ceze, L., and Gribble, S. D. DDOS: taming nondeterminism in distributed systems. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013* (2013), V. Sarkar and R. Bodík, Eds., ACM, pp. 499–508.

[42] Jacobs, J., Balzer, S., and Krebbers, R. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang. 6*, POPL (2022), 1–33.

[43] Jacobs, J., Balzer, S., and Krebbers, R. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang. 6*, POPL (2022), 1–33.

[44] Jacobs, J., Balzer, S., and Krebbers, R. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proceedings of the ACM on Programming Languages 6*, ICFP (2022), 466–495.

[45] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D. A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an os kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 207–220.

[46] Kouzapas, D., Dardha, O., Perera, R., and Gay, S. J. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Sci. Comput. Program. 155* (2018), 52–75.

[47] Krebbers, R., Timany, A., and Birkedal, L. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 205–217.

[48] Lange, J., Ng, N., Toninho, B., and Yoshida, N. Fencing off go: liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017* (2017), G. Castagna and A. D. Gordon, Eds., ACM, pp. 748–761.

[49] Lange, J., Ng, N., Toninho, B., and Yoshida, N. A static verification framework for message passing in go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (2018), M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds., ACM, pp. 1137–1148.

[50] Leroy, X. A formally verified compiler back-end. *J. Autom. Reason. 43*, 4 (2009), 363–446.

[51] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. The ocaml system: Documentation and user's manual. *INRIA 3*, 42.

[52] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics.

In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008* (2008), S. J. Eggers and J. R. Larus, Eds., ACM, pp. 329–339.

[53] SIPLAN selection committee. Most Influential POPL Paper Award (2008). `https://www.sigplan.org/Awards`. Accessed: July 2024.

[54] The Coq development team. The Coq Proof Assistant. `https://coq.inria.fr`. Accessed: July 2024.

[55] The Lean development team. The Lean Proof Assistant. `https://lean-lang.org`. Accessed: July 2024.

[56] Milner, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 3 (1978), 348–375.

[57] Milner, R., Parrow, J., and Walker, D. A calculus of mobile processes, I and II. *Information and Computation 100*, 1 (Sept. 1992), 1–40,41–77.

[58] Neykova, R. Session types go dynamic or how to verify your python conversations. In *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013* (2013), N. Yoshida and W. Vanderbauwhede, Eds., vol. 137 of *EPTCS*, pp. 95–102.

[59] Neykova, R., and Yoshida, N. Featherweight scribble. In *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday* (2019), M. Boreale, F. Corradini, M. Loreti, and R. Pugliese, Eds., vol. 11665 of *Lecture Notes in Computer Science*, Springer, pp. 236–259.

[60] Ng, N., Yoshida, N., and Honda, K. Multiparty session C: safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings* (2012), C. A. Furia and S. Nanz, Eds., vol. 7304 of *Lecture Notes in Computer Science*, Springer, pp. 202–218.

[61] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[62] O'Hearn, P. W., and Pym, D. J. The logic of bunched implications. *Bull. Symb. Log. 5*, 2 (1999), 215–244.

[63] O'Hearn, P. W., Reynolds, J. C., and Yang, H. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings* (2001), L. Fribourg, Ed., vol. 2142 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[64] Pierce, B. C. *Types and programming languages*. MIT Press, 2002.

[65] Rouvoet, A., Poulsen, C. B., Krebbers, R., and Visser, E. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020* (2020), J. Blanchette and C. Hritcu, Eds., ACM, pp. 284–298.

[66] Sano, C., Kavanagh, R., and Pientka, B. Mechanizing session-types using a structural view: Enforcing linearity without linearity. *Proc. ACM Program. Lang. 7*, OOPSLA2 (2023), 374–399.

[67] Scalas, A., and Yoshida, N. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang. 3*, POPL (2019), 30:1–30:29.

[68] Stark, K., Schäfer, S., and Kaiser, J. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of CPP* (2019), ACM, pp. 166–180.

[69] Takeuchi, K., Honda, K., and Kubo, M. An interaction-based language and its typing system. In *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings* (1994), C. Halatsis, D. G. Maritsas, G. Philokyprou, and S. Theodoridis, Eds., vol. 817 of *Lecture Notes in Computer Science*, Springer, pp. 398–413.

[70] Tassarotti, J., Jung, R., and Harper, R. A higher-order logic for concurrent termination-preserving refinement. *CoRR abs/1701.05888* (2017).

[71] Thiemann, P. Intrinsically-typed mechanized semantics for session types. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019* (2019), E. Komendantskaya, Ed., ACM, pp. 19:1–19:15.

[72] Tirore, D. L., Bengtson, J., and Carbone, M. A sound and complete projection for global types. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland* (2023), A. Naumowicz and R. Thiemann, Eds., vol. 268 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 28:1–28:19.

[73] Tu, T., Liu, X., Song, L., and Zhang, Y. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019* (2019), I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds., ACM, pp. 865–878.

[74] van Glabbeek, R., Höfner, P., and Horne, R. Assuming just enough fairness to make session types complete for lock-freedom. In *Proceedings of LICS* (2021), IEEE, pp. 1–13.

[75] Wadler, P. A taste of linear logic. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings* (1993), A. M. Borzyszkowski and S. Sokolowski, Eds., vol. 711 of *Lecture Notes in Computer Science*, Springer, pp. 185–210.

[76] Wadler, P. Propositions as sessions. *Journal of Functional Programming 24*, 2–3 (2014), 384–418. Also: ICFP, pages 273–286, 2012.

[77] Yoshida, N., Deniélou, P., Bejleri, A., and Hu, R. Parameterised multiparty session types. In *Foundations of Software Science*

and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (2010), C. L. Ong, Ed., vol. 6014 of *Lecture Notes in Computer Science*, Springer, pp. 128–145.

[78] Yoshida, N., and Hou, P. Less is more revisit. *CoRR abs/2402.16741* (2024).

[79] Yoshida, N., Hu, R., Neykova, R., and Ng, N. The scribble protocol language. In *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers* (2013), M. Abadi and A. Lluch-Lafuente, Eds., vol. 8358 of *Lecture Notes in Computer Science*, Springer, pp. 22–41.

[80] Yoshida, N., and Vasconcelos, V. T. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In *Proceedings of the First International Workshop on Security and Rewriting Techniques, SecReT@ICALP 2006, Venice, Italy, July 15, 2006* (2006), M. Fernández and C. Kirchner, Eds., vol. 171 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 73–93.

[81] Yoshida, N., Zhou, F., and Ferreira, F. Communicating finite state machines and an extensible toolchain for multiparty session types. In *Fundamentals of Computation Theory - 23rd International Symposium, FCT 2021, Athens, Greece, September 12-15, 2021, Proceedings* (2021), E. Bampis and A. Pagourtzis, Eds., vol. 12867 of *Lecture Notes in Computer Science*, Springer, pp. 18–35.

Appendix A

A Sound and Complete Projection for Global Types

A Sound and Complete Projection for Global Types

Dawit Tirore
IT University of Copenhagen, Denmark

Jesper Bengtson
IT University of Copenhagen, Denmark

Marco Carbone
IT University of Copenhagen, Denmark

Abstract

Multiparty session types is a typing discipline used to write specifications, known as global types, for branching and recursive message-passing systems. A necessary operation on global types is projection to abstractions of local behaviour, called local types. Typically, this is a computable partial function that given a global type and a role erases all details irrelevant to this role.

Computable projection functions in the literature are either unsound or too restrictive when dealing with recursion and branching. Recent work has taken a more general approach to projection defining it as a coinductive, but not computable, relation. Our work defines a new computable projection function that is sound and complete with respect to its coinductive counterpart and, hence, equally expressive. All results have been mechanised in the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Type theory; Computing methodologies → Distributed programming languages; Theory of computation → Program verification

Keywords and phrases Session types, Mechanisation, Projection, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.28

Supplementary Material *Software (Coq Source Code)*: `https://github.com/Tirore96/projection` archived at `swh:1:dir:c4f38245d064fc65ed000d62ebd15826ef8da337`

Funding This work was supported by DFF-Research Project 1 Grant no. 1026-00385A, from The Danish Council for Independent Research for the Natural Sciences (FNU).

1 Introduction

Session types are types for abstracting the behaviour of communicating processes. First proposed by Honda et al. [15] for binary protocols, they specify the sequence of possible actions processes need to follow when sending and receiving messages over a channel. Session types provide a clear language for describing protocols that are guaranteed to not deadlock or contain communication errors, e.g., never receive an integer when expecting a boolean. A decade after their conception, Honda et al. [16] proposed a generalisation, called *multiparty session types*, that specifies how an arbitrary but fixed number of processes should interact with each other. Multiparty session types are based on the concept of *global types* which provide a global description of the multiparty protocol being abstracted. Recently, multiparty session types have gained interest from several communities, resulting in their integration into several mainstream programming languages [2].

Multiparty session types follow a precise approach to designing and implementing communicating processes: from global types that specify the protocols, we can automatically generate *local types*, the local specifications of the behaviour of each *role* in the protocol; then, each local type specification is (type) checked against the local code being written by the programmer. The automatic generation of local types from global types, called *projection*, is key for relating global types to implementations. Given a role, projection is an operation that erases the parts of the global type irrelevant for the role. When projection is defined the

output is a local type specifying the behaviour of this role. As an example, let us consider a global type where Carl can ask Dave to either go `Left` or `Right` over some channel k:

$$\mathsf{Carl} \to \mathsf{Dave} : k \left\{ \begin{array}{l} \texttt{Left} : \ \mathsf{Carl} \to \mathsf{Dave} : k'\langle\texttt{Int}\rangle.\mathsf{Alice} \to \mathsf{Bob} : k''\langle\texttt{Int}\rangle.\mathsf{end} \\ \texttt{Right} : \ \mathsf{Carl} \to \mathsf{Dave} : k'\langle\texttt{String}\rangle.\mathsf{Alice} \to \mathsf{Bob} : k''\langle\texttt{Int}\rangle.\mathsf{end} \end{array} \right\}$$

Above, if Carl chooses `Left`, he will also send an integer (`Int`) over some other channel k'; otherwise, he will send a string (`String`). No matter what branch Carl chooses, all roles must collectively follow the description of that branch.

Nested in both branches, there is a communication over k'' of an integer `Int` between Alice and Bob. The projections of Carl and Alice are:

$$\mathsf{Carl} : \quad k \oplus \left\{ \begin{array}{l} \texttt{Left} : !k'\langle\texttt{Int}\rangle.\mathsf{end} \\ \texttt{Right} : !k'\langle\texttt{String}\rangle.\mathsf{end} \end{array} \right\} \qquad\qquad \mathsf{Alice} : \quad !k''\langle\texttt{Int}\rangle.\ \mathsf{end}$$

Above, Carl makes a choice (denoted by $\oplus$), and then outputs on channel k' either something of type `Int` or something of type `String`. Alice is instead sending over channel k''. An important observation is that, since neither Alice nor Bob are informed of the choice made by Carl, their behaviour should be independent from Carl's choice. In fact, a *restriction* that projection usually imposes is that all those roles not participating to a branching communication behave the same on all branches.

In order to be able to express repetitive behaviour, global types (and local types) are usually equipped with recursion, expressed as μ*-types* [22]. For example, consider

$$\mu\mathbf{t}.\ \mathsf{Alice} \to \mathsf{Bob} : k\langle\texttt{String}\rangle.\ \mu\mathbf{t}'.\ \mathsf{Carl} \to \mathsf{Dave} : k' \left\{ \begin{array}{l} \texttt{Left} : \ \mathbf{t} \\ \texttt{Right} : \ \mathsf{Alice} \to \mathsf{Bob} : k\langle\texttt{String}\rangle.\mathbf{t}' \end{array} \right\} \quad (1)$$

The example above poses some questions on how projection should work. For Alice, should it be undefined since we cannot syntactically see her behaviour on the first branch? Or, can the projection first unfold on $\mathbf{t}$ and then generate a local type? We observe that the following global type is equivalent to (1) but does not violate our constraint on branches:

$$\mu\mathbf{t}.\ \mathsf{Alice} \to \mathsf{Bob} : k\langle\texttt{String}\rangle.\ \mathsf{Carl} \to \mathsf{Dave} : k' \{\texttt{Left} : \ \mathbf{t}, \quad \texttt{Right} : \ \mathbf{t}\} \quad (2)$$

Since both recursive global types (1) and (2) seem to specify the same behaviour, we would assume that Alice is projected to $\mu\mathbf{t}.\ !k\langle\texttt{String}\rangle.\ \mathbf{t}$, i.e., she repeatedly sends something of type `String`. The bad news is that the projection algorithms available in the literature do not allow global types like (1) to be projected while the equivalent type (2) can be projected.

The most common way of defining projection is as a structurally recursive partial function on global types, which we call *standard projection*. Recent work [13] defines projection as a coinductive relation on coinductive types, which intuitively are a complete (possibly infinite) unfolding of recursive protocols. Both approaches come with trade-offs. Standard projection is a computable procedure, which is necessary for multiparty session types to support decidable type checking, but it has limits as pointed out above. The coinductive approach is more general but, to the best of our knowledge, there are no equivalent computable algorithms available in the literature. The discrepancy between standard and coinductive projection was initially pointed out by Ghilezan et al. [13]. They correctly show that the canonical way partial projection treats the binders of μ-types causes standard projection to be undefined for some μ-types that have a coinductive projection, such as global type (1). In this paper, we define a procedure on μ-types that implements the projection on coinductive types.

Contributions and Structure. The main contribution of this paper is the definition of a computable projection function that is sound and complete with respect to a coinductive projection relation. All our proofs have been mechanised in the Coq [21] proof assistant[1].

We structure the paper as follows. Section 2 walks through existing variants of standard projection and their pitfalls. Section 3 introduces global and local coinductive types as well as a coinductive projection relation from the former to the latter. Section 4 introduces a projection function from global to local μ-types, proves that it is sound and complete with respect to its coinductive counterpart, and Section 5 proves that it is decidable. Section 6 describes key insights from our Coq mechanisation, Section 7 covers related and future work, and Section 8 concludes.

2 Global Types, Local Types, and Standard Projection

The purpose of this section is two-fold: introducing the syntax of global and local types and a walk through computable definitions of projection found in the literature.

Syntax. Let $\mathcal{P}$ be a set of roles (ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{t}$), $\mathcal{L}$ a totally ordered set of labels (ranged over by l), and $\mathcal{X}$ a set of recursion variables ranged over by $\mathbf{t}$.

▶ **Definition 1** (Inductive Types [17]). *Global types G^μ and local types T^μ are μ-types generated inductively by the following grammars, where U represents primitive types:*

$$G^\mu ::= \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \mid \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \mid \mu\mathbf{t}.G^\mu \mid \mathbf{t} \mid \mathsf{end}^\mu$$
$$T^\mu ::= !^\mu k\langle U\rangle.T^\mu \mid ?^\mu k\langle U\rangle.T^\mu \mid k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \mid k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J} \mid \mu\mathbf{t}.T^\mu \mid \mathbf{t} \mid \mathsf{end}^\mu$$

The type $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu$ denotes a communication between roles p_1 and p_2 via channel k of a message of type U, which then proceeds as G^μ. Similarly, the type $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J}$ denotes a communication between two roles where, given the set of indices J, role p_1 selects a branch with label l_i, and then proceeds as G_i^μ. Types $\mu\mathbf{t}.G^\mu$ and $\mathbf{t}$ model recursive protocols. Finally, end^μ denotes the successful termination of a protocol. A message type U is just a basic value type: extensions of this are irrelevant for the focus of this paper.

For local types, the type $!^\mu k\langle U\rangle.T^\mu$ outputs a message of type U over channel k, while its dual, $?^\mu k\langle U\rangle.T^\mu$ receives a message of type U over k. Types $k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J}$ and $k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J}$ implement branching where the former is the type of a process that internally selects a branch l_i and communicates it over channel k, while the latter is the type of a process that offers choices $l_1, \ldots, l_n$ (for $J = \{1, \ldots, n\}$ with $n \geq 1$) over channel k. We overload the type end^μ and use it also for local types.

We deal with recursive variables in a standard way and write capture-avoiding substitution as $G_1^\mu[G_2^\mu/\mathbf{t}]$. Moreover, types can be contractive: a μ-type G^μ (or T^μ) is contractive if, for any of its subexpressions with shape $\mu\mathbf{t}_0.\mu\mathbf{t}_1 ... \mu\mathbf{t}_n.\mathbf{t}$, the body $\mathbf{t}$ is not $\mathbf{t}_0$ [22]. We allow non-contractive μ-types and will in the next section show how to enforce contractiveness by requiring that a μ-type is related to a coinductive type.

Overview of projections. For each role, we use projection to relate global and local types. We start our overview with the projection proposed by Castro-Perez et al. [7] which can be found in Figure 1. The projection $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.\ G^\mu \upharpoonright^\mu_{\mathsf{p}}$ produces either a sending or a

[1] https://github.com/Tirore96/projection

$$(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} !^\mu k\langle U\rangle.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ ?^\mu k\langle U\rangle.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ G^\mu\upharpoonright^\mu_{\mathsf{p}} & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

$$(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J})\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} k \oplus^\mu \{l_j : (G^\mu_j\upharpoonright^\mu_{\mathsf{p}})\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ k \,\&^\mu \{l_j : (G^\mu_j\upharpoonright^\mu_{\mathsf{p}})\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ (G^\mu_1\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \text{ and } \forall i,j \in J.\ G^\mu_i\upharpoonright^\mu_{\mathsf{p}} = G^\mu_j\upharpoonright^\mu_{\mathsf{p}} \\ \bot & \textit{otherwise} \end{cases}$$

$$(\mu\mathbf{t}.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{guardedVar}(\mathbf{t}, G^\mu\upharpoonright^\mu_{\mathsf{p}}) \\ \mathsf{end}^\mu & \text{otherwise} \end{cases} \qquad \mathbf{t}\upharpoonright^\mu_{\mathsf{p}} = \mathbf{t} \qquad \mathsf{end}^\mu\upharpoonright^\mu_{\mathsf{p}} = \mathsf{end}^\mu.$$

$$\mathsf{guardedVar}(\mathbf{t}, G^\mu) = \begin{cases} \mathsf{guardedVar}(\mathbf{t}, G^\mu_1) & \text{if } G^\mu = \mu\mathbf{t}'.G^\mu_1 \\ \mathbf{t} \neq \mathbf{t}' & \text{if } G^\mu = \mathbf{t}' \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Figure 1 The *standard projection of G onto* p, written $G\upharpoonright^\mu_{\mathsf{p}}$ [7].

receiving action if the role p is equal to p_1 or p_2 respectively, otherwise the action is deleted. The projection of branching $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}\upharpoonright^\mu_{\mathsf{p}}$ works similarly but, when role p is not involved, all branches must project to exactly the same type. This requirement is known as *plain merge*. *Full merge*, used for example by Ghilezan et al. [13], is a more permissive operation which merges local types with distinct external choices. We discuss an extension of our work to full merge in Section 7. For recursion $\mu\mathbf{t}.G^\mu$, G^μ is projected only if the result is a contractive local type (checked by the $\mathsf{guardedVar}$ predicate). Finally, variable $\mathbf{t}$ and the type end^μ project directly to their local counterparts.

The use of $\mathsf{guardedVar}$ formally fixes a problem with the original projection [17] that could generate non-contractive types, which is unsound (informally fixed by forbidding non-contractive types). Alternatively, Demangeon and Yoshida [11] fix this issue by replacing the side condition with $G^\mu\upharpoonright^\mu_{\mathsf{p}} \neq \mathbf{t}$. However, all these projections invite the counterexample:

$$\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathsf{end}^\mu,\ l_2 : \mathbf{t}\}\upharpoonright^\mu_{\mathsf{p}} \tag{3}$$

which is undefined because the branch condition fails. Since p is not a role in the branch, the desired result of this projection should be $!^\mu k\langle U\rangle.\ \mathsf{end}^\mu$. Bejleri and Yoshida [5] solve this with a recursion condition testing participation in the body

$$(\mu\mathbf{t}.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p} \in G^\mu \\ \mathsf{end}^\mu \end{cases} \tag{4}$$

This function always generates contractive types, but the projection of

$$\mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U'\rangle.\ \mathbf{t}\upharpoonright^\mu_{\mathsf{p}} \tag{5}$$

incorrectly results in the local type $\mu\mathbf{t}.!^\mu k\langle U\rangle.\mathsf{end}^\mu$ rather than the desired $\mu\mathbf{t}.!^\mu k\langle U\rangle.\ \mathbf{t}$. Glabbeek et al. [27] fix it by adding a variable constraint to the recursion condition:

$$(\mu\mathbf{t}.G^\mu)\upharpoonright^\mu_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \\ \mathsf{end}^\mu & \text{if } \mathsf{p} \notin G^\mu \text{ and } \mu\mathbf{t}.G^\mu \text{ is closed} \end{cases} \tag{6}$$

This way, the projection in (5) correctly results in the type $\mu\mathbf{t}.!^{\mu}k\langle U\rangle.\mathbf{t}$. To the best of our knowledge, this is the most general and sound version of projection, but it still does not capture certain global types whose infinite unfolding is intuitively projectable. One such example is equivalent to (1), modulo renaming, from the introduction:

$$\mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t},\quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\} \upharpoonright^{\mu}_{\mathsf{p}} \tag{7}$$

Here, the branching condition fails because $\mathbf{t}$ is syntactically not the same type as $!^{\mu}k\langle U\rangle.\mathbf{t}'$. But how can we recognise that $\mathbf{t}$ and $!^{\mu}k\langle U\rangle.\mathbf{t}'$ are equivalent in this case? Our main insight is that standard projection can be performed in two steps: first, a boolean predicate tests projectability by unfolding μ-operators; and, when the check is passed, a translation function generates the local type by instead structurally recursing under the μ-operators. Checking projectability by unfolding μ-operators makes termination non-trivial and we explore this in Section 5. This approach lets us recognise (7) as projectable.

3 Projection on Coinductive Types

In this section, we define what an ideal projection is. The inductive definition of global types uses μ-types in order to represent infinite behaviour which, as shown by our examples, can create issues with projection. A possible solution to this issue is to get rid of μ-types and work with fully unfolded types (infinite trees). Originally, Honda et al. [17] suggested this approach informally. Later, Ghilezan et al. [13] turned this intuition into a version of global types which, instead of using an inductive definition, uses coinductive types. This had the drawback of projection not being computable. The goal of this section is to define coinductive types, a way to relate them to inductive types, and then a definition of projection without μ-types. Although we do not compute projections of coinductive types, we use them as a specification of how a correct projection should behave.

Syntax. We start by giving the coinductive definition of both global and local types.

▶ **Definition 2** (Coinductive Types). *The syntax of coinductive global and local types, denoted as G^{ν} and T^{ν} respectively, is coinductively defined as:*

$$\begin{aligned} G^{\nu} &::= \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu} \mid \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J} \mid \mathsf{end}^{\nu} \\ T^{\nu} &::= !^{\nu}k\langle U\rangle.T^{\nu} \mid ?^{\nu}k\langle U\rangle.T^{\nu} \mid k \oplus^{\nu} \{l_j : T^{\nu}_j\}_{j\in J} \mid k \,\&^{\nu}\, \{l_j : T^{\nu}_j\}_{j\in J} \mid \mathsf{end}^{\nu} \end{aligned}$$

Coinductive types can be infinite but regular coinductive types can be finitely represented. A regular coinductive type has a finite set of distinct subterms [20] meaning that it must be circularly defined and have repeating structure if it is infinitely large. This makes it possible to store a regular coinductive type in, e.g., computer memory, or represent it as a μ-type.

In order to reason effectively about μ-types and their coinductive counterparts we need a means to relate the two. We follow the style of Castro-Perez et al. [7], using an unravelling relation $\mathcal{R}$, formally defined as:

▶ **Definition 3** (Unravelling). Unravelling, *for both global and local types, and denoted by* $G^\mu \mathrel{\mathcal{R}} G^\nu$ *and* $T^\mu \mathrel{\mathcal{R}} T^\nu$ *respectively, is defined by the following rules:*

$$\frac{}{\mathsf{end}^\mu \mathrel{\mathcal{R}} \mathsf{end}^\nu} \qquad \frac{G^\mu[\mu\mathbf{t}.G^\mu/\mathbf{t}] \mathrel{\mathcal{R}} G^\nu}{\mu\mathbf{t}.G^\mu \mathrel{\mathcal{R}} G^\nu} \qquad \frac{G^\mu \mathrel{\mathcal{R}} G^\nu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \mathrel{\mathcal{R}} \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu}$$

$$\frac{\forall j \in J.\ G_j^\mu \mathrel{\mathcal{R}} G_j^\nu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \mathrel{\mathcal{R}} \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G_j^\nu\}_{j\in J}} \qquad \frac{\forall j \in J.\ T_j^\mu \mathrel{\mathcal{R}} T_j^\nu}{k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \mathrel{\mathcal{R}} k \oplus^\nu \{l_j : T_j^\nu\}_{j\in J}}$$

$$\frac{T^\mu \mathrel{\mathcal{R}} T^\nu}{!^\mu k\langle U\rangle.T^\mu \mathrel{\mathcal{R}} !^\nu k\langle U\rangle.T^\nu} \qquad \frac{T^\mu \mathrel{\mathcal{R}} T^\nu}{?^\mu k\langle U\rangle.T^\mu \mathrel{\mathcal{R}} ?^\nu k\langle U\rangle.T^\nu} \qquad \frac{\forall j \in J.\ (T_j^\mu \mathrel{\mathcal{R}} T_j^\nu)}{k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J} \mathrel{\mathcal{R}} k \,\&^\nu\, \{l_j : T_j^\nu\}_{j\in J}}$$

The unravelling relation is defined using both inductive and coinductive inference rules, where we use single lines for inductive rules and double lines for coinductive ones. A coinductive derivation may be circular and discharged by referring to a previous identical part of the inference tree whereas inductive leaves are discharged using an inductive base-case rule in the standard manner, which in our case are the rules relating end^μ and end^ν. The reason for this split is that if the μ-operator could be unravelled using a coinductive rule [unfold^ν] then we could relate any non-contractive μ-type to any coinductive type G^ν.

$$\text{Incorrect rule: } \frac{G[\mu\mathbf{t}.G] \mathrel{\mathcal{R}} G^\nu}{\mu\mathbf{t}.G \mathrel{\mathcal{R}} G^\nu}\ [\mathsf{unfold}^\nu] \qquad \text{Unwanted derivation: } \frac{\dfrac{}{\mu\mathbf{t}.t \mathrel{\mathcal{R}} G^\nu}}{\mu\mathbf{t}.t \mathrel{\mathcal{R}} G^\nu}\ [\mathsf{unfold}^\nu] \tag{8}$$

Castro-Perez et al. have a rule like [unfold^ν] and they solve this problem by requiring that all μ-types are contractive. We make this side condition redundant by making [unfold^ν] inductive and we have found that this simplifies our proofs. This is because the usual two conditions on μ-types, namely closedness and contractiveness, are captured by unravelling.

▶ **Proposition 4.** *G^μ is closed and contractive iff there exists G^ν such that $G^\mu \mathrel{\mathcal{R}} G^\nu$*

Proof. The direction ($\Longleftarrow$) is harder than the other. We prove it by contradiction, assuming both an inductive definition of non-contractiveness and $G^\mu \mathrel{\mathcal{R}} G^\nu$. ◀

The mixing of inductive and coinductive inference rules is non-standard and in Section 6 we show concretely how to formally define such inference systems. For now, we show an example of how to relate an inductive and coinductive global type by $\mathcal{R}$.

▶ **Example 5.** Consider the unravelling of

$$\mu\mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\}$$

One branch is a recursion variable and the other is end indicating we will need both inductive and coinductive rules to close the derivation. We show this inductive global type unravels to

$$G^\nu := \mathsf{r} \xrightarrow{\nu} \mathsf{s} : k\{l_1 : G^\nu,\ l_2 : \mathsf{end}^\nu\}$$

This is shown by the derivation below

$$\cfrac{\cfrac{\cfrac{}{\mu\mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\} \mathrel{\mathcal{R}} G^\nu} \qquad \cfrac{}{\mathsf{end}^\mu \mathrel{\mathcal{R}} \mathsf{end}^\nu}}{\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mu\mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\},\ l_2 : \mathsf{end}^\mu\} \mathrel{\mathcal{R}} G^\nu}}{\longrightarrow \mu\mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k\{l_1 : \mathbf{t},\ l_2 : \mathsf{end}^\mu\} \mathrel{\mathcal{R}} G^\nu} \tag{9}$$

where the arrow marks the cycle that solves the coinductive part of the proof. Visually, the arrow must pass a double line for the proof to be valid.

$$\frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu})} \qquad \frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \forall j.\ \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu}_j)}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J})} \qquad \frac{\mathsf{guarded}^{\mu}_{\mathsf{p}}(G[\mu\mathbf{t}.G/t])}{\mathsf{guarded}^{\mu}_{\mathsf{p}}(\mu\mathbf{t}.G)}$$

$$\frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})}{\mathsf{partOf}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu})} \qquad \frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \exists j \in J.\ \mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu}_j)}{\mathsf{partOf}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J})} \qquad \frac{\mathsf{partOf}^{\mu}_{\mathsf{p}}(G[\mu\mathbf{t}.G/t])}{\mathsf{partOf}^{\mu}_{\mathsf{p}}(\mu\mathbf{t}.G)}$$

Figure 2 Definitions of predicates $\mathsf{guarded}^{\nu}$, $\mathsf{guarded}^{\mu}$, partOf^{ν}, and partOf^{μ}. The $\mathsf{guarded}^{\mu}$ and partOf^{μ} predicates additionally have identical rules to their $\mathsf{guarded}^{\nu}$ and partOf^{ν} counterparts, except for being defined for G^{μ} and not G^{ν} – these rules have been elided.

$$\frac{G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}{\begin{array}{c}\mathsf{p} \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} \\ !^{\nu}k\langle U\rangle.T^{\nu}\end{array}}\ [\mathsf{M1}\upharpoonright^{\nu}] \qquad \frac{\mathsf{p} \neq \mathsf{p}_1 \quad G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}{\begin{array}{c}\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p} : k\langle u\rangle.G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} \\ ?^{\nu}k\langle U\rangle.T^{\nu}\end{array}}\ [\mathsf{M2}\upharpoonright^{\nu}] \qquad \frac{\neg\mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})}{G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} \mathsf{end}^{\nu}}\ [\mathsf{End}\upharpoonright^{\nu}]$$

$$\frac{\mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \qquad \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu}) \qquad G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}\ [\mathsf{M}\upharpoonright^{\nu}] \qquad \frac{\forall j.\ G^{\nu}_j \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}_j}{\begin{array}{c}\mathsf{p} \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J} \upharpoonright^{\nu}_{\mathsf{p}} \\ k \oplus^{\nu} \{l_j : T^{\nu}_j\}_{j\in J}\end{array}}\ [\mathsf{B1}\upharpoonright^{\nu}]$$

$$\frac{J \neq \{\} \qquad \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \qquad \begin{array}{r}\forall j.\ G^{\nu}_j \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu} \wedge \\ \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu}_j)\end{array}}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}}\ [\mathsf{B}\upharpoonright^{\nu}] \qquad \frac{\mathsf{p} \neq \mathsf{p}_1 \qquad \forall j,\ G^{\nu}_j \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}_j}{\begin{array}{c}\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p} : k\{l_j : G^{\nu}_j\}_{j\in J} \upharpoonright^{\nu}_{\mathsf{p}} \\ k \,\&^{\nu}\, \{l_j : T^{\nu}_j\}_{j\in J}\end{array}}\ [\mathsf{B2}\upharpoonright^{\nu}]$$

Figure 3 The projection on coinductive types, denoted $G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} T^{\nu}$, is defined by coinductive rules.

In order to define projection from coinductive global types to coinductive local types, we require the two auxiliary predicates $\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})$ and $\mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})$. The former asserts that p appears in all branches of G^{ν} at finite depth, and the latter asserts that p occurs somewhere in G^{ν} at finite depth. To reason about finite depth these predicates are inductively defined. We also define similar predicates $\mathsf{guarded}^{\mu}_{\mathsf{p}}(G^{\mu})$ and $\mathsf{partOf}^{\mu}_{\mathsf{p}}(G^{\mu})$ for inductive global types G^{μ}. All four predicates are defined in Fig. 2.

The rules for projection are presented in Figure 3. Rules $[\mathsf{M1}\upharpoonright^{\nu}]$, $[\mathsf{M2}\upharpoonright^{\nu}]$, $[\mathsf{B1}\upharpoonright^{\nu}]$, and $[\mathsf{B2}\upharpoonright^{\nu}]$ handle the cases where a projected role p takes part in communication or branching. Note that our projection allows sender and receiver in a communication to be equal. This case is a special case of rule $[\mathsf{M1}\upharpoonright^{\nu}]$. The rules $[\mathsf{M}\upharpoonright^{\nu}]$, $[\mathsf{B}\upharpoonright^{\nu}]$, and $[\mathsf{End}\upharpoonright^{\nu}]$ handle the cases where p does not take part. In these cases, in order for projection to continue, p must occur in all possible future branches, otherwise the projection maps to end. These rules are similar to those given by Castro-Perez et al. [7] as well as Jacobs et al. [19].

Guardedness, enforced by the predicate $\mathsf{guarded}^{\nu}_{\mathsf{p}}$ in the $[\mathsf{M}\upharpoonright^{\nu}]$ and $[\mathsf{B}\upharpoonright^{\nu}]$ rules, is necessary in order to avoid unwanted derivations similar to that for unravelling in (8).

▶ **Example 6.** We can now use unravelling and coinductive projection to relate the global type $\mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$ seen in (7) with $\mu\mathbf{t}.!^{\mu}k\langle U\rangle.\mathbf{t}$. They respectively unravel to

$$G^{\nu} := \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^{\nu}, \quad l_2 : G^{\nu}\}$$
$$E^{\nu} :=!^{\nu}k\langle U\rangle.\ E^{\nu}$$

We can now derive $G^\nu \downarrow^\nu_{\mathsf{p}} E^\nu$ by $[\mathsf{M1}\downarrow^\nu]$, $[\mathsf{B}\downarrow^\nu]$ followed by $[\mathsf{M1}\downarrow^\nu]$ and mark a cycle to the conclusion $G^\nu \downarrow^\nu_{\mathsf{p}} E^\nu$. This precisely justifies why we wish to project the inductive global type in (7) over role p to the local type $\mu\mathbf{t}.!^\mu k\langle U\rangle.\mathbf{t}$.

4 Projection on Inductive Types: Soundness and Completeness

The coinductive projection predicate $\downarrow^\nu$ represents the specification of an ideal projection from coinductive global types to coinductive local types. In this section, we present a projection function proj on μ-types that is sound and complete with respect to the $\downarrow^\nu$ projection predicate. We extend on previous work by Castro-Perez et al. [7] whose projection function is shown to be sound but not complete.

▶ **Definition 7** (proj). *The function* $\mathsf{proj} : \mathcal{P} \to G^\mu \rightharpoonup T^\mu$, *written* $\mathsf{proj}_{\mathsf{p}}(G^\mu)$, *is the projection of the global* μ*-type* G^μ *with respect to the role* p *and is defined as:*

$$\mathsf{proj}_{\mathsf{p}}(G^\mu) = \begin{cases} \mathsf{trans}_{\mathsf{p}}(G^\mu) & \textit{if } \mathsf{projectable}_{\mathsf{p}}(G^\mu) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Our projection function features two auxiliary entities, namely the translation function trans and the predicate $\mathsf{projectable}$ which precisely separate the generation of the local type and the check for projectability respectively.

▶ **Definition 8** (trans). *The function* $\mathsf{trans} : \mathcal{P} \to G^\mu \to T^\mu$ *is identical to the function* $\lfloor^\mu$ *(see Figure 1) except for the branching case, defined as:*

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}) \quad = \quad \begin{cases} k \oplus^\mu \{l_j : \mathsf{trans}_{\mathsf{p}}(G^\mu_j)\}_{j\in J} & \textit{if } \mathsf{p} = \mathsf{p}_1 \textit{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ k \,\&^\mu\, \{l_j : \mathsf{trans}_{\mathsf{p}}(G^\mu_j)\}_{j\in J} & \textit{if } \mathsf{p} = \mathsf{p}_2 \textit{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G^\mu_1) & \textit{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

The only difference from Definition 2 is that the removal of the branching condition has made trans total. These conditions are checked by the $\mathsf{projectable}$ predicate and the challenging part of implementing proj is proving decidability of this predicate.

▶ **Definition 9** ($\mathsf{projectable}$). *The predicate* $\mathsf{projectable}_{\mathsf{p}}(G^\mu)$ *states that the global* μ*-type* G^μ *is projectable with respect to the role* p *and is defined as:*

$$\mathsf{projectable}_{\mathsf{p}}(G^\mu) = \exists G^\nu\, T^\nu.\ G^\mu\ \mathcal{R}\ G^\nu \ \wedge\ \mathsf{trans}_{\mathsf{p}}(G^\mu)\ \mathcal{R}\ T^\nu \ \wedge\ G^\nu \downarrow^\nu_{\mathsf{p}} T^\nu$$

The predicate states that the μ-types G^μ and $\mathsf{trans}_{\mathsf{p}}(G^\mu)$ are related by unravelling to some coinductive types G^ν and T^ν respectively, and that T^ν is the coinductive projection of G^ν with respect to p. This predicate is decidable and we detail why in Section 5.

Soundness. Proving that proj is sound with respect to $\downarrow^\nu$ is relatively straightforward.

▶ **Theorem 10.** *If* $\mathsf{proj}_{\mathsf{p}}(G^\mu)$ *is defined then there exist coinductive types* G^ν *and* T^ν *such that* $G^\mu\ \mathcal{R}\ G^\nu$, $\mathsf{proj}_{\mathsf{p}}(G^\mu)\ \mathcal{R}\ T^\nu$ *and* $G^\nu \downarrow_p T^\nu$.

Proof. Follows directly from the definition of proj and $\mathsf{projectable}$ by setting G^ν and T^ν to their corresponding types obtained from $\mathsf{projectable}$. ◀

$$\begin{array}{ccc} G^\nu & \xrightarrow{\downarrow^\nu} & T^\nu \\ \mathcal{R}\uparrow & & \mathcal{R}\uparrow \\ G^\mu & \xrightarrow{\mathsf{proj}} & T^\mu \end{array}$$

Completeness. For completeness, we require an auxiliary operation $\mathsf{unfold}(\cdot)$ on global and local μ-types that unfolds all binders until an interaction prefix or end are exposed.

$$\mathsf{unfold}(G^\mu) = \mathsf{unfold\_once}^{|G^\mu|}(G^\mu) \qquad \mathsf{unfold\_once}(G^\mu) = \begin{cases} G_1^\mu[\mu\mathbf{t}.G_1^\mu/t], & \text{if } G^\mu = \mu\mathbf{t}.G_1^\mu \\ G^\mu & \text{otherwise} \end{cases}$$

Above, $|G^\mu|$ is the μ-height of G^μ, i.e., the number of initial consecutive binders found in G^μ. Here, f^n denotes repeated function composition. For example, $|\mu\mathbf{t}.\mathsf{end}| = 1$ and $|\mathsf{p} \xrightarrow{\mu} \mathsf{p}' : k\langle U\rangle.\mu\mathbf{t}.\ \mathsf{end}| = 0$. We overload unfolding with $\mathsf{unfold}(T^\mu)$ and $|T^\mu|$, for having the corresponding meaning on local types.

$$\begin{array}{ccc} G^\nu & \xrightarrow{\downarrow^\nu} & T^\nu \\ \mathcal{R}\big\downarrow & & \mathcal{R}\big\downarrow \\ G^\mu & \xrightarrow{\mathsf{proj}} & T^\mu \end{array}$$

In order to show completeness of proj with respect to $\downarrow^\nu$, we need to show that if $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$ and $G^\mu \ \mathcal{R}\ G^\nu$ then $\mathsf{proj}_\mathsf{p}(G^\mu)$ is defined and $\mathsf{proj}_\mathsf{p}(G^\mu) \ \mathcal{R}\ T^\nu$. We prove this by showing that $\mathsf{trans}_\mathsf{p}(G^\mu)$ unravels to $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu))$; then, we show $\mathsf{trans}_\mathsf{p}(G^\mu) = \mathsf{proj}_\mathsf{p}(G^\mu)$ and $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = T^\nu$. The function $\mathsf{tocoind}$ is defined as

▶ **Definition 11** ($\mathsf{tocoind}$). *The corecursive function* $\mathsf{tocoind} : T^\mu \to T^\nu$ *is defined as*

$$\mathsf{tocoind}(T^\mu) = \begin{cases} !^\nu k\langle U\rangle.\mathsf{tocoind}(T^\mu) & \textit{if } \mathsf{unfold}(T^\mu) = !^\mu k\langle U\rangle.T^\mu \\ ?^\nu k\langle U\rangle.\mathsf{tocoind}(T^\mu) & \textit{if } \mathsf{unfold}(T^\mu) = ?^\mu k\langle U\rangle.T^\mu \\ k \oplus^\nu \{l_j : \mathsf{tocoind}(T_j^\mu)\}_{j\in J} & \textit{if } \mathsf{unfold}(T^\mu) = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \\ k \ \&^\nu \{l_j : \mathsf{tocoind}(T_j^\mu)\}_{j\in J} & \textit{if } \mathsf{unfold}(T^\mu) = k \ \&^\mu \{l_j : T_j^\mu\}_{j\in J} \\ \mathsf{end}^\nu & \textit{otherwise} \end{cases}$$

Note that, $T^\mu \ \mathcal{R}\ \mathsf{tocoind}(T^\mu)$ does not always hold, as $\mathcal{R}$ is only defined for closed and contractive T^μ. However, for closed global types, trans does unravel to a coinductive type.

▶ **Lemma 12** ($\mathsf{Unraveling\ of\ trans}$). *If* G^μ *is closed then* $\mathsf{trans}_\mathsf{p}(G^\mu) \ \mathcal{R}\ \mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu))$.

Proof. Since G^μ is closed, we know that $\mathsf{trans}_\mathsf{p}(G^\mu)$ is closed. Moreover, the image of $\mathsf{trans}_\mathsf{p}$ is always contractive. For any closed and contractive local type T^μ, we know that $T^\mu \ \mathcal{R}\ \mathsf{tocoind}(T^\mu)$, by coinduction on $\mathcal{R}$. In particular this holds for $\mathsf{trans}_\mathsf{p}(G^\mu)$. ◀

▶ **Lemma 13** ($\mathsf{trans\ as\ projection}$). *If* $G^\mu \ \mathcal{R}\ G^\nu$ *and* $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$ *then* $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = T^\nu$.

Proof. By coinduction using the candidate relation $\{(\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)), T^\nu) \mid G^\nu \downarrow^\nu_\mathsf{p} T^\nu \wedge G^\mu \ \mathcal{R}\ G^\nu\}$. From $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$, derive $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu) \vee T^\nu = \mathsf{end}^\nu$. The first case is proven by induction on $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu)$; for the second we know from $G^\nu \downarrow^\nu_\mathsf{p} \mathsf{end}^\nu$ and $G^\mu \ \mathcal{R}\ G^\nu$ that $\neg\mathsf{partOf}^\nu_\mathsf{p}(G^\mu)$ and hence $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{end}^\nu$. ◀

From these Lemmas, completeness follows immediately.

▶ **Theorem 14.** *If* $G^\nu \downarrow^{co}_p T^\nu$ *and* $G^\mu \ \mathcal{R}\ G^\nu$ *then* $\mathsf{proj}_\mathsf{p}(G^\mu)$ *is defined and* $\mathsf{proj}_\mathsf{p}(G^\mu) \ \mathcal{R}\ T^\nu$.

Proof. From $G^\mu \ \mathcal{R}\ G^\nu$, we know using Proposition 4 that G^μ is closed. Applying Lemma 12, we have that $\mathsf{trans}_\mathsf{p}(G^\mu) \ \mathcal{R}\ \mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu))$. Finally, from Lemma 13, we have that $\mathsf{trans}_\mathsf{p}(G^\mu) \ \mathcal{R}\ T^\nu$. It thus holds that $\mathsf{projectable}_\mathsf{p}(G^\mu)$, so $\mathsf{proj}_\mathsf{p}(G^\mu)$ is defined and $\mathsf{trans}_\mathsf{p}(G^\mu) = \mathsf{proj}_\mathsf{p}(G^\mu)$ letting us conclude $\mathsf{proj}_\mathsf{p}(G^\mu) \ \mathcal{R}\ T^\nu$. ◀

$$\frac{\mathsf{unfold}(G^\mu) \Downarrow^\mu_\mathsf{p} \ \mathsf{unfold}(T^\mu)}{G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu} \ [\mathsf{Unf}\downarrow^\mu] \qquad \frac{\mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \quad \mathsf{guarded}^\mu_\mathsf{p}(G^\mu) \quad G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \Downarrow^\mu_\mathsf{p} \ T^\mu} \ [\mathsf{M}\downarrow^\mu]$$

$$\frac{G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu}{\mathsf{p} \xrightarrow{\mu} \mathsf{p}_1 : k\langle U\rangle.G^\mu \Downarrow^\mu_\mathsf{p} \ !^\mu k\langle U\rangle.T^\mu} \ [\mathsf{M1}\downarrow^\mu] \qquad \frac{\forall j.\ G^\mu_j \downarrow^\mu_\mathsf{p} \ T^\mu_j}{\mathsf{p} \xrightarrow{\mu} \mathsf{p}_1 : k\{l_j : G^\mu_j\}_{j\in J} \Downarrow^\mu_\mathsf{p} \ k \oplus^\mu \{l_j : T^\mu_j\}_{j\in J}} \ [\mathsf{B1}\downarrow^\mu]$$

$$\frac{\mathsf{p} \neq \mathsf{p}_1 \quad G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p} : k\langle U\rangle.G^\mu \Downarrow^\mu_\mathsf{p} \ ?^\mu k\langle U\rangle.T^\mu} \ [\mathsf{M2}\downarrow^\mu] \qquad \frac{\mathsf{p} \neq \mathsf{p}_1 \quad \forall j.\ G^\mu_j \downarrow^\mu_\mathsf{p} \ T^\mu_j}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p} : k\{l_j : G^\mu_j\}_{j\in J} \Downarrow^\mu_\mathsf{p} \ k \ \&^\mu \{l_j : T^\mu_j\}_{j\in J}} \ [\mathsf{B2}\downarrow^\mu]$$

$$\frac{\neg\mathsf{partOf}^\mu_\mathsf{p}(G^\mu) \qquad \mathsf{Unravels}(G^\mu)}{G^\mu \Downarrow^\mu_\mathsf{p} \ \mathsf{end}_c} \ [\mathsf{End}\downarrow^\mu] \qquad \frac{J \neq \{\} \quad \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \quad \forall j.\ G^\mu_j \downarrow^\mu_\mathsf{p} \ T^\mu \wedge \mathsf{guarded}^\mu_\mathsf{p}(G^\mu_j)}{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J} \Downarrow^\mu_\mathsf{p} \ T^\mu} \ [\mathsf{B}\downarrow^\mu]$$

Figure 4 *Intermediate projection* on inductive types, written as $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$.

5 Deciding Projectability

In this section, we show that projectable is decidable. We do this in two steps: first, we define the *intermediate projection* $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ and show that it is sound and complete with respect to our coinductive projection; second, given a pair (G^μ, T^μ), we construct a graph and show that deciding $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ can be reduced to checking properties of that graph.

An Intermediate Projection. The rules defining $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$, presented in Figure 4, are similar to those for coinductive projection, but also enforce the unfolding operation unfold on both μ-types. Initially, the only applicable rule is $[\mathsf{Unf}\downarrow^\mu]$, which unfolds μ-types. Then, the rules inspired by coinductive projection are used. In order to enforce unfolding every time we apply any other rule, we use the auxiliary relation $\Downarrow^\mu_\mathsf{p}$.

We now show that there is a correspondence between intermediate projection $\downarrow^\mu_\mathsf{p}$ and coinductive projection $\downarrow^\nu_\mathsf{p}$. In order to do so, we need to define how to construct a coinductive type from an inductive one. We have shown how to do this for inductive local types with $\mathsf{tocoind}(T^\mu)$ and we overload this tocoind function to similarly work with inductive global types G^μ. We use the abbreviations $\mathsf{Unravels}(G^\mu)$ and $\mathsf{Unravels}(T^\mu)$ for $G^\mu \ \mathcal{R} \ \mathsf{tocoind}(G^\mu)$ and $T^\mu \ \mathcal{R} \ \mathsf{tocoind}(T^\mu)$ respectively.

▶ **Lemma 15** (Unraveling of Projection)**.**
$G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ *iff* $\mathsf{Unravels}(G^\mu)$ *and* $\mathsf{Unravels}(T^\mu)$ *and* $\mathsf{tocoind}(G^\mu) \downarrow^\nu_\mathsf{p} \ \mathsf{tocoind}(T^\mu)$.

Proof. ($\Longrightarrow$) Derive both $\mathsf{Unravels}(G^\mu)$ and $\mathsf{Unravels}(T^\mu)$ by coinduction on $\mathcal{R}$ and inversion on $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$. Prove $\mathsf{tocoind}(G^\mu) \downarrow^\nu_\mathsf{p} \ \mathsf{tocoind}(T^\mu)$ by coinduction on $\downarrow^\nu_\mathsf{p}$ and derive from $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ that $\mathsf{guarded}^\mu_\mathsf{p}(G^\mu) \vee \mathsf{unfold}(T^\mu) = \mathsf{end}^\mu$ and proceed as in Lemma 13.
($\Longleftarrow$) Proof by coinduction on $\downarrow^\mu_\mathsf{p}$ and derive from $\mathsf{tocoind}(G^\mu) \downarrow^\nu_\mathsf{p} \ \mathsf{tocoind}(T^\mu)$ that $\mathsf{guarded}^\nu_\mathsf{p}(\mathsf{tocoind}(G^\mu)) \vee \mathsf{tocoind}(T^\mu) = \mathsf{end}^\nu$, case analysis on the disjunction as in Lemma 13, inverting $\mathsf{Unravels}(G^\mu)$ and $\mathsf{Unravels}(T^\mu)$ to derive the shape of $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$. ◀

▶ **Corollary 16.** $\mathsf{projectable}_\mathsf{p}(G^\mu)$ *iff* $G^\mu \downarrow^\mu_\mathsf{p} \ \mathsf{trans}_\mathsf{p}(G^\mu)$.

Proof. For ($\Longrightarrow$) we first show for any G^μ and G^ν, if $G^\mu \ \mathcal{R} \ G^\nu$ then $G^\nu = \mathsf{tocoind}(G^\mu)$ (and similarly for local types). Then both directions follow from Lemma 15. ◀

Deciding $G^\mu \downarrow^\mu_\mathsf{p} \ T^\mu$ is similar to deciding recursive type equivalence. Treatment of recursive types as graphs for equivalence testing is a well known approach [26] and solves the

problem by testing properties of reachable nodes in a directed graph. In this section, we do the same for deciding $G^\mu \downarrow^\mu_{\mathsf{p}} T^\mu$. First, we show how to transform global and local types into graphs. Then, we obtain a graph of the pair (G^μ, T^μ) by joining the graphs of G^μ and T^μ. Deciding $G^\mu \downarrow^\mu_{\mathsf{p}} T^\mu$ corresponds to testing a property on all reachable nodes of that graph.

Graphs. We first give the formal definition of graph, following that of Eikelder [26].

▶ **Definition 17** (Graph). *A directed graph is a triple (Q, d, δ) where:*

- *Q is a finite set of nodes*
- *d: $Q \to \mathbb{N}$ is a function returning the number of outgoing edges from a node*
- *$\delta : (Q \times \mathbb{N}) \rightharpoonup Q$ is the partial successor function such that $\delta(q, i)$ is the i^{th} successor of q, for $0 < i \leq d(q)$ nodes, and is undefined for all other i.*

Given a graph (Q, d, δ), we define the procedure sat_P which computes if all reachable nodes from an initial node q satisfy a given property P.

▶ **Definition 18** (sat_P). *The function $\mathit{sat}_P : 2^Q \to Q \to \{0, 1\}$, parameterised by a boolean predicate $P : Q \to \{0, 1\}$, is defined as:*

$$\mathit{sat}_P(V, q) = \begin{cases} 1 & \text{if } q \in V \\ P(q) \ \wedge \ \bigwedge_{i<d(q)} \mathit{sat}_P(\{q\} \cup V, \delta(q, i)) & \text{otherwise} \end{cases}$$

Given a set of visited nodes V, a current state q, and the predicate P, the function returns 1 if the node has already been visited; otherwise, it will recursively check the successors.

Global and Local Types as Graphs. We now give a procedure for constructing a graph from a global type. The graph construction for local types is similar and therefore omitted.

▶ **Definition 19** (Global type graph). *The graph of a global type G^μ is $(\mathit{enum}_g(G^\mu), d_g, \delta_g)$ where enum_g, d_g and δ_g are defined as:*

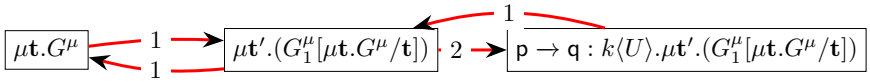
$$\mathit{enum}_g(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu) = \{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu\} \cup \mathit{enum}_g(G^\mu) \qquad \mathit{enum}_g(\mathsf{end}) = \{\mathsf{end}\}$$
$$\mathit{enum}_g(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}) = \{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J}\} \ \cup \ \bigcup_{j\in J} \mathit{enum}_g(G^\mu_j)$$
$$\mathit{enum}_g(\mathbf{t}) = \{\mathbf{t}\} \qquad \mathit{enum}_g(\mu\mathbf{t}.G^\mu) = \{\mu\mathbf{t}.G^\mu\} \cup \{G^\mu_1[\mu\mathbf{t}.G^\mu/\ \mathbf{t}] \mid G^\mu_1 \in \mathit{enum}_g(G^\mu)\}$$

$$d_g(G^\mu) = \begin{cases} 1 & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle u\rangle.G^\mu \\ |J| & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J} \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_g(G^\mu, i) = \begin{cases} G^\mu_1 & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle u\rangle.G^\mu_1 \ \wedge \ i = 1 \\ G^\mu_i & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J} \ \wedge \ 0 < i \leq |J| \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The enumeration function enum_g collects all subterms of a global type. In the case of $\mu\mathbf{t}.G^\mu$, it enumerates all subterms of the body G^μ that can contain free occurrences of $\mathbf{t}$, and substitute them all for $\mu\mathbf{t}.G^\mu$. These subterms are all nodes of the global type graph G^μ.

▶ **Example 20.** We show the global type graph of our main example from (7).



where $G^\mu := \mathsf{p} \to \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.G_1^\mu$ and $G_1^\mu := \mathsf{r} \to \mathsf{s} : k'\{l_1 : \mathbf{t},\ l_2 : \mathsf{p} \to \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$.

Given a global type G^μ, we wish to use $\mathsf{sat}_P(\{\}, G^\mu)$ to assert whether P holds for all nodes reachable by δ_g. To ensure termination of this procedure, we show that the set of reachable nodes is finite, a consequence of Q being closed under δ_g.

▶ **Lemma 21.** *If $G_1^\mu \in \mathit{enum}_g(G^\mu)$ and $0 \leq i < d_g(G_1^\mu)$, then $\delta_g(G_1^\mu, i) \in \mathit{enum}_g(G^\mu)$.*

Proof. Let δ_{aux} be δ_g without the use of unfold in the first two case distinctions, i.e., $\delta_g = \delta_{aux} \circ \mathsf{unfold}$. Showing $\mathsf{enum}_g(G^\mu)$ is closed under δ reduces to showing $\mathsf{enum}_g(G^\mu)$ is closed under δ_{aux} and unfold. By definition, $\mathsf{enum}_g(G^\mu)$ is closed under δ_{aux}. For $\mathsf{enum}_g(G^\mu)$ to be closed under unfold, it suffices to show that it is closed under unfold_once, which follows by induction on the μ-height. ◀

We are now ready to show how to use our graph construction for proving a property of a global type using sat_P. We do that by proving that $\mathsf{Unravels}(G^\mu)$ is decidable. In this case, we instantiate P in sat_P with a predicate that disallows global types to unfold to a top level μ-operator or a recursion variable.

▶ **Definition 22** (UnravelPred). *The predicate $\mathit{UnravelPred} : G^\mu \to \{0, 1\}$ is defined as:*

$$\mathit{UnravelPred}(G^\mu) = \begin{cases} 0 & \text{if } \mathit{unfold}(G^\mu) = \mu\mathbf{t}.G_1^\mu \vee \mathit{unfold}(G^\mu) = \mathbf{t} \\ 1 & \text{otherwise} \end{cases}$$

▶ **Lemma 23.** *$\mathit{Unravels}(G^\mu)$ iff $\mathit{sat}_{\mathit{UnravelPred}}(\{\}, G^\mu) = 1$*

The instantiation $\mathsf{sat}_{\mathsf{UnravelPred}}$ tests that G^μ and all successors of G^μ unfold to a message communication, a branching or end^μ. The procedure will for example fail for $\mu\mathbf{t}.\mathbf{t}$. More details on this procedure are given in Section 6.

We conclude this part by defining the partial functions LG, LT and PL_p. Given an inductive global type, function LG returns its *unfolded prefix*, i.e., information about its first occurring interaction.

▶ **Definition 24** (LG). *The function $LG \in G \rightharpoonup (\mathcal{P} \times \mathcal{P} \times \mathcal{C} \times (\{\bot\} \cup U))$ is defined as:*

$$LG(G^\mu) = \begin{cases} (\mathsf{p}_1, \mathsf{p}_2, k, U) & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \to \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \\ (\mathsf{p}_1, \mathsf{p}_2, k, \bot) & \text{if } \mathit{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Similar to how LG returns the unfolded prefix in a global type, we define the corresponding operation on local types as LT. We use the set $\{!, ?\}$ to indicate whether the communication is a send (!) or a receive (?).

▶ **Definition 25** (LT). *The function $LT : T \rightharpoonup (\{!, ?\} \times \mathcal{C} \times (\{\bot\} \cup U))$ is defined as:*

$$LT(T^\mu) = \begin{cases} (!, k, U) & \text{if } \mathit{unfold}(T^\mu) = !^\mu k\langle U\rangle.T_1^\mu \\ (?, k, U) & \text{if } \mathit{unfold}(T^\mu) = ?^\mu k\langle U\rangle.T_1^\mu \\ (!, k, \bot) & \text{if } \mathit{unfold}(T^\mu) = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \\ (?, k, \bot) & \text{if } \mathit{unfold}(T^\mu) = k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J} \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Finally, we can define a projection function on prefixes, i.e., a function that given a role and an unfolded prefix of a global type, returns an unfolded prefix of a local type.

▶ **Definition 26.** *The function $PL_\mathsf{p} \in (\mathcal{P} \times \mathcal{P} \times \mathcal{C} \times (\{\bot\} \cup U)) \rightharpoonup (\{!, ?\} \times \mathcal{C} \times (\{\bot\} \cup U))$ is defined as:*

$$PL_\mathsf{p}(\mathsf{p}_1, \mathsf{p}_2, k, U) = \begin{cases} (!, k, U), & \text{if } \mathsf{p}_1 = \mathsf{p} \\ (?, k, U), & \text{if } \mathsf{p}_2 = \mathsf{p} \text{ and } \mathsf{p} \neq \mathsf{p}_1 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Combining global and local type graphs. The next step towards deciding membership in $\lfloor \cdot \rfloor^{\mu}_{\mathsf{p}}$ is to combine the graphs of G^{μ} and T^{μ} into a single graph with respect to a role p.

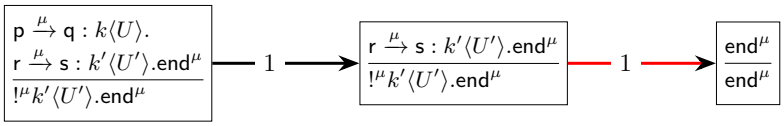
▶ **Definition 27** (Joint Global and Local Type Graph). *The graph of* (G^{μ}, T^{μ}) *with respect to* p *is the graph* $(\mathit{enum}(G^{\mu}, T^{\mu}), d, \delta_{\mathsf{p}})$, *where* enum, d *and* δ_{p} *defined as:*

$$\mathit{enum}(G^{\mu}, T^{\mu}) = \mathit{enum}(G^{\mu}) \times \mathit{enum}(T^{\mu}) \qquad d(G^{\mu}, T^{\mu}) = min(d_g(G^{\mu}), d_l(T^{\mu}))$$

$$\delta_{\mathsf{p}}((G^{\mu}, T^{\mu}), i) = \begin{cases} (\delta_g(G^{\mu}, i), \delta_l(T^{\mu}, i)) & \text{if } \mathsf{p} \in LG(G^{\mu}) \wedge 0 < i \leq d(G^{\mu}, T^{\mu}) \\ (\delta_g(G^{\mu}, i), T^{\mu}) & \text{if } \mathsf{p} \notin LG(G^{\mu}) \wedge 0 < i \leq d_g(G^{\mu}) \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The set of nodes is the Cartesian product of the nodes in the global type graph and the local type graph. The successor function δ_{p} takes a step with respect to a role p and the case distinction depends on this role. If p is in $LG(G^{\mu})$, the i^{th} successor of the graph is the i^{th} successor of the global and local type graph respectively. If p is not in the unfolded prefix, the global type moves to its successor while the local type stays fixed.

▶ **Example 28.** We show the joint graph of $\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U'\rangle.\mathsf{end}^{\mu}$ and $!^{\mu}k\langle U'\rangle.\mathsf{end}^{\mu}$ with respect to role c marking the edges black when the local type stays fixed.



Deciding membership in $\lfloor \cdot \rfloor^{\mu}_{\mathsf{p}}$. We define the predicate $\mathsf{ProjPred}_{\mathsf{p}}$ to decide membership in $\lfloor \cdot \rfloor^{\mu}_{\mathsf{p}}$. Intuitively, this predicate partitions the rules of $\lfloor \cdot \rfloor^{\mu}_{\mathsf{p}}$ into three sets such that the projected role p

1. is in the unfolded prefix ($[\mathsf{M1}\rfloor^{\mu}], [\mathsf{M2}\rfloor^{\mu}], [\mathsf{B1}\rfloor^{\mu}], [\mathsf{B2}\rfloor^{\mu}]$)
2. is not in the unfolded prefix, but the role is guarded in the global type ($[\mathsf{B}\rfloor^{\mu}$],$[\mathsf{M}\rfloor^{\mu}]$)
3. is not part of the global type ($[\mathsf{End}\rfloor^{\mu}]$).

We call rules in the first set *prefix rules* and rules in the second set *guarded rules*. The only rule that is not yet mentioned is unfolding, $[\mathsf{Unf}\rfloor^{\mu}]$, which is implicitly applied by the definition of δ.

▶ **Definition 29** ($\mathsf{ProjPred}_{\mathsf{p}}$). *The boolean predicate* $\mathit{ProjPred}_{\mathsf{p}} \in G^{\mu} \times T^{\mu} \to \{0, 1\}$ *is defined as:*

$$\mathit{ProjPred}_{\mathsf{p}}(G^{\mu}, T^{\mu}) = \begin{cases} (\mathbf{1})\ PL_{\mathsf{p}}(LG(G^{\mu})) = LT(T^{\mu}) \wedge & \\ \quad d_g(G^{\mu}) = d_l(T^{\mu}) & \text{if } PL_{\mathsf{p}}(LG(G^{\mu})) \text{ is defined} \\ (\mathbf{2})\ 0 < d_g(G^{\mu}) & \text{if } \mathsf{partOf}^{\mu}_{\mathsf{p}}(G^{\mu}) \text{ and } \mathsf{guarded}^{\mu}_{\mathsf{p}}(G^{\mu}) \\ (\mathbf{3})\ \mathit{sat}_{\mathit{UnravelPred}}(\{\}, G^{\mu}) \wedge & \\ \quad \neg\mathsf{partOf}^{\mu}_{\mathsf{p}}(G^{\mu}) \wedge & \\ \quad \mathit{unfold}(T^{\mu}) = \mathit{end} & \text{otherwise} \end{cases}$$

We explain the three cases of the predicate.

1. Attempt to apply a prefix rule: This requires p to be in the unfolded prefix of the global type. This is checked by requiring that PL_{p} is defined. We then apply PL_{p} to the unfolded prefix, and assert it equal to the unfolded prefix of the local type. All prefix rules require the global and local type to have equally many outgoing edges, which we check by $d_g(G^{\mu}) = d_l(T^{\mu})$.

2. Attempt to apply a guarded rule: We rely on decidability of partOf^{μ} and $\mathsf{guarded}^{\mu}$ which is straightforward so we do not detail how[2]. All guarded-rules require the set of outgoing edges of the global type to be greater than zero, which we assert. Concretely this test corresponds to the first premise of rule $[\mathsf{B}\rfloor^{\mu}]$, asserting its label set is non-empty.
3. Attempt to apply $[\mathsf{End}\rfloor^{\mu}]$.

▶ **Theorem 30.** $G^{\mu} \upharpoonright^{\mu}_{\mathsf{p}} T^{\mu}$ *iff* $\mathit{sat}_{\mathit{ProjPred}_{\mathsf{p}}}(\{\}, (G^{\mu}, T^{\mu})) = 1$

Proof. For $(\Longrightarrow)$, we show the property for any visited list v, that is, $G^{\mu} \upharpoonright^{\mu}_{\mathsf{p}} T^{\mu}$ implies $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu})) = 1$. Proceed by functional induction on $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu}))$. For $(\Longleftarrow)$, for any v, it suffices to show $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu})) = 1$ implies $(G^{\mu}, T^{\mu}) \in v \vee G^{\mu} \upharpoonright^{\mu}_{\mathsf{p}} T^{\mu}$. Proceed by functional induction on $\mathsf{sat}_{\mathsf{ProjPred}_{\mathsf{p}}}(v, (G^{\mu}, T^{\mu}))$. In the second case where v is non-empty, pick the right disjunct $G^{\mu} \upharpoonright^{\mu}_{\mathsf{p}} T^{\mu}$ and proceed by coinduction. ◀

▶ **Corollary 31.** $\mathsf{projectable}_{\mathsf{p}}(G^{\mu})$ *is decidable.*

Proof. Follows from Theorem 30 and Corollary 16. ◀

6 Mechanisation

All of our results are mechanised in Coq [6] using SSReflect [14] for writing proofs, the Paco library [18] for defining coinductive predicates, the Equations package [24] for defining functions by well-founded recursion (such as sat_P), and Autosubst2 [25] to generate syntax of inductive global and local types with binders represented by De Bruijn indices [10].

The mechanisation uses coinductive extensional equivalence relations to equate coinductive terms. For presentation purposes, e.g. in the conclusion of Lemma 12, we use propositional equality to equate coinductive types. These two types of equality are consistent [1].

In this section, we cover how to create predicates and relations that are defined using both inductive and coinductive inference rules, like our unravelling relation from Definition 3. We discuss how to create an inversion principle that allows us to do case analysis on predicates of the form $\mathsf{Unravels}(G^{\mu})$ which, as discussed in Section 5, is defined as $G^{\mu}\ \mathcal{R}\ \mathsf{tocoind}(G^{\mu})$. Finally, we show how we prove decidability of $\mathsf{Unravels}$ using sat_P.

Mixed inductive and coinductive definitions. The unravelling relation presented in Definition 3 uses a combination of inductive and coinductive rules, which is non-standard. We do this because it greatly simplifies our proofs and disallows unwanted derivations like the one presented in Section 3 (8) by construction. We mix inductive and coinductive rules by taking the greatest fixed point of a generating function defined as a least fixed point, a technique that Zakowski et al. [29] also have used to define weak bisimilarity of streams.

```
Definition grel := gType -> gcType -> Prop
Inductive UnravelF (R : grel) : grel := (* Generating function UnravelF *)
 | UnrF1 g gc a u : R g gc -> UnravelF R (GMsg a u g) (GCMsg a u gc)
 (* The branching rule is elided *)
 | UnrF_unf1 g gc : UnravelF R (unf1 (GRec g)) gc -> UnravelF R (GRec g) gc
 | UnrF_end       : UnravelF R GEnd GCEnd.
Definition Unravelling : grel := paco2 UnravelF bot2 (* gfp UnravelF *)
```

[2] We need to assert both $\mathsf{partOf}^{\mu}_{\mathsf{p}}(G^{\mu})$ and $\mathsf{guarded}^{\mu}_{\mathsf{p}}(G^{\mu})$ for completeness as we from $G^{\nu} \upharpoonright^{\nu}_{\mathsf{p}} \mathsf{end}^{\nu}$ and $G^{\mu}\ \mathcal{R}\ G^{\nu}$ then can conclude the third case of $\mathsf{ProjPred}_{\mathsf{p}}$.

We represent $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu$ and $\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu$ as GMsg a u g of type gType and GCMsg a u gc of type gcType respectively, where a contains roles p_1, p_2 and channel k, u is U, g is G^μ, and gc is G^ν. The terms GEnd and GCEnd represent end^μ and end^ν respectively and the function unf1 is the unfold_once function from Section 4.

UnravelF is an inductively defined relation relating global inductive types to global coinductive types. It is parameterised by a relation R of the same type where (g, gc) $\in$ UnravelF R if, after unfolding a finite number of binders from g resulting in type g', either g' = GEnd and gc = GCEnd, or g' = GMsg a u g'', gc = GCMsg a u gc'', and (g'', gc'') $\in$ R, or similarly for the elided branch case.

Intuitively, UnravelF is a generating function defined as a least fixed point and by taking the greatest fixed point of this function we obtain a hybrid inductive/coinductive relation where any occurrence of R in a premise of UnfoldF require us to take coinductive steps in our proofs and any recursive occurrence of UnfoldF requires us to take inductive steps. This allows us to do proofs like (9) where proofs are finished by circling back to previous equivalent nodes in the tree in the coinductive cases or by reaching a base case in the inductive cases. Moreover this approach forbids us from unfolding binders indefinitely since UnrF_unf1 is inductive and not coinductive.

We use paco2 from the Paco library to define Unravelling as the greatest fixed point of UnravelF. Paco stands for parameterised coinduction and paco2 F R defines the greatest fixed point of F parameterised by a binary relation R, which is equivalent to $\mathsf{gfp}(\lambda X.\ \mathsf{F}(X \cup \mathsf{R}))$. When R is the empty set this coincides with the standard greatest fixed point.

Custom inversion principles. Many proofs on inductive global types work up to unfolding. Unravelling, for instance, unravels a finite number of μ-binders at every step and our intermediate projection function $\lfloor \cdot \rfloor^\mu_{\mathsf{p}}$ and sat procedure both work in a similar way. To abstract away from finite unfoldings we use the following InvPred predicate.

```
Variant InvPredF (P : gType -> Prop) : gType -> Prop :=
| HTM g a u : P g          -> InvPredF P (GMsg a u g)
| HTB gs d  : Forall P es -> InvPredF P (GBranch d gs)
| HTE       :                 InvPredF P GEnd
Definition unf g := (iter (mu_height g) unf1 g).
Variant UnfoldF (P : gType -> Prop) : gType -> Prop :=
 | UnfF1 g :  P (unf g) -> UnfoldF g.
Definition InvPred : (gType -> Prop) := paco1 (UnfoldF \o InvPredF) bot.
                                                (*function composition*)
```

We define two generating functions InvPredF and UnfoldF and generate InvPred as the greatest fixed point of their composition. The function unf corresponds to unfold from Section 4. InvPredF contains cases for all constructors of inductive global types except for $\mu\mathbf{t}$ and $\mathbf{t}$. UnfoldF unfolds the top-level μ-binders from a global type. The key insight is that InvPred(G^μ) is equivalent to asserting closedness and contractiveness of G^μ.

The inversion principle of InvPred is convenient for proving predicates P that are closed under unfolding of inductive global types, i.e. $\forall G.\ P\ \mu\mathbf{t}.G \iff P\ G[\mu\mathbf{t}.G]$, as any unfolding applied by inverting UnfoldF can similarly be applied in the goal. In particular the predicate Unravels(G^μ) is closed under unfolding and provable by inversion of UnfoldF.

Well-foundedness of sat$_P$. Lemma 23 proves decidability of Unravels. This proof is mechanised by proving decidability of InvPred(G^μ), which as we show above implies Unravels(G^μ). The invP predicate corresponds to Definition 22.

```
Definition invP g :=
match unf g with | GRec _ | GVar _ => false | _ => true end.

Definition invpred g := sat nil invP g.
Theorem InvPred_dec : forall g, InvPred g <-> invpred g = true
```

We use the Equations package to define sat_P by well-founded recursion on the decreasing measure gmeasure g V which is defined as the number of unique nodes in the graph created from g minus the cardinality of the visited set V. The successor function δ_g is implemented by nextg : gType -> seq gType.

```
1  Definition gmeasure (g : gType) (V : seq gType) :=
2    size (rep_rem V (undup (enumg g))).
3  Lemma closed_enum  : forall g0 g1 g2, g1 \in nextg (unf g) ->
4                                        g2 \in enumg g1  -> g2 \in enumg g.
5  Equations sat (V : seq  gType)  (P : gType -> bool)
6                    (g : gType) : bool by wf (gmeasure g V) :=
7   sat V P g with (dec (g \in V)) => {
8    sat  _ _ _ in_left  := true;
9    sat V P g  in_right :=  (P g) &&
10                           (foldInMap (nextg (unf g))
11                                      (fun g' _ => sat (g::V) P g')) }.
```

Defining sat generates one obligation that must be proved to show termination. If we write gmeasure g V as $M(g, V)$, then we must show it is decreasing for arguments to the recursive call, i.e. that $M(g', \{g\} \cup V) < M(g, V)$

Using a variant of the familiar map on inductive lists called foldInMap our obligation is enriched with the assumption that $g' = \delta(g, i)$ for some $0 < i \leq d_g(g)$. The boolean wrapper dec further enriches the obligation with the case of the if-statement, $g \notin V$.

What must be proven in this obligation is slightly different from the termination argument in Section 5 which relied on the finiteness of a graph's nodes. The obligation instead relies on a lemma closed_enum (l. 3). The lemma states that the enumerations of a global types continuations, will all be part of the initial global types enumeration. The proof of this lemma is short, less than 100 lines.

The full termination proof for sat is short (about 250 lines) and the approach is general. The mechanisation also proves termination of the decision procedure for membership in $\lfloor^{ind}$. This task only requires adapting the algorithm to pairs of terms. This termination proof is also short. The conciseness is due to the space of continuations being computed by structural recursion by enum. This makes it straightforward to prove substitution properties about it by induction on syntax.

7 Related Work and Discussion

Related Work. Ghilezan et al. [13] are the first to introduce coinductive projection on coinductive global and local types. They use it to show soundness and completeness of synchronous multiparty session subtyping. A key difference is that whereas we represent the infinite unfolding of a μ-type as a coinductive type, they represent it as a partial function. Projection on μ-types is then defined indirectly in terms of the coinductive projection of their corresponding partial functions. Because of this indirect definition, their projection is not computable. Our intermediate projection $\lfloor^{\mu}$ is similar to their projection on μ-types.

However, ours is defined with inference rules stated directly on the μ-types which is why we can decide membership and thus compute projection. Castro-Perez et al. [7] use coinductive projection to express their meta theory about multiparty session types. Their main result is trace equivalence between processes, coinductive local types and coinductive global types, which they mechanise in Coq. Like us, they show soundness of their projection on μ-types. Their projection is however not complete, which is what inspired us to investigate approaches to sound and complete projection. A consequence of their projection on μ-types not being complete, is that there are many inductive global types that have the trace equivalence property, but must be excluded since their projection is undefined. Jacob et al. [19] show deadlock and leak freedom of multiparty GV, an extension of the functional language GV [12, 28]. They use coinductive projection to define when local types are compatible and do not define a projection on μ-types. Other work has formalised the notion of projection in Coq. Cruz-Filipe et al. [9, 8] formalise syntax and semantics of tail-recursive choreographies and a projection that includes full merge. However, this work does not approach coinductive syntax and therefore does not show any soundness and completeness results.

Our graph algorithm from Section 5 implements a procedure proposed by Eikelder [26]. This work provides several algorithms for deciding recursive type equivalence that, like ours, use predicates on reachable nodes of a graph. Also, our proof of termination is quite similar to theirs. However, they define the set of reachable states as set comprehension, whereas we constructively produce a list of nodes. Similarly, showing their set comprehension is finite, boils down to substitution lemmas. Unlike ours, their work has not been mechanised in a proof assistant/theorem prover. The idea of defining the space of continuations for global and local type as an explicit enumeration is inspired by Asperti [3] who mechanise a concise proof of regular expression equivalence in the Matita theorem prover [4]. They do this by a new construction called pointed regular expressions. Essentially, this adds marks to a regular expression, such that one can encode state transitions by moving marks. This makes computing reachable configurations as trivial as computing all markings.

We define unravelling using a mix of inductive and coinductive rules. In Section 6, we make this precise by defining unravelling as the greatest fixed point of a generating function itself defined as a least fixed point. Zakowski et al. [29] use the same technique to define a weak bisimilarity on streams.

The primary focus of this work is on global types. Scalas and Yoshida [23] propose a more general approach that shows that properties such as deadlock freedom can be derived directly on local types without the need for global types and the corresponding projection. However, their approach misses the main advantage provided by global types which is providing a specification (blueprint) of the used protocols.

Discussion and Future work. This work is part of the MECHANIST project that aims at mechanising the full theory of multiparty asynchronous session types [17]. Our next step is to mechanise a proof of semantic equivalence between global types and their projections to local types through $\mathsf{proj_p}$. Semantic equivalence is a property similar to trace equivalence which Castro-Perez et al. [7] mechanised. However, there are some key differences in our objectives. Their main result is Zooid, a tool that extracts certified message-passing programs, which is why their process syntax differs significantly from the original syntax by Honda et al (e.g., no parallel composition). Instead, we aim at mechanising the exact process calculus presented by Honda et al.. As the meta theory in Castro-Perez et al. [7] is independent of their projection function, it would also be interesting future work to adapt $\mathsf{proj_p}$ to their setting. Finally, $\mathsf{proj_p}$ implements the restrictive plain merge but related work also uses full merge [13, 8]. It would be interesting to define a binder-agnostic projection using full merge.

8 Conclusions

Projection is a function that maps global types to local types. The projections found in the literature impose syntactic restrictions that make them incomplete with respect to coinductive projection. This work shows the existence of a decidable projection that is sound and complete. Our procedure works in two phases: first a decision procedure tests a soundness property and, if successful, a second procedure translates the global type to a local type. The latter is very similar to the existing projections in the literature. The novelty of our work is in the decision procedure. All results have been mechanised in Coq.

References

1 Coinductive types and corecursive functions. https://coq.inria.fr/refman/language/core/coinductive.html. Accessed: May 2023.

2 Session types in programming languages: A collection of implementations. http://www.simonjf.com/2016/05/28/session-type-implementations.html. Accessed: May 2023.

3 Andrea Asperti. A compact proof of decidability for regular expression equivalence. In *proceedings of ITP*, volume 7406 of *LNCS*, pages 283–298. Springer, 2012. doi:10.1007/978-3-642-32347-8_19.

4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *Proceedings of CADE*, volume 6803 of *LNCS*, pages 64–69. Springer, 2011. doi:10.1007/978-3-642-22438-6_7.

5 Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. In *Proceedings of PLACES*, volume 241 of *ENTCS*, pages 3–33. Elsevier, 2008. doi:10.1016/j.entcs.2009.06.002.

6 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07964-5.

7 David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In *Proceedings of PLDI*, pages 237–251. ACM, 2021. doi:10.1145/3453483.3454041.

8 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Proceedings of ICTAC 2021*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.

9 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a turing-complete choreographic language in coq. In Liron Cohen and Cezary Kaliszyk, editors, *Proceedings of ITP 2021*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.15.

10 Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.

11 Romain Demangeon and Nobuko Yoshida. On the expressiveness of multiparty sessions. In *Proceedings of FSTTCS*, volume 45 of *LIPIcs*, pages 560–574, 2015. doi:10.4230/LIPIcs.FSTTCS.2015.560.

12 Simon Gay and Vasco Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

13 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *Journal of Logical and Albegraic Methods in Programming*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.

14 Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the coq system, 2016. URL: https://inria.hal.science/inria-00258384.

15 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

16 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of POPL*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.

17 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

18 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of POPL*, pages 193–206. ACM, 2013. doi:10.1145/2429069.2429093.

19 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proceedings of the ACM on Programming Languages*, 6(ICFP):466–495, 2022. doi:10.1145/3547638.

20 Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150:347–377, 2017. doi:10.3233/FI-2017-1473.

21 The Coq development team. The Coq Proof Assistant. https://coq.inria.fr. Accessed: May 2023.

22 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

23 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

24 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP):86:1–86:29, 2019. doi:10.1145/3341690.

25 Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In *Proceedings of CPP*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

26 Huub ten Eikelder. Some algorithms to decide the equivalence of recursive types. https://pure.tue.nl/ws/files/2150345/9211264.pdf, 1991. Accessed: May 2023.

27 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make session types complete for lock-freedom. In *Proceedings of LICS*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470531.

28 Philip Wadler. Propositions as sessions. In *Proceedings of ICFP*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.

29 Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of CPP*, pages 71–84. ACM, 2020. doi:10.1145/3372885.3373813.

Appendix B

A Sound and Complete Projection for Global Types (journal version)

A Sound and Complete Projection for Global Types

Dawit Tirore, Jesper Bengtson and Marco Carbone

*Computer Science Department, IT University of Copenhagen, Rued Langgards Vej 7, 2300 Copenhagen, Denmark.

*Corresponding author(s). E-mail(s): dati@itu.dk; bengtson@itu.dk; carbonem@itu.dk;

Abstract

Multiparty session types is a typing discipline used to write specifications, known as global types, for branching and recursive message-passing systems. A necessary operation on global types is projection to abstractions of local behaviour, called local types. Typically, this is a computable partial function that given a global type and a role erases all details irrelevant to this role.
Computable projection functions in the literature are either unsound or too restrictive when dealing with recursion and branching. Recent work has taken a more general approach to projection defining it as a coinductive, but not computable, relation. Our work defines a new computable projection function that is sound and complete with respect to its coinductive counterpart and, hence, equally expressive. All results have been mechanised in the Coq proof assistant.

Keywords: Session types, Mechanisation, Projection, Coq

1 Introduction

Session types are types for abstracting the behaviour of communicating processes. First proposed by Honda et al. [1] for binary protocols, they specify the sequence of possible actions processes need to follow when sending and receiving messages over a channel. Session types provide a clear language for describing protocols that are guaranteed to not deadlock or contain communication errors, e.g., never receive an integer when expecting a boolean. A decade after their conception, Honda et al. [2]

proposed a generalisation, called *multiparty session types*, that specifies how an arbitrary but fixed number of processes should interact with each other. Multiparty session types are based on the concept of *global types* which provide a global description of the multiparty protocol being abstracted. Recently, multiparty session types have gained interest from several communities, resulting in their integration into several mainstream programming languages [3].

Multiparty session types follow a precise approach to designing and implementing communicating processes: from global types that specify the protocols, we can automatically generate *local types*, the local specifications of the behaviour of each *role* in the protocol; then, each local type specification is (type) checked against the local code being written by the programmer. The algorithmic generation of local types from global types, called *projection*, is key for relating global types to implementations. Projection is a partial function which, when defined, outputs a local type that captures the specification of the projected role given by the global type. As an example, let us consider a global type where a role r asks another role s to either go `Left` or `Right`, over some channel k:

$$\mathsf{r} \to \mathsf{s} : k \left\{ \begin{array}{l} \texttt{Left} : \ \mathsf{r} \to \mathsf{s} : k'\langle \texttt{Int} \rangle.\mathsf{p} \to \mathsf{q} : k''\langle \texttt{Int} \rangle.\texttt{end} \\ \texttt{Right} : \ \mathsf{r} \to \mathsf{s} : k'\langle \texttt{String} \rangle.\mathsf{p} \to \mathsf{q} : k''\langle \texttt{Int} \rangle.\texttt{end} \end{array} \right\} \tag{1}$$

Above, if r chooses `Left`, it will also send an integer (`Int`) over some other channel k'; otherwise, it will send a string (`String`). No matter what branch r chooses, all roles must collectively follow the description of that branch.

Nested in both branches, there is a communication over k'' of an integer `Int` between p and q. The projections of r and p are:

$$\mathsf{r} : \quad k \oplus \left\{ \begin{array}{l} \texttt{Left} : !k'\langle \texttt{Int} \rangle.\texttt{end} \\ \texttt{Right} : !k'\langle \texttt{String} \rangle.\texttt{end} \end{array} \right\} \qquad\qquad \mathsf{p} : \quad !k''\langle \texttt{Int} \rangle.\ \texttt{end}$$

Above, r makes a choice (denoted by $\oplus$), and depending on this choice, outputs either a message of type `Int` or `String` over channel k, while p sends an `Int` over channel k''. An important observation is that, since neither p nor q are informed of the choice made by r, their behaviour should be independent from the choice of r. In fact, a *restriction* that projection usually imposes is that all roles that do not participate in a branching interaction behave the same on all branches. Global types that satisfy this condition are called projectable global types. The global type in Equation (1) is projectable because the projection of p on the `Left` branch produces the same local type as the projection on the `Right` branch. The same holds for q.

In order to express repetitive behaviour, global types (and local types) are usually equipped with recursion, expressed as μ*-types* [4]. Consider for example the global type below, which similarly to the global type in Equation (1) has an initial branching interaction between r and s:

$$\mathsf{r} \to \mathsf{s} : k \left\{ \begin{array}{l} \texttt{Left} : \mu \mathsf{t}.\mathsf{p} \to \mathsf{q} : k''\langle \texttt{Int} \rangle.\mathsf{t} \\ \texttt{Right} : \mu \mathsf{t}.\mathsf{p} \to \mathsf{q} : k''\langle \texttt{Int} \rangle.\mathsf{t} \end{array} \right\} \tag{2}$$

In both branches of the global type above, a μ-binder is used to specify repetition of the interaction $\mathtt{p} \to \mathtt{q} : k''\langle \mathtt{Int} \rangle$. In this case, projectability of this global type is easy to check because both branches are exactly the same.

The recursive behavior of μ-types is due to their intrinsic operation, known as unfolding. Consider the global type obtained by unfolding the binder in the `Left` branch, and leaving the `Right` branch unchanged:

$$\mathtt{r} \to \mathtt{s} : k \left\{ \begin{array}{l} \mathtt{Left} : \mathtt{p} \to \mathtt{q} : k''\langle \mathtt{Int} \rangle.\ \mu \mathbf{t}.\mathtt{p} \to \mathtt{q} : k''\langle \mathtt{Int} \rangle.\mathbf{t} \\ \mathtt{Right} : \mu \mathbf{t}.\mathtt{p} \to \mathtt{q} : k''\langle \mathtt{Int} \rangle.\mathbf{t} \end{array} \right\} \tag{3}$$

To the best of our knowledge, no computable definition of projection can determine the projectability of the global type in Equation (3). Recognising that this global type is projectable for $\mathtt{p}$ requires comparing the projections of the branches in a more permissive way than strict equality, relating the two local types:

$$!k''\langle \mathtt{Int} \rangle.\ \mu \mathbf{t}.!k''\langle \mathtt{Int} \rangle.\ \mathbf{t} \qquad \mu \mathbf{t}.!k''\langle \mathtt{Int} \rangle.\ \mathbf{t} \tag{4}$$

The standard way to relate these local types is by a coinductive equality, which uses the unfolding operation [5], and there exists multiple ways of of defining decision procedures for this relation [4, 6]. Using this more permissive way of relating the projections of branches, makes the global type in Equation (3) projectable. It is however insufficient to recognise the projectability of the following global type:

$$\mu \mathbf{t}.\ \mathtt{p} \to \mathtt{q} : k\langle \mathtt{String} \rangle.\ \mu \mathbf{t}'.\ \mathtt{r} \to \mathtt{s} : k' \left\{ \begin{array}{l} \mathtt{Left} :\ \mathbf{t} \\ \mathtt{Right} :\ \mathtt{p} \to \mathtt{q} : k\langle \mathtt{String} \rangle.\mathbf{t}' \end{array} \right\} \tag{5}$$

Intuitively, the behaviour of $\mathtt{p}$ is the same on both branches. That is, $\mathtt{p}$ always sends a message of type `String` to $\mathtt{q}$ over channel k, which we can represent by the local type $\mu \mathbf{t}.\ !k\langle \mathtt{String} \rangle.\ \mathbf{t}$. That is, the same specification can be given using a single binder in the following way:

$$\mu \mathbf{t}.\ \mathtt{p} \to \mathtt{q} : k\langle \mathtt{String} \rangle.\ \mathtt{r} \to \mathtt{s} : k' \{\mathtt{Left} :\ \mathbf{t}, \quad \mathtt{Right} :\ \mathbf{t}\} \tag{6}$$

While the projection of (6) onto $\mathtt{p}$ is defined for most, if not all, projections in the literature, the projection of $\mathtt{p}$ onto (5) is undefined for all projection algorithms in the literature. The projection of (5) remains undefined. This makes the algorithmic projection functions in the literature incomplete, since there are some global types that should be projectable, but are not. This is even the case if coinductive equality, rather than syntactic equality, is used to relate the projection of branches, since the projected branches in (5) are not related by coinductive equality:

$$\mathbf{t} \not\approx\ !^{\mu}k\langle \mathsf{String} \rangle.\mathbf{t}' \tag{7}$$

The projected branches are however equivalent, when we consider the context they appear in. In the projection of (5) on $\mathtt{p}$, the following local type context has been

generated so far, when we reach the branches:

$$\mu \mathbf{t}.!^{\mu} k\langle \mathsf{String} \rangle.\mu \mathbf{t}'.[\cdot]$$

Here $[\cdot]$ denotes a hole that will be instantiated by the projection of a branch. Applying this context to the local types in (7) yields local types that are related by coinductive equality:

$$\mu \mathbf{t}.!^{\mu} k\langle \mathsf{String} \rangle.\mu \mathbf{t}'.\mathbf{t} \approx \mu \mathbf{t}.!^{\mu} k\langle \mathsf{String} \rangle.\mu \mathbf{t}'.!^{\mu} k\langle \mathsf{String} \rangle.\mathbf{t}' \tag{8}$$

We are thus faced with a discrepancy: the global types in Equations (5) and (6) are semantically equivalent, yet the projection onto p is defined only for the latter. This makes projection incomplete and is a consequence of how projection is usually implemented. This discrepancy cannot be rectified by integrating decision procedures for coinductive equality into existing definitions of projection. To address this issue, projection must be defined in a way that combines coinductive equality with contextual information, as was done in (8).

The most common way of defining projection is as a structurally recursive, partial function on global types, which we call *standard projection*. More recently, Ghilezan et al. [7] define projection in a declarative manner as a coinductive relation on coinductive global and local types. Unlike most definitions in the literature, theirs is not an algorithmic procedure that can be executed. They use this relation to prove properties about a relation on local types called synchronous subtyping (cf. Definition 3.15 [7]). Our focus is the projection operation, which can be used in both synchronous and asynchronous settings, and our work thus covers both synchronous and asynchronous multiparty session types. Following the approach of Ghilezan et al., we define a coinductive global type that represents both the inductive global types in Equations (5) and (6), and this coinductive global type can project onto p. Projection on coinductive types, called coinductive projection, is defined as a relation, making the definition declarative, and not algorithmic. Defining projection as an actual algorithm that can be executed is important for multiparty session types, as it makes decidable type checking possible (cf. Proposition 4.6 [8]). Using an example similar to (3), Ghilezan et al. are the first to point out the expressiveness of coinductive projection in comparison to the standard projection. They show that the way standard projection treats binders, causes the procedure to be undefined for some global types, whose alternative representation as coinductive global types do have a coinductive projection. An example of this is our running example in Equation (5).

Contributions and Structure. In this paper, we define an algorithmic version of the declarative definition of projection by Ghilezan et al. That is, we define a computable projection on inductive global types that can handle global types such as that in Equation (5), making it more general than existing computable projection definitions in the literature. As an algorithmic version of coinductive projection, our procedure is not only more general, but also complete with respect to the coinductive projection. All our proofs have been mechanised[1] in the Coq proof assistant [9].

[1] The code can be found at: https://github.com/Tirore96/projection

This is an extended version of Tirore et al. [10], which includes the following additions:

- We extend our presentation of inductive types. This includes definitions which were previsouly elided and Lemma 12 which states a list of properties about unfolding μ-types. We also extend our presentation of coinductive types and the unravelling relation that relates inductive and coinductive types.
- We have simplified the presentation of our projection procedure, seen in Definition 5. Section 4.1 now contains a more detailed account of the termination proof for the decision procedure which projection relies on.
- We give a more detailed account of the soundness and completeness proofs, and we simplify their presentation by containing aspects of soundness in Section 5 and completeness in Section 6.

We structure the paper as follows. Section 2 defines inductive global and local types, and provides an overview of existing variants of standard projection along with their pitfalls. Section 3 introduces coinductive global types, coinductive local types, and the coinductive projection that relates them. Section 4 presents our new projection function on inductive global types and proves termination for this procedure. Section 5 proves soundness of the procedure, and Section 6 proves completeness. Section 7 details key insights from our Coq mechanisation. Finally, Section 8 covers related and future work, and Section 9 concludes.

Throughout the paper, we use the symbol 🐓 to link definitions, lemmas, theorems, and other elements to the corresponding code in our repository.

2 Inductive Types and Standard Projection

We recapitulate the standard inductive syntax of global and local types and introduce useful predicates on these types. To properly position our work we cover existing inductive definitions of projection found in the literature.

2.1 Syntax

Let $\mathcal{P}$ be a set of roles, ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{t}$, $\mathcal{L}$ a set of labels, ranged over by l, and $\mathcal{X}$ a set of recursion variables ranged over by $\mathbf{t}$.

Definition 1 (Inductive Global and Local Types [8])**.** 🐓 *Global types G^μ and local types T^μ are μ-types generated inductively by the following grammars, where U represents primitive types:*

$$G^\mu \ ::= \ \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \ \mid \ \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \ \mid \ \mu\mathbf{t}.G^\mu \ \mid \ \mathbf{t} \ \mid \ \mathsf{end}^\mu$$

$$T^\mu \ ::= \ !^\mu k\langle U\rangle.T^\mu \ \mid \ ?^\mu k\langle U\rangle.T^\mu \ \mid \ k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \ \mid \ k \,\&^\mu \{l_j : T_j^\mu\}_{j\in J} \ \mid \ \mu\mathbf{t}.T^\mu \ \mid \ \mathbf{t} \ \mid \ \mathsf{end}^\mu$$

The type $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu$ denotes a communication between roles p_1 and p_2 via channel k of a message of type U, which then proceeds as G^μ. Similarly, the type $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J}$ denotes a communication between two roles where, given the

set of indices J, role p_1 selects a branch with label l_i, and then proceeds as G_i^μ. Types $\mu\mathbf{t}.G^\mu$ and $\mathbf{t}$ model recursive protocols. Finally, end^μ denotes the successful termination of a protocol. A message type U is an unspecified value type such as **int** or **bool**: their formal treatment, necessary for the typing system [2], is not relevant for the focus of this paper.

The local type $!^\mu k\langle U\rangle.T^\mu$ outputs a message of type U over channel k, while its dual, $?^\mu k\langle U\rangle.T^\mu$, receives a message of type U over k. Types $k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J}$ and $k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J}$ implement branching where the former is the type of a process that internally selects a branch l_i and communicates it over channel k, while the latter is the type of a process that offers choices $l_1, \ldots, l_n$ (for $J = \{1, \ldots, n\}$ with $n \geq 1$) over channel k. We overload end^μ and $\mu\mathbf{t}.$ using it also for local types. We deal with recursive variables in a standard way and write capture-avoiding substitution as $G_1^\mu[G_2^\mu/\mathbf{t}]$.

2.2 Predicates on Global and Local Types

It is standard practice to work only with closed and contractive μ-types [2, 4]. Closedness prohibits free recursion variables in types and the type $\mathsf{p} \to \mathsf{q} : k\langle U\rangle.\mathbf{t}$, for instance, is not closed. A μ-type G^μ (or T^μ) is contractive if, for any of its subterms with shape $\mu\mathbf{t}_0.\mu\mathbf{t}_1...\mu\mathbf{t}_n.\mathbf{t}$, the body $\mathbf{t}$ is not $\mathbf{t}_0$ [4]. In particular, the type $\mu\mathbf{t}.\mathbf{t}$ is not contractive.

Closedness and contractiveness, collectively called *well-formedness* of μ-types, are structural properties because they are defined by structural recursion on the type syntax. The formal definition for closedness (closed) and contractiveness (contr) are presented in Figures 1a and 1b respectively. It is common practice to assume that μ-types are always well-formed and we deviate from this practice by not making this assumption. We do this both to stay closer to the formalisation, explicitly stating in all lemmas and theorems what is assumed, and because Section 3 introduces a novel way of capturing well-formedness coinductively.

A type is closed if it has no free recursion variables. The predicates $\mathsf{closed}(G^\mu)$ and $\mathsf{closed}(T^\mu)$, defined in Figure 1a, are true if G^μ, and T^μ respectively, contain no free recursion variables.

The definition of contractiveness depends on auxiliary predicates $\mathsf{guardedVar}(\mathbf{t}, G^\mu)$ and $\mathsf{guardedVar}(\mathbf{t}, T^\mu)$, defined in Figure 1b, which check that G^μ, and T^μ respectively, are not a sequence of μ-binders followed by $\mathbf{t}$. As an example, $\mathsf{guardedVar}(\mathbf{t}, \mu\mathbf{t_0}.\mu\mathbf{t_1}.\mathbf{t})$ returns False. Note that we do not use the Barendregt convention [11] in the definition of contractiveness[2].

One key feature of contractive μ-types is that unfolding top level μ-operators a finite number of times, by replacing $\mu\mathbf{t}.G^\mu$ with $G^\mu[\mu\mathbf{t}.G^\mu/\mathbf{t}]$, will result in a type that has a communicating action at the top. More precisely, the finite number of unfoldings required is the number of top-level μ-operators of a contractive type, which we call the μ-height and denote by $|\cdot|$. For example, $|\mu\mathbf{t}.\mathsf{end}| = 1$ and $|\mathsf{p} \xrightarrow{\mu} \mathsf{p}' : k\langle U\rangle.\mu\mathbf{t}.\ \mathsf{end}| = 0$.

[2]This is a consequence of the Coq mechanisation representing binders with de Bruijn indices, while the paper presentation using a named representation. To present the mechanisation faithfully, the Barendregt convention is not used in the definition of contractiveness. However, we use it in the proof of Lemma 13, since it captures how we deal with de Bruijn indices in the mechanisation.

$$\mathsf{FV}(G^\mu) \triangleq \begin{cases} \mathsf{FV}(G_1^\mu) & \text{if } G^\mu = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \\ \bigcup_{1..j} \mathsf{FV}(G_j^\mu) & \text{if } G^\mu = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \\ \mathsf{FV}(G_1^\mu) \setminus \{\mathbf{t}\} & \text{if } G^\mu = \mu\mathbf{t}.G_1^\mu \\ \{\mathbf{t}\} & \text{if } G^\mu = \mathbf{t} \\ \{\} & \text{if } T^\mu = \mathsf{end}^\mu \end{cases}$$

$$\mathsf{FV}(T^\mu) \triangleq \begin{cases} \mathsf{FV}(T_1^\mu) & \text{if } T^\mu \in \{\ !^\mu k\langle U\rangle.T_1^\mu,\ ?^\mu k\langle U\rangle.T_1^\mu\ \} \\ \bigcup_{1..j} \mathsf{FV}(T_j^\mu) & \text{if } T^\mu \in \{\ k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J},\ k \&^\mu \{l_j : T_j^\mu\}_{j\in J}\ \} \\ \mathsf{FV}(T_1^\mu) \setminus \{\mathbf{t}\} & \text{if } T^\mu = \mu\mathbf{t}.T_1^\mu \\ \{\mathbf{t}\} & \text{if } T^\mu = \mathbf{t} \\ \{\} & \text{if } T^\mu = \mathsf{end}^\mu \end{cases}$$

$$\mathsf{closed}(T^\mu) \triangleq \mathsf{FV}(T^\mu) = \{\} \qquad \mathsf{closed}(G^\mu) \triangleq \mathsf{FV}(G^\mu) = \{\}$$

(a) Free recursion variables (FV) and closedness (closed) for global types and local types .

$$\mathsf{guardedVar}(\mathbf{t}, G^\mu) \triangleq \begin{cases} \mathsf{guardedVar}(\mathbf{t}, G_1^\mu) \ \wedge\ \mathbf{t} \neq \mathbf{t}' & \text{if } G^\mu = \mu\mathbf{t}'.G_1^\mu \\ \mathbf{t} \neq \mathbf{t}' & \text{if } G^\mu = \mathbf{t}' \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathsf{guardedVar}(\mathbf{t}, T^\mu) \triangleq \begin{cases} \mathsf{guardedVar}(\mathbf{t}, T_1^\mu) \ \wedge\ \mathbf{t} \neq \mathbf{t}' & \text{if } T^\mu = \mu\mathbf{t}'.T_1^\mu \\ \mathbf{t} \neq \mathbf{t}' & \text{if } T^\mu = \mathbf{t}' \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathsf{contr}(G^\mu) \triangleq \begin{cases} \mathsf{guardedVar}(\mathbf{t}, G_1^\mu) \ \wedge\ \mathsf{contr}(G_1^\mu) & \text{if } G^\mu = \mu\mathbf{t}.G_1^\mu \\ \mathsf{contr}(G_1^\mu) & \text{if } G^\mu = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \\ \forall j \in J.\ \mathsf{contr}(G_j^\mu) & \text{if } G^\mu = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathsf{contr}(T^\mu) \triangleq \begin{cases} \mathsf{guardedVar}(\mathbf{t}, T_1^\mu) \ \wedge\ \mathsf{contr}(T_1^\mu) & \text{if } T^\mu = \mu\mathbf{t}.T_1^\mu \\ \mathsf{contr}(T_1^\mu) & \text{if } T^\mu \in \{\ !^\mu k\langle U\rangle.T_1^\mu,\ ?^\mu k\langle U\rangle.T_1^\mu\ \} \\ \forall j \in J.\ \mathsf{contr}(T_j^\mu) & \text{if } T^\mu \in \{\ k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J}, \\ & \qquad k \&^\mu \{l_j : T_j^\mu\}_{j\in J}\ \} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

(b) Guarded variables ($\mathsf{guardedVar}$) and contractivess (contr) for global types and local types .

Fig. 1: Closedness and contractiveness

We create an auxiliary function $\mathsf{unfold}_1(\cdot)$ which unfolds a single binder by turning $\mu\mathbf{t}.G^\mu$ into $G^\mu[\mu\mathbf{t}.G^\mu/\mathbf{t}]$ and define the complete unfolding operation for global

$$|G^\mu| \triangleq \begin{cases} 1 + |G_1^\mu| & \text{if } G^\mu = \mu \mathbf{t}.G_1^\mu \\ 0 & \text{otherwise} \end{cases}$$

$$\mathsf{unfold}_1(G^\mu) \triangleq \begin{cases} G_1^\mu[\mu \mathbf{t}.G_1^\mu/\mathbf{t}] & \text{if } G^\mu = \mu \mathbf{t}.G_1^\mu \\ G^\mu & \text{otherwise} \end{cases}$$

$$\mathsf{unfold}(G^\mu) \triangleq \mathsf{unfold}_1^{|G^\mu|}(G^\mu)$$

Fig. 2: Calculating μ-height and unfolding global μ-types . Similar definitions for local μ-types exist , but have been elided.

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U \rangle.G_1^\mu \quad \mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{guarded}_\mathsf{p}^\mu(G_1^\mu)}{\mathsf{guarded}_\mathsf{p}^\mu(G^\mu)}$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j \in J} \quad \mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \forall j.\ \mathsf{guarded}_\mathsf{p}^\mu(G_j^\mu)}{\mathsf{guarded}_\mathsf{p}^\mu(G^\mu)}$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U \rangle.G_1^\mu \quad \mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{partOf}_\mathsf{p}^\mu(G_1^\mu)}{\mathsf{partOf}_\mathsf{p}^\mu(G^\mu)}$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j \in J} \quad \mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \exists j \in J.\ \mathsf{partOf}_\mathsf{p}^\mu(G_j^\nu)}{\mathsf{partOf}_\mathsf{p}^\mu(G^\mu)}$$

Fig. 3: The predicates $\mathsf{guarded}_\mathsf{p}^\mu(G^\mu)$ and $\mathsf{partOf}_\mathsf{p}^\mu(G^\mu)$ denote that p occurs in all branches of G^μ (resp. some branch of G^μ)

and local types, $\mathsf{unfold}(G^\mu)$ and $\mathsf{unfold}(T^\mu)$ respectively, as repeating the application $\mathsf{unfold}_1(G^\mu)$ and $\mathsf{unfold}_1(T^\mu)$ their μ-height times. The definitions of μ-height, unfold_1 and unfold are presented in Figure 2, where we write iterated function composition as f^n.

The $\mathsf{unfold}(\cdot)$ operator allows us to create predicates that work up to finite unfoldings of the μ-operator which is necessary to guarantee termination. We will go into this in more detail in Section 5 where we cover decidability properties.

We use unfolding to define the two predicates $\mathsf{guarded}_\mathsf{p}^\mu(G^\mu)$ (not to be confused with $\mathsf{guardedVar}(\mathbf{t}, G)$ used for defining contractiveness) and $\mathsf{partOf}_\mathsf{p}^\mu(G^\mu)$ where $\mathsf{guarded}_\mathsf{p}^\mu(G^\mu)$ means that a role p exists in all branches of G^μ and $\mathsf{partOf}_\mathsf{p}^\mu(G^\mu)$ means that p exists in at least one branch of G^μ. The definitions of both predicates is presented in Figure 3.

$$(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu)\upharpoonright^\mu_{\mathsf{p}} \triangleq \begin{cases} !^\mu k\langle U\rangle.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p}=\mathsf{p}_1 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ ?^\mu k\langle U\rangle.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{p}=\mathsf{p}_2 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ G^\mu\upharpoonright^\mu_{\mathsf{p}} & \text{if } \mathsf{p}\notin\{\mathsf{p}_1,\mathsf{p}_2\} \end{cases}$$

$$(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J})\upharpoonright^\mu_{\mathsf{p}} \triangleq \begin{cases} k\oplus^\mu \{l_j : (G^\mu_j\upharpoonright^\mu_{\mathsf{p}})\}_{j\in J} & \text{if } \mathsf{p}=\mathsf{p}_1 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ k\,\&^\mu \{l_j : (G^\mu_j\upharpoonright^\mu_{\mathsf{p}})\}_{j\in J} & \text{if } \mathsf{p}=\mathsf{p}_2 \text{ and } \mathsf{p}_1\neq\mathsf{p}_2 \\ G^\mu_1\upharpoonright^\mu_{\mathsf{p}} & \text{if } \mathsf{p}\notin\{\mathsf{p}_1,\mathsf{p}_2\} \text{ and } \forall i,j\in J.\ G^\mu_i\upharpoonright^\mu_{\mathsf{p}} = G^\mu_j\upharpoonright^\mu_{\mathsf{p}} \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

$$(\mu\mathbf{t}.G^\mu)\upharpoonright^\mu_{\mathsf{p}} \triangleq \begin{cases} \mu\mathbf{t}.(G^\mu\upharpoonright^\mu_{\mathsf{p}}) & \text{if } \mathsf{guardedVar}(\mathbf{t}, G^\mu\upharpoonright^\mu_{\mathsf{p}}) \\ \mathsf{end}^\mu & \text{otherwise} \end{cases} \qquad \mathbf{t}\upharpoonright^\mu_{\mathsf{p}} \triangleq \mathbf{t} \qquad \mathsf{end}^\mu\upharpoonright^\mu_{\mathsf{p}} \triangleq \mathsf{end}^\mu$$

Fig. 4: The standard projection of G onto p

2.3 Standard Projections

For each role, we use projection to relate global and local types. We start our overview with the standard projection proposed by Castro-Perez et al. [12], which is inductively defined by the rules given in Figure 4. The projection $(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.\ G^\mu)\upharpoonright^\mu_{\mathsf{p}}$ produces either a sending or a receiving action if the role p is equal to p_1 or p_2 respectively, otherwise the action is deleted. The projection of branching $(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G^\mu_j\}_{j\in J})\upharpoonright^\mu_{\mathsf{p}}$ works similarly but, when role p is not involved, all branches must project to exactly the same type. This requirement is known as *plain merge*. *Full merge*, used for example by Ghilezan et al. [7], is a more permissive operation which merges local types with distinct external choices. We discuss an extension of our work to full merge in Section 8. For recursion $\mu\mathbf{t}.G^\mu$, G^μ is projected only if the result is a contractive local type (checked by the $\mathsf{guardedVar}$ predicate). Finally, variable $\mathbf{t}$ and the type end^μ project directly to their local counterparts.

The use of $\mathsf{guardedVar}$ formally fixes a problem with the original projection [8] that could generate non-contractive types, which is unsound.

Alternatively, Demangeon and Yoshida [13] fix this issue by replacing the side condition with $G^\mu\upharpoonright^\mu_{\mathsf{p}} \neq \mathbf{t}$. However, both these projections invite the counterexample:

$$\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathsf{end}^\mu,\ l_2 : \mathbf{t}\}\upharpoonright^\mu_{\mathsf{p}} \tag{9}$$

which is undefined because the branch condition fails, as not all branches project to the same local type. Since p is not a role in the branch, the desired result of this projection should be $!^\mu k\langle U\rangle.\ \mathsf{end}^\mu$. Bejleri and Yoshida [14] solve this with a recursion

condition testing participation in the body:

$$(\mu \mathbf{t}.G^{\mu}) \upharpoonright^{\mu}_{\mathsf{p}} \triangleq \begin{cases} \mu \mathbf{t}.(G^{\mu} \upharpoonright^{\mu}_{\mathsf{p}}) & \text{if } \mathsf{p} \in G^{\mu} \\ \mathsf{end}^{\mu} & \textit{otherwise} \end{cases} \tag{10}$$

This function always generates contractive types, but the projection of

$$\mu \mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U \rangle.\ \mu \mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U' \rangle.\ \mathbf{t} \upharpoonright^{\mu}_{\mathsf{p}} \tag{11}$$

incorrectly results in the local type $\mu \mathbf{t}.!^{\mu}k\langle U \rangle.\mathsf{end}^{\mu}$ rather than the desired $\mu \mathbf{t}.!^{\mu}k\langle U \rangle.\ \mathbf{t}$. Glabbeek et al. [15] fix it by adding a variable constraint to the recursion condition:

$$(\mu \mathbf{t}.G^{\mu}) \upharpoonright^{\mu}_{\mathsf{p}} \triangleq \begin{cases} \mu \mathbf{t}.(G^{\mu} \upharpoonright^{\mu}_{\mathsf{p}}) & \\ \mathsf{end}^{\mu} & \text{if } \mathsf{p} \notin G^{\mu} \text{ and } \mu \mathbf{t}.G^{\mu} \text{ is closed} \end{cases} \tag{12}$$

This way, the projection in (11) correctly results in the type $\mu \mathbf{t}.!^{\mu}k\langle U \rangle.\mathbf{t}$. To the best of our knowledge, this is the most general and sound version of projection, but it still does not capture certain global types that intuitively should be projectable. One such example is (5), from the introduction:

$$\mu \mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U \rangle.\ \mu \mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U \rangle.\ \mathbf{t}'\} \upharpoonright^{\mu}_{\mathsf{p}}$$

Here, the branching condition fails because $\mathbf{t}$ is syntactically not the same type as $!^{\mu}k\langle U \rangle.\mathbf{t}'$. But how can we recognise that the two branches specify the same behavior for p even though their projections $\mathbf{t}$ and $!^{\mu}k\langle U \rangle.\mathbf{t}'$ are not the same? Our main insight is that standard projection can be performed in two steps: first, a translation function generates a candidate local type by structurally recursing under the μ-operators; then a boolean predicate tests projectability of the global type against this candidate by unfolding μ-operators. The procedure returns the candidate if the check is passed and is undefined otherwise. Checking projectability by unfolding μ-operators makes termination non-trivial and we explore this in Section 5. This approach lets us recognise (5) as projectable, and its projection on p is the following local type:

$$\mu \mathbf{t}.!^{\mu}k\langle U \rangle.\mu \mathbf{t}'.\mathbf{t} \tag{13}$$

The local type above is derived by using the Castro-Perez et al. projection *without* checking whether branches project to the same local type. In the sequel, we explain why (13) is a correct projection of (5).

3 Coinductive Types and Coinductive Projection

We have seen that inductive μ-types can be used to write specifications that use repetitions. One such specification is the global type from Equation (5), which cannot be handled by standard projections due to μ-binders. In their seminal work, Ghilezan

$$\frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu})} \qquad \frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \forall j.\ \mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu}_j)}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J})}$$

$$\frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})}{\mathsf{partOf}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu})} \qquad \frac{\mathsf{p} \in \{\mathsf{p}_1, \mathsf{p}_2\} \vee \exists j \in J.\ \mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu}_j)}{\mathsf{partOf}^{\nu}_{\mathsf{p}}(\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J})}$$

Fig. 5: Definitions of predicates $\mathsf{guarded}^{\nu}$ 🐓 and partOf^{ν} 🐓.

et al. [7] propose to define projection on type trees [4]. Type trees is an approach that transforms a μ-type into a partial function from paths to nodes in the tree that correspond to constructors of the inductive type. Crucially, in the traversal of the path, the μ-binder is unfolded when it occurs at top-level. Therefore type trees offer a representation of μ-types that hides the μ-binder. This is important in our setting about projection, because it is the μ-binder that makes projection nontrivial. Inspired by their work, this section introduces a coinductive projection. We however follow the presentation by Castro-Perez et al. who use coinductive types, rather than type trees.

3.1 Syntax

The coinductive syntax of global and local types is defined as follows:

Definition 2 (Coinductive Types)**.** *The syntax of coinductive global types denoted by* G^{ν} 🐓 *and coinductive local types denoted by* T^{ν} 🐓*, is coinductively defined as:*

$$G^{\nu} ::= \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^{\nu} \mid \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^{\nu}_j\}_{j\in J} \mid \mathsf{end}^{\nu}$$

$$T^{\nu} ::= !^{\nu}k\langle U\rangle.T^{\nu} \mid ?^{\nu}k\langle U\rangle.T^{\nu} \mid k \oplus^{\nu} \{l_j : T^{\nu}_j\}_{j\in J} \mid k \,\&^{\nu}\, \{l_j : T^{\nu}_j\}_{j\in J} \mid \mathsf{end}^{\nu}$$

Coinductive types are objects that can be infinitely large. However, the fragment of types we are interested in is regular since these types have only a finite set of distinct subterms [16]. As a consequence, infinite objects must be circular. For example, the coinductive global type $G^{\nu}_{reg} = \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.G^{\nu}_{reg}$ is regular and has two distinct subterms. The μ-types we have defined can only represent regular coinductive types so naturally we are only interested in those. It is however not necessary to impose any such restriction on G^{ν} or T^{ν} because this property will be enforced by an unravelling relation, which relates μ-types with their corresponding coinductive types.

3.2 Predicates on Coinductive Global Types

We define the predicates $\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})$ and $\mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})$. Unlike their inductive counterparts from Section 2, they are not unfolding predicates, as coinductive types have no μ-binders to unfold. Due to the structural similarity of inductive and coinductive types, the definitions are however very similar. The predicates $\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})$ and $\mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})$ are formally defined in Figure 5. Intuitively, $\mathsf{partOf}^{\nu}_{\mathsf{p}}(G^{\nu})$ checks the occurence of a role in any branch, while $\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})$ checks the occurence of a role in all branches.

$$\frac{}{G^\nu \downarrow^\nu_{\mathsf{p}} \; T^\nu}\;[\mathsf{M1}\downarrow^\nu] \qquad \frac{}{G^\nu \downarrow^\nu_{\mathsf{p}} \; T^\nu}\;[\mathsf{M2}\downarrow^\nu]$$
$$\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p} : k\langle u\rangle.G^\nu \;\downarrow^\nu_{\mathsf{p}}\; ?^\nu k\langle U\rangle.T^\nu \qquad \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu \;\downarrow^\nu_{\mathsf{p}}\; !^\nu k\langle U\rangle.T^\nu$$

$$\frac{\mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \qquad \mathsf{guarded}^\nu_{\mathsf{p}}(G^\nu) \qquad G^\nu \downarrow^\nu_{\mathsf{p}} \; T^\nu}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu \downarrow^\nu_{\mathsf{p}} \; T^\nu}\;[\mathsf{M}\downarrow^\nu] \qquad \frac{\neg \mathsf{partOf}^\nu_{\mathsf{p}}(G^\nu)}{G^\nu \downarrow^\nu_{\mathsf{p}} \; \mathsf{end}^\nu}\;[\mathsf{End}\downarrow^\nu]$$

$$\frac{\forall j.\; G^\nu_j \downarrow^\nu_{\mathsf{p}} \; T^\nu_j}{\mathsf{p} \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^\nu_j\}_{j\in J} \;\downarrow^\nu_{\mathsf{p}}\; k \oplus^\nu \{l_j : T^\nu_j\}_{j\in J}}\;[\mathsf{B1}\downarrow^\nu]$$

$$\frac{\forall j.\; G^\nu_j \downarrow^\nu_{\mathsf{p}} \; T^\nu_j}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p} : k\{l_j : G^\nu_j\}_{j\in J} \downarrow^\nu_{\mathsf{p}} \; k \,\&^\nu\, \{l_j : T^\nu_j\}_{j\in J}}\;[\mathsf{B2}\downarrow^\nu]$$

$$\frac{\mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \qquad \forall j.\; G^\nu_j \downarrow^\nu_{\mathsf{p}} \; T^\nu \;\wedge\; \mathsf{guarded}^\nu_{\mathsf{p}}(G^\nu_j)}{\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G^\nu_j\}_{j\in J} \downarrow^\nu_{\mathsf{p}} \; T^\nu}\;[\mathsf{B}\downarrow^\nu]$$

Fig. 6: Coinductive projection on coinductive types .

3.3 Coinductive Projection

Inspired by Ghilezan et al. [7], we give a coinductive definition of projection that relates coinductive global types to coinductive local types. The coinductive projection, denoted by $G^\nu \downarrow^\nu_{\mathsf{p}} \; T^\nu$, is a coinductive relation on global and local types. The relation is partial, relating only some global types to a local type. It is defined by the rules given in Figure 6. Rules $[\mathsf{M1}\downarrow^\nu]$, $[\mathsf{M2}\downarrow^\nu]$, $[\mathsf{B1}\downarrow^\nu]$, and $[\mathsf{B2}\downarrow^\nu]$ handle the cases where a projected role p takes part in communication or branching. Note that our projection allows sender and receiver in a communication to be equal. Rules $[\mathsf{M}\downarrow^\nu]$, $[\mathsf{B}\downarrow^\nu]$, and $[\mathsf{End}\downarrow^\nu]$ handle the cases where p does not take part using the negation of $\mathsf{partOf}^\nu_{\mathsf{p}}(G^\nu)$. In these cases, in order for projection to continue, p must occur in all possible future branches, otherwise the projection returns end. These rules are similar to those given by Castro-Perez et al. [12] as well as Jacobs et al. [17]. Our projection uses $\mathsf{guarded}^\nu_{\mathsf{p}}(G^\nu)$, in the $[\mathsf{M}\downarrow^\nu]$ and $[\mathsf{B}\downarrow^\nu]$ rules, to avoid vacuous derivations.

Example 1 (Vacuous Derivation)**.** *If the rule* $[M\downarrow^\nu]$ *did not have* $\mathsf{guarded}^\nu_{\mathsf{p}}(G^\nu)$ *in its premise, we could incorrectly derive that the projection of* r *on* $G^\nu \triangleq \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.G^\nu$ *is related to some coinductive local type. This is a vacuous derivation because the shape of the local type is not inspected during the derivation. As an example, we will use the coinductive local type* $!^\nu k\langle U\rangle.\mathsf{end}^\nu$*, which we know is an incorrect projection of the global type* G^ν *on* r*. If we do not enforce* $\mathsf{guarded}^\nu_{\mathsf{p}}(G^\nu)$ *in the premise of* $[M\downarrow^\nu]$*, the following incorrect projection is derivable:*

$$\cdots \;\Big\}\; \frac{G^\nu \downarrow^\nu_{\mathsf{r}} \; !^\nu k\langle U\rangle.\mathsf{end}^\nu}{G^\nu \downarrow^\nu_{\mathsf{r}} \; !^\nu k\langle U\rangle.\mathsf{end}^\nu}\;[M\downarrow^\nu] \tag{14}$$

The arrow marks a cycle from the premise back to the conlusion; this is possible because coinductive definitions may be circular. Since no constraints are imposed on the local type, this is a vacuous derivation that admits any coinductive local type. The use of $\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})$ *in rule* $[M\downarrow^{\nu}]$ *disallows this derivation because we cannot derive* $\mathsf{guarded}^{\nu}_{\mathsf{r}}(G^{\nu})$.

Example 2. *Intuitively, the inductive global type in Equation (5)* $\mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t},\quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$ *corresponds to the coinductive type*

$$G^{\nu} \triangleq \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^{\nu},\quad l_2 : G^{\nu}\} \tag{15}$$

Similarly, the inductive local type $\mu\mathbf{t}.!^{\mu}k\langle U\rangle.\mu\mathbf{t}'.\mathbf{t}$ *from Equation (13), which we claim is the* p *projection of the global type in Equation (5), corresponds to the coinductive type*

$$T^{\nu} \triangleq !^{\nu}k\langle U\rangle.\ T^{\nu} \tag{16}$$

We can derive the corresponding coinductive types of inductive global type (5) and local type (13), and relate them by coinductive projection on p*:*

$$\dfrac{\dfrac{}{\mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^{\nu},\quad l_2 : G^{\nu}\} \downarrow^{\nu}_{\mathsf{p}}\ T^{\nu}}\ [B\downarrow^{\nu}]}{\mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^{\nu},\quad l_2 : G^{\nu}\} \downarrow^{\nu}_{\mathsf{p}}\ !^{\nu}k\langle U\rangle.\ T^{\nu}}\ [M1\downarrow^{\nu}] \tag{17}$$

We mark cycles where premises and conclusions match. In this particular case, we marked two cycles, one for each of the branches in the global type. Note that the application of $[B\downarrow^{\nu}]$ *is possible because of the following derivation:*

$$\dfrac{\dfrac{\mathsf{p} \in \{\mathsf{p},\mathsf{q}\}}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})}\quad \dfrac{\mathsf{p} \in \{\mathsf{p},\mathsf{q}\}}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(G^{\nu})}}{\mathsf{guarded}^{\nu}_{\mathsf{p}}(\mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^{\nu},\quad l_2 : G^{\nu}\})}$$

3.4 Relating Inductive and Coinductive types

To reason effectively about μ-types and their coinductive counterparts, we need a means to relate the two. We follow the style of Castro-Perez et al. [12], defining an unravelling relation for global types denoted by $G^{\mu}\ \mathcal{R}\ G^{\nu}$ and an unravelling of local types denoted by $T^{\mu}\ \mathcal{R}\ T^{\nu}$. These relations are used to relate inductive and coinductive types. This is because the relations essentially map the inductive type onto the coinductive type by repeatedly unfolding the inductive type. This limits the number of distinct subterms that can be represented finitely by a μ-type. Our unravelling relation is intentionally more restrictive than that of Castro-Perez et al.. We do this to ensure unravelling subsumes well-formedness of inductive types, which Castro-Perez et al. enforce separately. Formally, *unravelling* of global types (resp. local types), denoted by $G^{\mu}\ \mathcal{R}\ G^{\nu}$ (resp. $T^{\mu}\ \mathcal{R}\ T^{\nu}$), is defined by the rules reported in Figure 7. Intuitively,

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{end}^\mu}{G^\mu \;\mathcal{R}\; \mathsf{end}^\nu}\;[\mathcal{R}\text{-G-END}] \qquad \frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \quad G_1^\mu \;\mathcal{R}\; G^\nu}{G^\mu \;\mathcal{R}\; \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu}\;[\mathcal{R}\text{-MSG}]$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \quad \forall j \in J.\; G_j^\mu \;\mathcal{R}\; G_j^\nu}{G^\mu \;\mathcal{R}\; \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : G_j^\nu\}_{j\in J}}\;[\mathcal{R}\text{-BRANCH}]$$

$$\frac{\mathsf{unfold}(T^\mu) = !^\mu k\langle U\rangle.T_1^\mu \quad T_1^\mu \;\mathcal{R}\; T^\nu}{T^\mu \;\mathcal{R}\; !^\nu k\langle U\rangle.T^\nu}\;[\mathcal{R}\text{-!}] \qquad \frac{\mathsf{unfold}(T^\mu) = ?^\mu k\langle U\rangle.T_1^\mu \quad T_1^\mu \;\mathcal{R}\; T^\nu}{T^\mu \;\mathcal{R}\; ?^\nu k\langle U\rangle.T^\nu}\;[\mathcal{R}\text{-?}]$$

$$\frac{\mathsf{unfold}(T^\mu) = \mathsf{end}^\mu}{T^\mu \;\mathcal{R}\; \mathsf{end}^\nu}\;[\mathcal{R}\text{-T-END}] \qquad \frac{\mathsf{unfold}(T^\mu) = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \quad \forall j \in J.\; T_j^\mu \;\mathcal{R}\; T_j^\nu}{T^\mu \;\mathcal{R}\; k \oplus^\nu \{l_j : T_j^\nu\}_{j\in J}}\;[\mathcal{R}\text{-}\oplus]$$

$$\frac{\mathsf{unfold}(T^\mu) = k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J} \quad \forall j \in J.\; T_j^\mu \;\mathcal{R}\; T_j^\nu}{T^\mu \;\mathcal{R}\; k \,\&^\nu\, \{l_j : T_j^\nu\}_{j\in J}}\;[\mathcal{R}\text{-}\&]$$

Fig. 7: The Unravelling Relation for global types ↯ and local types ↯.

unravelling maps the unfolded inductive global (resp. local) type onto the coinductive type. As expected, the predicates $\mathsf{guarded}^\mu_\mathsf{p}(G^\mu)$, $\mathsf{partOf}^\mu_\mathsf{p}(G^\mu)$, $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu)$ and $\mathsf{partOf}^\nu_\mathsf{p}(G^\nu)$ are equivalent notions between inductive and coinductive types.

Lemma 1. *Let G^μ be an inductive global type, G^ν a coinductive global type, and p a role. Then,*

- ↯ *If $G^\mu \;\mathcal{R}\; G^\nu$ then $\mathsf{guarded}^\mu_\mathsf{p}(G^\mu)$ if and only if $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu)$*
- ↯ *If $G^\mu \;\mathcal{R}\; G^\nu$ then $\mathsf{partOf}^\mu_\mathsf{p}(G^\mu)$ if and only if $\mathsf{partOf}^\nu_\mathsf{p}(G^\nu)$*

The following example shows that our definition of unravelling is more strict than the definition used by Castro-Perez et al.

Example 3 (Unravelling $\mu\mathbf{t}.\mathbf{t}$)**.** *The unravelling of Castro-Perez et al. is more permissive than our definition due to them having the following rule:*

$$\frac{G^\mu[\mu\mathbf{t}.G^\mu/\mathbf{t}] \;\mathcal{R}\; G^\nu}{\mu\mathbf{t}.G^\mu \;\mathcal{R}\; G^\nu}$$

With this rule one can derive that non-contractive $\mu\mathbf{t}.\mathbf{t}$ unravels to any coinductive type G^ν:

$$\frac{}{\mu\mathbf{t}.\mathbf{t} \;\mathcal{R}\; G^\nu} \qquad (18)$$

It is not possible to derive $\mu\mathbf{t}.\mathbf{t} \;\mathcal{R}\; G^\nu$ with our rules because $\mu\mathbf{t}.\mathbf{t}$ unfolds to itself, that is $\mathsf{unfold}(\mu\mathbf{t}.\mathbf{t}) = \mu\mathbf{t}.\mathbf{t}$ and there is no rule for this case.

Castro-Perez et al. rule out problematic cases like the one in Example 3 by imposing a contractiveness side condition when μ-types are unravelled. Recall that we did not take the standard approach that assumes μ-types are well-formed. This is not necessary because it is subsumed by unravelling. We show this by defining the corecursive

procedures $\mathsf{tocoind}(G^\mu)$ and $\mathsf{tocoind}(T^\mu)$ that respectively generate coinductive global types and coinductive local types from their inductive counterpart.

Definition 3 ($\mathsf{tocoind}$)**.** *The corecursive functions* $\mathsf{tocoind} : G^\mu \to G^\nu$ 🦈 *and* $\mathsf{tocoind} : T^\mu \to T^\nu$ 🦈 *are defined as:*

$$\mathsf{tocoind}(G^\mu) \triangleq \begin{cases} \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.\mathsf{tocoind}(G^\mu) & \text{if } \mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu \\ \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\{l_j : \mathsf{tocoind}(G_j^\mu)\}_{j\in J} & \text{if } \mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \\ \mathsf{end}^\nu & \text{otherwise} \end{cases}$$

$$\mathsf{tocoind}(T^\mu) \triangleq \begin{cases} !^\nu k\langle U\rangle.\mathsf{tocoind}(T^\mu) & \text{if } \mathsf{unfold}(T^\mu) = !^\mu k\langle U\rangle.T^\mu \\ ?^\nu k\langle U\rangle.\mathsf{tocoind}(T^\mu) & \text{if } \mathsf{unfold}(T^\mu) = ?^\mu k\langle U\rangle.T^\mu \\ k \oplus^\nu \{l_j : \mathsf{tocoind}(T_j^\mu)\}_{j\in J} & \text{if } \mathsf{unfold}(T^\mu) = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \\ k \,\&^\nu \{l_j : \mathsf{tocoind}(T_j^\mu)\}_{j\in J} & \text{if } \mathsf{unfold}(T^\mu) = k \,\&^\mu \{l_j : T_j^\mu\}_{j\in J} \\ \mathsf{end}^\nu & \text{otherwise} \end{cases}$$

We can now state that well-formedness of inductive global and local types is equivalent to being able to unravel them to the coinductive type generated by applying $\mathsf{tocoind}(\cdot)$.

Proposition 2. *Let* G^μ *and* T^μ *be an inductive global type and an inductive local type respectively. Then,*

1. 🦈 $\mathsf{closed}(G^\mu) \wedge \mathsf{contr}(G^\mu)$ *if and only if* $G^\mu \mathrel{\mathcal{R}} \mathsf{tocoind}(G^\mu)$
2. 🦈 *If* $\mathsf{closed}(T^\mu) \wedge \mathsf{contr}(T^\mu)$ *then* $T^\mu \mathrel{\mathcal{R}} \mathsf{tocoind}(T^\mu)$

Proof. For (1), the only-if case is by coinduction on $\mathcal{R}$. The if case is challenging. The proof is by contradiction, assuming $G^\mu \mathrel{\mathcal{R}} G^\nu$ and that G^μ is not contractive. For (2), the proof is analogous to the only-if case of (1). □

Proposition 2 shows that well-formedness can be characterised both structurally in terms of closedness and contractiveness, and also coinductively in terms of unravelling. For the proof of completeness we need the if-direction. We prove the other direction to demonstrate that the two characterisations are equivalent, and because closedness and contractiveness is an efficient way to derive the premise $G^\mu \mathrel{\mathcal{R}} \mathsf{tocoind}(G^\mu)$ of rule $[\mathsf{End}\downarrow^\mu]$ in our decision procedure (see next section, Definition 6). The proof of the if-direction is quite involved and because we do not need it, we do not mechanise this direction for local types.

Example 4. 🦈 *Consider the unravelling of the inductive global type from Equation (5) to the coinductive global type from Equation (15). We informally related these types in Example (2) and now do it formally with unravelling. First, we recall their definitions:*

$$G^\mu \triangleq \mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$$
$$G^\nu \triangleq \mathsf{p} \xrightarrow{\nu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^\nu, \quad l_2 : G^\nu\}$$

Additionally, for the sake of presentation, we introduce the following shorthand:

$$G_1^\mu \triangleq \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : G^\mu, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$$

We can then derive $G^\mu \mathcal{R} G^\nu$ *as follows:*

$$\dfrac{\dfrac{G^\mu \mathcal{R} G^\nu \qquad \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.G_1^\mu \mathcal{R} G^\nu}{G_1^\mu \mathcal{R} \mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^\nu, \quad l_2 : G^\nu\}} \quad G_1^\mu \mathcal{R} \mathsf{r} \xrightarrow{\nu} \mathsf{s} : k'\{l_1 : G^\nu, \quad l_2 : G^\nu\}}{G^\mu \mathcal{R} G^\nu} \tag{19}$$

Example 5. *Consider the unravelling of the inductive local type in Equation (13) to the coinductive local type in Equation (16). We informally related these types in Example 2 and now do this formally with unravelling. We recall their definitions:*

$$T^\mu \triangleq \mu \mathbf{t}.!^\mu k\langle U\rangle.\mu \mathbf{t}'.\mathbf{t} \qquad\qquad T^\nu \triangleq !^\nu k\langle U\rangle.\, T^\nu$$

We now derive $T^\mu \mathcal{R} T^\nu$ *as follows:*

$$\dfrac{\dfrac{\mu \mathbf{t}'.T^\mu \mathcal{R} T^\nu}{\mu \mathbf{t}'.T^\mu \mathcal{R} T^\nu}}{T^\mu \mathcal{R} T^\nu} \tag{20}$$

Examples 2, 4 and 5 precisely justify why we wish to project the inductive global type in Equation (5) over role p to the local type in Equation (13). The justification is that we can derive that their unravellings are related by coinductive projection.

4 Projection on Inductive Types

We now present a new computable projection on inductive global types, denoted by $\mathsf{proj}_\mathsf{p}(G^\mu)$. Our main idea is to keep the syntactic translation performed by standard projection but remove the syntactic side condition that requires all branches to be equal. Standard projection without the syntactic side condition becomes then a total function, an operation we call translation, which is defined by the following definition:

Definition 4 (trans)**.** *The function* $\mathsf{trans} : \mathcal{P} \to G^\mu \to T^\mu$ *is identical to the function* $\upharpoonright^\mu$ *(see Figure 4) except for the branching case, defined as:*

$$\mathsf{trans}_\mathsf{p}(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J}) = \begin{cases} k \oplus^\mu \{l_j : \mathsf{trans}_\mathsf{p}(G_j^\mu)\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \\ k \,\&^\mu\, \{l_j : \mathsf{trans}_\mathsf{p}(G_j^\mu)\}_{j\in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \\ \mathsf{trans}_\mathsf{p}(G_1^\mu) & \text{otherwise} \end{cases}$$

We observe that the total function $\mathsf{trans}_\mathsf{p}(G^\mu)$ is a complete but unsound projection function. Totality makes this an unsound projection function.

Example 6. *The total function* $\mathsf{trans}_{\mathsf{p}}(G^{\mu})$ *is an unsound projection function. Consider the following global type:*

$$G^{\mu}_{bad} \triangleq \mathsf{p} \xrightarrow{\mu} \mathsf{q}\colon k\left\{\begin{array}{l}\mathtt{Left}\colon \mathsf{p} \xrightarrow{\mu} \mathsf{r} : k'\langle \mathtt{Int}\rangle.\mathsf{end}^{\mu} \\ \mathtt{Right}\colon \mathsf{p} \xrightarrow{\mu} \mathsf{r} : k'\langle \mathtt{String}\rangle.\mathsf{end}^{\mu}\end{array}\right\} \tag{21}$$

This global type is not projectable, and this is because the type of the message that will be received by r *depends on the branching which* r *is not involved in. Translating this global type onto* r *produces the following local type:*

$$!^{\mu}k'\langle \mathtt{Int}\rangle.\mathsf{end}^{\mu}$$

This local type was produced by translating the `Left` *branch of the global type.*

We avoid (21) by applying $\mathsf{trans}_{\mathsf{p}}$ only when the side condition $\mathsf{projectableB}_{\mathsf{p}}(G^{\mu})$ is satisfied. This is seen in our projection function $\mathsf{proj}_{\mathsf{p}}(G^{\mu})$ defined below.

Definition 5 (proj)**.** *The function* $\mathsf{proj} : \mathcal{P} \to G^{\mu} \rightharpoonup T^{\mu}$*, written* $\mathsf{proj}_{\mathsf{p}}(G^{\mu})$*, is the projection of the global* μ*-type* G^{μ} *with respect to the role* p *and is defined as:*

$$\mathsf{proj}_{\mathsf{p}}(G^{\mu}) = \begin{cases} \mathsf{trans}_{\mathsf{p}}(G^{\mu}) & \textit{if } \mathsf{projectable}_{\mathsf{p}}(G^{\mu}) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

We define a decision procedure which relies on a transition function $\mathsf{next}_{\mathsf{p}}(G^{\mu})$ which is defined below.

Definition 6 (projectable)**.** *The functions* $\mathsf{next} : \mathcal{P} \times G^{\mu} \times T^{\mu} \rightharpoonup \{G^{\mu} \times T^{\mu}\}$, $\mathsf{nextunf} : \mathcal{P} \times G^{\mu} \times T^{\mu} \rightharpoonup \{G^{\mu} \times T^{\mu}\}$, $\mathsf{dec} : \mathcal{P} \times \{G^{\mu} \times T^{\mu}\} \to G^{\mu} \times T^{\mu} \to \{\mathsf{True}, \mathsf{False}\}$

and $\mathsf{projectable} : \mathcal{P} \times G^\mu \to \{\mathsf{True}, \mathsf{False}\}$ *are such that:*

$$\mathsf{next}_\mathsf{p}(G,T) = \begin{cases} \{(G_1,T_1)\} & \text{if } G^\mu = \mathsf{p} \to \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \ \wedge \\ & T^\mu = !^\mu k\langle U\rangle.T_1^\mu \\ \{(G_1,T_1)\} & \text{if } G^\mu = \mathsf{p}_1 \to \mathsf{p} : k\langle U\rangle.G_1^\mu \ \wedge \\ & T^\mu = ?^\mu k\langle U\rangle.T_1^\mu \\ \{(G_1,T)\} & \text{if } G^\mu = \mathsf{p}_1 \to \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \ \wedge \mathsf{p} \notin \{\mathsf{p}_1,\mathsf{p}_2\} \wedge \mathsf{guarded}_\mathsf{p}^\mu(G^\mu) \\ \{G_j,T_j\}_{j\in J} & \text{if } G^\mu = \mathsf{p} \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \ \wedge \\ & T^\mu = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J} \\ \{G_j,T_j\}_{j\in J} & \text{if } G^\mu = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p} : k\{l_j : G_j^\mu\}_{j\in J} \ \wedge \\ & T^\mu = k \ \&^\mu \{l_j : T_j^\mu\}_{j\in J} \\ \{(G_j,T)\}_{j\in J} & \text{if } G^\mu = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \ \wedge \mathsf{p} \notin \{\mathsf{p}_1,\mathsf{p}_2\} \wedge \\ & \mathsf{guarded}_\mathsf{p}^\mu(G_j), j \in J \\ \{\} & \text{if } \mathsf{closed}(G^\mu) \wedge \mathsf{contr}(G^\mu) \wedge T^\mu = \mathsf{end}^\mu \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

$$\mathsf{nextunf}_\mathsf{p}(G^\mu, T^\mu) \triangleq \mathsf{next}_\mathsf{p}(\mathsf{unfold}(G^\mu), \mathsf{unfold}(T^\mu))$$

$$\mathsf{dec}_p(V, G^\mu, T^\mu) \triangleq \begin{cases} \mathsf{True} & \text{if } (G^\mu,T^\mu) \in V \\ \forall i \in I. \\ \ \mathsf{dec}(p, V \cup \{(G^\mu,T^\mu), G_i^\mu, T_i^\mu) & \text{if } \mathsf{nextunf}_\mathsf{p}(G^\mu,T^\mu) = \{(G_i^\mu,T_i^\mu)\}_{i\in I} \\ \mathsf{False} & \textit{otherwise} \end{cases}$$

$$\mathsf{projectable}_\mathsf{p}(G^\mu) \triangleq \mathsf{dec}_p(\{\}, G^\mu, \mathsf{trans}_\mathsf{p}(G^\mu))$$

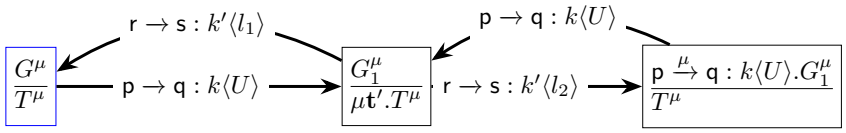
Intuitively $\mathsf{next}_\mathsf{p}(G^\mu,T^\mu)$ corresponds to a rule application of the coinductive projection $G^\nu \ \downarrow_\mathsf{p}^\nu \ T^\nu$. The crucial difference between inductive and coinductive types is that the former must be unfolded. For this reason we wrap $\mathsf{next}_\mathsf{p}(G^\mu,T^\mu)$ in $\mathsf{nextunf}_\mathsf{p}(G^\mu,T^\mu)$ which applies unfolding before attempting a rule application. This transition function is then repeatedly applied by our decision procedure $\mathsf{dec}_\mathsf{p}(V,G^\mu,T^\mu)$, terminating either if the transition function is undefined, or if the current pair has already been seen, a property that is checked by accumulating the visited set $V : \{G^\mu \times T^\mu\}$.

We now show an example of how this procedure determines projectability.

Example 7. *Let us consider the following μ-types:*

$$\begin{aligned} G^\mu &= \mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\} \\ G_1^\mu &= \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : G^\mu, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\} \\ T^\mu &= \mu\mathbf{t}.!^\mu k\langle U\rangle.\mu\mathbf{t}'.\mathbf{t} \end{aligned}$$

The following graphs plots the execution of $\mathsf{projectable}_\mathsf{p}(G^\mu)$*, where we have marked the initial state in blue. Edges represent steps by* $\mathsf{nextunf}_\mathsf{p}$*. Those edges indicated with curved arrows return to pairs previously seen.*



4.1 Termination

We now prove the termination of $\mathsf{projectable}_{\mathsf{p}}(G^\mu)$. In order to do so, we first introduce enumerations on global and local types:

Definition 7 (Enumeration)**.** *The functions* $\mathit{enum}_g : G^\mu \to \{G^\mu\}$ *and* $\mathit{enum}_l : T^\mu \to \{T^\mu\}$ *are such that:*

$$\mathit{enum}_g(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu) = \{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu\} \cup \mathit{enum}_g(G^\mu) \qquad \mathit{enum}_g(\mathsf{end}^\mu) = \{\mathsf{end}^\mu\}$$
$$\mathit{enum}_g(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J}) = \{\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J}\} \cup \bigcup_{j\in J} \mathit{enum}_g(G_j^\mu)$$
$$\mathit{enum}_g(\mathbf{t}) = \{\mathbf{t}\} \qquad \mathit{enum}_g(\mu\mathbf{t}.G^\mu) = \{\mu\mathbf{t}.G^\mu\} \cup \{G_1^\mu[\mu\mathbf{t}.G^\mu/\ \mathbf{t}] \mid G_1^\mu \in \mathit{enum}_g(G^\mu)\}$$

$$\mathit{enum}_l(!^\mu k\langle U\rangle.T^\mu) = \{!^\mu k\langle U\rangle.T^\mu\} \cup \mathit{enum}_l(T^\mu) \qquad \mathit{enum}_l(\mathsf{end}^\mu) = \{\mathsf{end}^\mu\}$$
$$\mathit{enum}_l(?^\mu k\langle U\rangle.T^\mu) = \{?^\mu k\langle U\rangle.T^\mu\} \cup \mathit{enum}_l(T^\mu)$$
$$\mathit{enum}_l(k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J}) = \{k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J}\} \cup \bigcup_{j\in J} \mathit{enum}_g(T_j^\mu)$$
$$\mathit{enum}_l(k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J}) = \{k \,\&^\mu\, \{l_j : T_j^\mu\}_{j\in J}\} \cup \bigcup_{j\in J} \mathit{enum}_g(T_j^\mu)$$
$$\mathit{enum}_l(\mathbf{t}) = \{\mathbf{t}\} \qquad \mathit{enum}_l(\mu\mathbf{t}.T^\mu) = \{\mu\mathbf{t}.T^\mu\} \cup \{T_1^\mu[\mu\mathbf{t}.T^\mu/\ \mathbf{t}] \mid T_1^\mu \in \mathit{enum}_l(T^\mu)\}$$

$$\mathit{enum}(G^\mu, T^\mu) = \mathit{enum}_g(G^\mu) \times \mathit{enum}_l(T^\mu)$$

The enumeration functions enum_g (respectively enum_l) collect all subterms of a global type (respectively local type). In the case of $\mu\mathbf{t}.G^\mu$, each function enumerates all subterms of the body G^μ (respectively T^μ), which might contain free occurrences of $\mathbf{t}$, and substitute them all for $\mu\mathbf{t}.G^\mu$ (respectively $\mu\mathbf{t}.T^\mu$).

Example 8 (Enumeration of running example)**.** *We apply* enum_g *to the following global type:*

$$G^\mu = \mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'$$

And using the following shorthand:

$$G_1^\mu = \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : G^\mu, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'$$

The resulting set from $\mathit{enum}_g(G^\mu)$ *is*

$$\left\{G^\mu\ ,\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ G_1^\mu\ ,\ G_1^\mu\ ,\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : G^\mu, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ G_1^\mu\}\right\} \quad (22)$$

Enumerations are then used to define a termination measure M for dec.

Definition 8 (Termination measure)**.** *The measure* $M : \{G^\mu \times T^\mu\} \to G^\mu \to T^\mu \to \mathbb{N}$ *is such that:* $M(V, G^\mu, T^\mu) = |\mathit{enum}(G^\mu, T^\mu) \setminus V|$

For this to be a valid termination measure, it must decrease for the recursive call. A recursive call is performed exactly when the initial pair is not in the visited set *and* the next function is defined. The measure computed on the returned pairs will be smaller because the visited set (subtracted with) increases, and the enumeration set (subtracted from) does not increase. Formally we say that enumeration is closed under the next function.

Lemma 3 (Enumeration closure)**.** *If* $(G_1^\mu, T_1^\mu) \in \mathsf{next_p}(\mathsf{unfold}(G^\mu), \mathsf{unfold}(T^\mu))$ *then* $\mathit{enum}(G_1^\mu, T_1^\mu) \subseteq \mathit{enum}(G^\mu, T^\mu)$

Proof. We first observe that a similar property where we do not unfold G^μ and T^μ is immediate:

$$(G_1^\mu, T_1^\mu) \in \mathsf{next_p}(G^\mu, T^\mu) \implies \mathsf{enum}(G_1^\mu, T_1^\mu) \subseteq \mathsf{enum}(G^\mu, T^\mu)$$

This is because enum computes all subterms and the next function returns the direct subterms. Thus, applying the enumeration function to a subterm yields no new subterms. Convinced that the next function does not introduce new pairs, it suffices to show that unfolding does not introduce new pairs. Formally,

$$\mathsf{enum}(\mathsf{unfold}(G^\mu), \mathsf{unfold}(T^\mu)) \subseteq \mathsf{enum}(G^\mu, T^\mu)$$

Recall that $\mathsf{unfold}(\cdot)$ is defined as a repeated application of $\mathsf{unfold}_1(\cdot)$ and that enumeration of a pair is just the Cartesian product, it thus suffices to show a similar property for $\mathsf{unfold}_1(\cdot)$ that will be equivalent for global types and local types. Stated for global types, the property is

$$\mathsf{enum}_g(\mathsf{unfold}(G^\mu)) \subseteq \mathsf{enum}(G^\mu)$$

The interesting case is $G^\mu = \mu\mathbf{t}.G_1^\mu$, which we show now. One must show that

$$\mathsf{enum}_g(G_1^\mu[\mu\mathbf{t}.G_1^\mu/\mathbf{t}]) \subseteq \mathsf{enum}_g(\mu\mathbf{t}.G_1^\mu)$$

To proceed we requires a substitution lemma about enumerations. Using the shorthand $\big(\mathsf{enum}_g(G_1^\mu)\big)[G^\mu/\mathbf{t}]$ for $\{G_2^\mu[G^\mu/\mathbf{t}] \mid G_2^\mu \in \mathsf{enum}_g(G_1^\mu)\}$ we state the following substitution property of enumerations:

$$\mathsf{enum}_g(G_1^\mu[G^\mu/\mathbf{t}]) \subseteq \big(\mathsf{enum}_g(G_1^\mu)\big)[G^\mu/\mathbf{t}] \cup \mathsf{enum}_g(G^\mu)$$

From this property, it suffices to show that

$$\big(\mathsf{enum}_g(G_1^\mu)\big)[\mu\mathbf{t}.G_1^\mu/\mathbf{t}] \cup \mathsf{enum}_g(\mu\mathbf{t}.G_1^\mu) \subseteq \mathsf{enum}_g(\mu\mathbf{t}.G_1^\mu)$$

which holds by definition. We finish this proof by proving the substitution property just used, which we recall is

$$\mathsf{enum}_g(G_1^\mu[G^\mu/\mathbf{t}]) \subseteq \big(\mathsf{enum}_g(G_1^\mu)\big)[G^\mu/\mathbf{t}] \cup \mathsf{enum}_g(G^\mu)$$

The interesting proof case is recursion and we show it now.
Case: $G_1^\mu = \mu\mathbf{t}'.G_3^\mu$
By the Bergendragt convention, we may assume $\mathbf{t}' \notin \mathsf{FV}(G^\mu)$ and $\mathbf{t} \neq \mathbf{t}'$.
By induction hypothesis we have

$$\mathsf{enum}_g(G_3^\mu[G^\mu/\mathbf{t}]) \subseteq \big(\mathsf{enum}_g(G_3^\mu)\big)[G^\mu/\mathbf{t}] \cup \mathsf{enum}_g(G^\mu)$$

We must show

$$\mathsf{enum}_g(\mu\mathbf{t}'.(G_3^\mu[G^\mu/\mathbf{t}])) \subseteq \big(\mathsf{enum}_g(\mu\mathbf{t}'.G_3^\mu)\big)[G^\mu/\mathbf{t}] \cup \mathsf{enum}_g(G^\mu)$$

Which unfolds to

$$\big(\mathsf{enum}_g(G_3^\mu[G^\mu/\mathbf{t}])\big)[\mu\mathbf{t}'.(G_3^\mu[G^\mu/\mathbf{t}])/\mathbf{t}'] \subseteq \big(\mathsf{enum}_g(G_3^\mu)\big)[\mu\mathbf{t}'.G_3^\mu/\mathbf{t}'][G^\mu/\mathbf{t}] \cup \mathsf{enum}_g(G^\mu)$$

We use the induction hypothesis so that it suffices to show that

$$\big(\big(\mathsf{enum}_g(G_3^\mu)\big)[G^\mu/\mathbf{t}]\big)[\mu\mathbf{t}'.(G_3^\mu[G^\mu/\mathbf{t}])/\mathbf{t}'] \cup \mathsf{enum}_g(G^\mu)[\mu\mathbf{t}'.(G_3^\mu[G^\mu/\mathbf{t}])/\mathbf{t}'] \subseteq \tag{23}$$

$$\big(\mathsf{enum}_g(G_3^\mu)\big)[\mu\mathbf{t}'.G_3^\mu/\mathbf{t}'][G^\mu/\mathbf{t}] \cup \mathsf{enum}_g(G^\mu) \tag{24}$$

This holds because we can show two identities. The first identity is due to commutativity of successive substitutions (possible due to ($\mathbf{t}' \notin \mathsf{FV}(G^\mu)$ and $\mathbf{t} \neq \mathbf{t}'$)

$$\big(\big(\mathsf{enum}_g(G_3^\mu)\big)[G^\mu/\mathbf{t}]\big)[\mu\mathbf{t}'.(G_3^\mu[G^\mu/\mathbf{t}])/\mathbf{t}'] = \big(\mathsf{enum}_g(G_3^\mu)\big)[\mu\mathbf{t}'.G_3^\mu/\mathbf{t}'][G^\mu/\mathbf{t}] \tag{25}$$

The second identity is due to ($\mathbf{t}' \notin \mathsf{FV}(G^\mu)$

$$\mathsf{enum}_g(G^\mu)[\mu\mathbf{t}'.(G_3^\mu[G^\mu/\mathbf{t}])/\mathbf{t}'] = \mathsf{enum}_g(G^\mu) \tag{26}$$

□

Lemma 3 provides a termination argument for our decision procedure. This is because we have defined a measure that decreases in each recursive call. We can finally conclude that our decision procedure always terminates.

Lemma 4 (Termination of dec)**.** *Assuming* $(G^\mu, T^\mu) \notin V$*, and* $(G_1^\mu, T_1^\mu) \in \mathsf{next}_\mathsf{p}(G^\mu, T^\mu)$*, then* $M(V \cup \{(G^\mu, T^\mu)\}, G_1^\mu, T_1^\mu) < M(V, G^\mu, T^\mu)$*.*

Proof. The measure M is based on the cardinality of sets, if we can show the following strict inclusion we are done

$$\mathsf{enum}(G_1^\mu, T_1^\mu) \setminus \{(G^\mu, T^\mu)\} \cup V \subset \mathsf{enum}(G^\mu, T^\mu) \setminus V$$

By Lemma (3) we have $\mathsf{enum}(G_1^\mu, T_1^\mu) \subseteq \mathsf{enum}(G^\mu, T^\mu)$ so it suffices to show

$$\mathsf{enum}(G^\mu, T^\mu) \setminus \{(G^\mu, T^\mu)\} \cup V \subset \mathsf{enum}(G^\mu, T^\mu) \setminus V$$

Which follows as a consequence of the fact that an enumeration always contains the initial pair, that is, $(G^\mu, T^\mu) \in \mathsf{enum}(G^\mu, T^\mu)$. □

5 Soundness

We now prove the soundness theorem for the projection we introduced in the previous section. The theorem relates the global types for which the projection is defined, and the local types it produces, to the coinductive projection. The soundness theorem states that if $\mathsf{proj}_\mathsf{p}(G^\mu)$ is defined then the following must be derivable:

$$\exists G^\nu\ T^\nu.\ G^\mu\ \mathcal{R}\ G^\nu \wedge \mathsf{proj}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu \wedge G^\nu \downarrow^\nu_\mathsf{p}\ T^\nu \tag{27}$$

It is the case that $\mathsf{proj}_\mathsf{p}(G^\mu)$ is defined exactly when $\mathsf{projectable}_\mathsf{p}(G^\mu)$ holds. Thus, soundness reduces to the following statement:

$$\textit{if}\ \mathsf{projectable}_\mathsf{p}(G^\mu)\ \textit{then}\ \exists G^\nu\ T^\nu.\ G^\mu\ \mathcal{R}\ G^\nu \wedge \mathsf{trans}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu \wedge G^\nu \downarrow^\nu_\mathsf{p}\ T^\nu \tag{28}$$

This statement above says that when $\mathsf{projectable}_\mathsf{p}(G^\mu)$ holds, there exists coinductive types that G^μ and $\mathsf{proj}_\mathsf{p}(G^\mu)$ unravel to which are G^ν and T^ν respectively. which are related by coinductive projection $G^\nu \downarrow^\nu_\mathsf{p}\ T^\nu$. To prove this theorem, we will in this section show the following two equivalences:

$$\mathsf{projectable}_\mathsf{p}(G^\mu)\ \textit{if and only if}\ G^\mu \downarrow^\mu_\mathsf{p}\ \mathsf{trans}_\mathsf{p}(G^\mu) \tag{29}$$

$$G^\mu \downarrow^\mu_\mathsf{p}\ T^\mu\ \textit{if and only if}\ \exists G^\nu\ T^\nu.\ G^\mu\ \mathcal{R}\ G^\nu \wedge T^\mu\ \mathcal{R}\ T^\nu \wedge G^\nu \downarrow^\nu_\mathsf{p}\ T^\nu \tag{30}$$

Equation (29) states that the decision procedure $\mathsf{projectable}_\mathsf{p}(G^\mu)$ is specified by intermediate projection. Equation (30) states that intermediate projection and coinductive projection are equivalent. Connecting these two equivalences gives us the statement in Equation (28) and proves soundness.

The rest of this section introduces intermediate projection, proves Equations (29) and (30), and ends with the soundness theorem.

5.1 Intermediate Projection

The intermediate projection, denoted by $G^\mu \downarrow^\mu_\mathsf{p}\ T^\mu$, is formally defined by the rules presented in Figure 8. The rules restate those of the coinductive projection defined in Figure 6 on inductive types G^μ and T^μ with the necessary applications of $\mathsf{unfold}(\cdot)$ to hide μ-binders. For example, $[\mathsf{M1}\downarrow^\mu]$ restates rule $[\mathsf{M1}\downarrow^\nu]$ on inductive types as

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p} \xrightarrow{\mu} \mathsf{p}_1 : k\langle U\rangle.G_1^\mu \quad \mathsf{unfold}(T^\mu) = !^\mu k\langle U\rangle.T_1^\mu \quad G_1^\mu \downarrow^\mu_\mathsf{p}\ T_1^\mu}{G^\mu \downarrow^\mu_\mathsf{p}\ T^\mu}\ [\mathsf{M1}\downarrow^\mu]$$

The restatement of $[\mathsf{End}\downarrow^\nu]$ requires special care. Recall that the original rule is defined as

$$\frac{\neg\mathsf{partOf}^\nu_\mathsf{p}(G^\nu)}{G^\nu \downarrow^\nu_\mathsf{p}\ \mathsf{end}^\nu}\ [\mathsf{End}\downarrow^\nu]$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p} \xrightarrow{\mu} \mathsf{p}_1 : k\langle U\rangle.G_1^\mu \quad \mathsf{unfold}(T^\mu) = !^\mu k\langle U\rangle.T_1^\mu \quad G_1^\mu \downarrow_{\mathsf{p}}^\mu \; T_1^\mu}{G \downarrow_{\mathsf{p}}^\mu \; T} \; [\mathsf{M1}\downarrow^\mu]$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p} : k\langle U\rangle.G_1^\mu \quad \mathsf{unfold}(T^\mu) = ?^\mu k\langle U\rangle.T_1^\mu \quad G_1^\mu \downarrow_{\mathsf{p}}^\mu \; T_1^\mu}{G \downarrow_{\mathsf{p}}^\mu \; T} \; [\mathsf{M2}\downarrow^\mu]$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G_1^\mu \quad \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \quad \mathsf{guarded}_{\mathsf{p}}^\mu(G^\mu) \quad G_1^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu}{G^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu} \; [\mathsf{M}\downarrow^\mu]$$

$$\frac{\begin{array}{c}\forall j.\; G_j^\mu \downarrow_{\mathsf{p}}^\mu \; T_j^\mu \\ \mathsf{unfold}(G^\mu) = \mathsf{p} \xrightarrow{\mu} \mathsf{p}_1 : k\{l_j : G_j^\mu\}_{j\in J} \qquad \mathsf{unfold}(T^\mu) = k \oplus^\mu \{l_j : T_j^\mu\}_{j\in J}\end{array}}{G^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu} \; [\mathsf{B1}\downarrow^\mu]$$

$$\frac{\begin{array}{c}\forall j.\; G_j^\mu \downarrow_{\mathsf{p}}^\mu \; T_j^\mu \\ \mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p} : k\{l_j : G_j^\mu\}_{j\in J} \qquad \mathsf{unfold}(T^\mu) = k \;\&^\mu\; \{l_j : T_j^\mu\}_{j\in J}\end{array}}{G^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu} \; [\mathsf{B2}\downarrow^\mu]$$

$$\frac{\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\{l_j : G_j^\mu\}_{j\in J} \quad \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \quad \forall j.\; G_j^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu \wedge \mathsf{guarded}_{\mathsf{p}}^\mu(G_j^\mu)}{G^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu} \; [\mathsf{B}\downarrow^\mu]$$

$$\frac{\neg\mathsf{partOf}_{\mathsf{p}}^\mu(G^\mu) \quad G^\mu \; \mathcal{R} \; \mathsf{tocoind}(G^\mu)}{G^\mu \downarrow_{\mathsf{p}}^\mu \; \mathsf{end}^\mu} \; [\mathsf{End}\downarrow^\mu]$$

Fig. 8: *Intermediate projection* on inductive types, written as $G^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu$ 🐓.

If we naively restate this on inductive global types as

$$\frac{\neg\mathsf{partOf}_{\mathsf{p}}^\mu(G^\mu)}{G^\mu \downarrow_{\mathsf{p}}^\mu \; \mathsf{end}^\mu} \; [\mathsf{End}\downarrow^\mu]$$

then we are allowing non-contractive and open μ-types in derivations. That is, both $\mu\mathbf{t}.\mathbf{t} \downarrow_{\mathsf{p}}^\mu \; \mathsf{end}^\mu$ and $\mathbf{t} \downarrow_{\mathsf{p}}^\mu \; \mathsf{end}^\mu$ are derivable by $[\mathsf{End}\downarrow^\mu]$, but no coinductive type corresponds to $\mu\mathbf{t}.\mathbf{t}$ or $\mathbf{t}$. This suggests that $[\mathsf{End}\downarrow^\mu]$ requires a side condition. For it to behave as $[\mathsf{End}\downarrow^\nu]$, we could impose that G^μ is closed and contractive. This ensures that G^μ is well-formed and thus that there exists a corresponding G^ν. This is however not desirable. To see why, consider the purpose of restating $G^\nu \downarrow_{\mathsf{p}}^\nu \; T^\nu$ as $G^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu$. We wish to have the property that if $G^\mu \; \mathcal{R} \; G^\nu$ and $T^\mu \; \mathcal{R} \; T^\nu$ then

$$G^\mu \downarrow_{\mathsf{p}}^\mu \; T^\mu \text{ if and only if } G^\nu \downarrow_{\mathsf{p}}^\nu \; T^\nu$$

When showing the (if) direction for $[\mathsf{End}\lfloor^{\nu}]$, we must derive $\mathsf{closed}(G^{\mu})$ and $\mathsf{contr}(G^{\mu})$ from $G^{\mu}\ \mathcal{R}\ G^{\nu}$ by Proposition 2, which we recall shows the equivalence of well-formedness of G^{μ} and the existence of an unravelling. But since unravelling is an equivalent characterisation of well-formedness, we can use that as our side condition, eliminating the need for Proposition 2. Hence,

$$\frac{\neg\mathsf{partOf}^{\mu}_{\mathsf{p}}(G^{\mu}) \quad \exists G^{\nu}.G^{\mu}\ \mathcal{R}\ G^{\nu}}{G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ \mathsf{end}^{\mu}}\ [\mathsf{End}\lfloor^{\mu}]$$

Intermediate projection is decidable by the decision procedure $\mathsf{dec}_{\mathsf{p}}$.

Theorem 5. *Let G^{μ} and T^{μ} be inductive types and p a role. Then, $\mathsf{dec}_p(\{\}, G^{\mu}, T^{\mu}) = \mathsf{True}$ if and only if $G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu}$.*

Proof. For the (only if) case, we show that for any V, it suffices to show $\mathsf{dec}_p(V, G^{\mu}, T^{\mu}) = \mathsf{True}$ implies $(G^{\mu}, T^{\mu}) \in V \vee G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu}$. Proceed by functional induction on $\mathsf{dec}_p(V, G^{\mu}, T^{\mu})$. In the second case where V is non-empty, pick the right disjunct $G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu}$ and proceed by coinduction. For the (if) direction, we show the property for any visited list V, that is, $G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu}$ implies $\mathsf{dec}_p(V, G^{\mu}, T^{\mu}) = \mathsf{True}$. Proceed by functional induction on $\mathsf{dec}_p(V, G^{\mu}, T^{\mu})$. □

Corollary 6. *Let G^{μ} be an inductive type and p a role. Then, $\mathsf{projectable}_{\mathsf{p}}(G^{\mu})$ if and only if $G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ \mathsf{trans}_{\mathsf{p}}(G^{\mu})$.*

Proof. Follows directly from Theorem 5. □

Example 9. *Recall Examples 2, 4, and 5 which collectively use as an argument for why the running example should be projectable. Using the intermediate projection, a similar derivation can be done directly on the inductive types, making the unravelling to coinductive types unnecessary:*

$$G^{\mu} = \mu\mathbf{t}.\ \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : \mathbf{t}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ \mathbf{t}'\}$$
$$T^{\mu} = \mu\mathbf{t}.!^{\mu}k\langle U\rangle.\mathbf{t}$$

We derive

$$G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu} \tag{31}$$

After $[M1\lfloor^{\mu}]$ we must show that

$$\mu\mathbf{t}'.\ \mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : G^{\mu}, \quad l_2 : \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ G^{\mu}_1\}\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu} \tag{32}$$

We will by G^{μ}_1 refer to the global type in Equation (32). After $[B\lfloor^{\mu}]$, two premises must be shown

$$G^{\mu}\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu} \qquad \mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\ G^{\mu}_1\downarrow^{\mu}_{\mathsf{p}}\ T^{\mu} \tag{33}$$

The first premise is discharged by marking a cycle to Equation (31). For the second premise we apply $[M1\downarrow^{\mu}]$ and are left with showing

$$G_1^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu} \tag{34}$$

We mark a cycle to Equation (32). The full proof tree is

$$\dfrac{\dfrac{}{\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : G^{\mu}, \quad l_2 : G^{\mu}\} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}}\,[B\downarrow^{\mu}] \quad G^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}}{\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\{l_1 : G^{\mu}, \quad l_2 : G^{\mu}\} \downarrow_{\mathsf{p}}^{\mu} !^{\mu}k\langle U\rangle.\, T^{\nu}}\,[M1\downarrow^{\mu}] \tag{35}$$

The example above shows that some derivations of coinductive projections on coinductive types can equivalently be derived with intermediate projection on inductive types. We now prove the following equivalence between intermediate and coinductive projection.

Lemma 7. *Let G^{μ} and T^{μ} be inductive types, G^{ν} and T^{ν} coinductive types, and p a role. Then, if $G^{\mu} \mathcal{R} G^{\nu}$ and $T^{\mu} \mathcal{R} T^{\nu}$ then $G^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}$ if and only if $G^{\nu} \downarrow_{\mathsf{p}}^{\nu} T^{\nu}$.*

Proof. (*only-if*) Prove $G^{\nu} \downarrow_{\mathsf{p}}^{\nu} T^{\nu}$ by coinduction on $\downarrow_{\mathsf{p}}^{\nu}$ and derive from $G^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}$ that $\mathsf{guarded}_{\mathsf{p}}^{\mu}(G^{\mu}) \vee \mathsf{unfold}(T^{\mu}) = \mathsf{end}^{\mu}$ and proceed as in Lemma 15.
(*if*) Proof by coinduction on $\downarrow_{\mathsf{p}}^{\mu}$ and derive from $G^{\nu} \downarrow_{\mathsf{p}}^{\nu} T^{\nu}$ that $\mathsf{guarded}_{\mathsf{p}}^{\nu}(G^{\nu}) \vee T^{\nu} = \mathsf{end}^{\nu}$, and do case analysis on the disjunction as in Lemma 15, inverting $G^{\mu} \mathcal{R} G^{\nu}$ and $T^{\nu} \mathcal{R} T^{\mu}$ to derive the shape of $G^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}$. □

The lemma above is a weaker equivalence than Equation (30), and this is because it assumes the existence of the coinductive types G^{ν} and T^{ν} while Equation (30) asserts it. To prove the stronger equivalence we must show that $G^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}$is only derivable for inductive global and local types that can unravel to coinductive types.

Lemma 8. *Let G^{μ} and T^{μ} be inductive types and p a role. Then, if $G^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}$ then $G^{\mu} \mathcal{R} \mathsf{tocoind}(G^{\mu})$ and $T^{\mu} \mathcal{R} \mathsf{tocoind}(T^{\mu})$.*

With Lemma 7 and 8 we can now derive the stronger equivalence between intermediate projection and coinductive projection.

Theorem 9. *Let G^{μ} and T^{μ} be inductive types, G^{ν} and T^{ν} coinductive types, and p a role. Then, $G^{\mu} \downarrow_{\mathsf{p}}^{\mu} T^{\mu}$ if and only if $\exists G^{\nu}\, T^{\nu}.\, G^{\mu} \mathcal{R} G^{\nu} \wedge T^{\mu} \mathcal{R} T^{\nu} \wedge G^{\nu} \downarrow_{\mathsf{p}}^{\nu} T^{\nu}$*

Theorem 10 (Soundness)**.** *Let G^{μ} be an inductive type and p a role. Then, if $\mathsf{proj}_{\mathsf{p}}(G^{\mu})$ is defined then there exists coinductive types G^{ν} and T^{ν} such that $G^{\mu} \mathcal{R} G^{\nu}$, $\mathsf{proj}_{\mathsf{p}}(G^{\mu}) \mathcal{R} T^{\nu}$ and $G^{\nu} \downarrow_{\mathsf{p}}^{\nu} T^{\nu}$*

6 Completeness

We now prove the completeness theorem for the projection function in Section 4. This theorem states the coinductive condition under which the projection function must be

defined and specifies the property the resulting local type must satisfy More precisely, the theorem states that if $G^\nu \downarrow_{\mathsf{p}}^{\nu} T^\nu$ and $G^\mu \mathcal{R} G^\nu$ hold, then

$$\mathsf{proj}_{\mathsf{p}}(G^\mu) \textit{ is defined and } \mathsf{proj}_{\mathsf{p}}(G^\mu) \mathcal{R} T^\nu \tag{36}$$

Following this statement, the completeness of the projection function requires the satisfaction of two properties. Firstly, when $G^\nu \downarrow_{\mathsf{p}}^{\nu} T^\nu$ holds, the projection must be defined for any inductive type that unravels to G^ν. For example, consider our running example in Equation (5) which, as shown in Example 4, unravels to a coinductive global type. Completeness dictates that the projection must be defined for (5). Secondly, when $G^\nu \downarrow_{\mathsf{p}}^{\nu} T^\nu$ holds, $\mathsf{proj}_{\mathsf{p}}(G^\mu)$ must unravel to T^ν.

To prove completeness, we will first show the weaker property that if $G^\nu \downarrow_{\mathsf{p}}^{\nu} T^\nu$ and $G^\mu \mathcal{R} G^\nu$ hold, then

$$\mathsf{trans}_{\mathsf{p}}(G^\mu) \mathcal{R} T^\nu \tag{37}$$

To then show completeness, as stated in (36), it remains to show that $\mathsf{proj}_{\mathsf{p}}(G^\mu)$ is defined. This follows from Theorem 9 with $G^\nu \downarrow_{\mathsf{p}}^{\nu} T^\nu$, from which $G^\mu \downarrow_{\mathsf{p}}^{\mu} \mathsf{trans}_{\mathsf{p}}(G^\mu)$ can be derived, and then by Corollary 6 we have $\mathsf{projectable}_{\mathsf{p}}(G^\mu)$.

The rest of this section proves properties about $\mathsf{trans}_{\mathsf{p}}(G^\mu)$ from which we will derive (37) and then (36).

6.1 Properties about Translation and Unfolding

The translation function satisfies the following properties about closedness and contractiveness.

Lemma 11. *Let G^μ be an inductive global type and p a role. Then,*

1. *If $\mathsf{closed}(G^\mu)$ then $\mathsf{closed}(\mathsf{trans}_{\mathsf{p}}(G^\mu))$*
2. *$\mathsf{contr}(\mathsf{trans}_{\mathsf{p}}(G^\mu))$*

The lemma states that closedness is preserved by translation and that translation only produces contractive local types. With this lemma and Proposition 2 we can from $G^\mu \mathcal{R} G^\nu$ derive the following property:

$$\mathsf{trans}_{\mathsf{p}}(G^\mu) \mathcal{R} \mathsf{tocoind}(\mathsf{trans}_{\mathsf{p}}(G^\mu)) \tag{38}$$

In order to prove Equation (37), and therefore completeness, it suffices to show from $G^\nu \downarrow_{\mathsf{p}}^{\nu} T^\nu$ that the following equality holds:

$$\mathsf{tocoind}(\mathsf{trans}_{\mathsf{p}}(G^\mu)) = T^\nu \tag{39}$$

Unravelling is defined in terms of unfolding and, in order to prove (39), we need to prove some properties about substitution and unfolding.

Lemma 12. *The following properties hold for global μ-types. Similar properties hold for local μ-types but have been elided.*

$$
\begin{array}{lll}
(1) & \mathsf{unfold}(\mu \mathbf{t}.G^\mu) = \mathsf{unfold}(G^\mu[\mu \mathbf{t}.G^\mu/\mathbf{t}]) & \\
(2) & \mathsf{unfold}(\mathsf{unfold}(G^\mu)) = \mathsf{unfold}(G^\mu) & \\
(3) & \mathsf{unfold}(G^\mu[G_1^\mu/\mathbf{t}]) = \mathsf{unfold}(G^\mu)[G_1^\mu/\mathbf{t}] & \text{if } |G_1^\mu| = 0 \\
(4) & \mathsf{unfold}(G^\mu) = \mathbf{t} & \text{if } \mathsf{guardedVar}(\mathbf{t}, G^\mu) = \mathsf{False}
\end{array}
$$

The four properties above regard the unfolding of μ-types. Property (1) says that the number of unfoldings performed by unfold is sufficient and that more unfoldings yields the same result. As a result, property (2) states that unfolding is idempotent. The third case, property (3), says that substitution commutes with unfolding when the replacing type has no top-level μ-binders. Finally, property (4) states that if $\mathbf{t}$ is not guarded in a μ-type, then it will be exposed by unfolding.

We now prove properties about how translation interacts with substitution and unfolding from which we derive Lemma 15 and show (39).

Lemma 13 (Distributivity of translation over substitution)**.** *Let G^μ and G_1^μ be inductive global types and p a role. Then,* $\mathsf{trans}_\mathsf{p}\big(G^\mu[G_1^\mu/\mathbf{t}]\big) = \mathsf{trans}_\mathsf{p}(G^\mu)[\mathsf{trans}_\mathsf{p}(G_1^\mu)/\mathbf{t}]$.

Proof. The proof proceeds by induction on G^μ. The most interesting case is when $G^\mu = \mu \mathbf{t}'.G_2^\mu$, where we must show that

$$\mathsf{trans}_\mathsf{p}(\mu \mathbf{t}'.G_2^\mu[G_1^\mu/\mathbf{t}]) = \mathsf{trans}_\mathsf{p}(\mu \mathbf{t}'.G_2^\mu)[\mathsf{trans}_\mathsf{p}(G_1^\mu)/\mathbf{t}]$$

In this case, the induction hypothesis is

$$\mathsf{trans}_\mathsf{p}\big(G_2^\mu[G_1^\mu/\mathbf{t}]\big) = \mathsf{trans}_\mathsf{p}(G_2^\mu)[\mathsf{trans}_\mathsf{p}(G_1^\mu)/\mathbf{t}] \tag{40}$$

At this point, the proof proceeds by cases. We only address the case in which the following holds:

$$\mathsf{guardedVar}(\mathbf{t}', \mathsf{trans}_\mathsf{p}(G_2^\mu)) = \mathsf{True} \tag{41}$$

$$
\begin{aligned}
\mathsf{trans}_\mathsf{p}(\mu \mathbf{t}'.G_2^\mu[G_1^\mu/\mathbf{t}]) &= \mu \mathbf{t}'.\big(\mathsf{trans}_\mathsf{p}(G_2^\mu[G_1^\mu/\mathbf{t}])\big) \\
&= \mu \mathbf{t}'.\big(\mathsf{trans}_\mathsf{p}(G_2^\mu)[\mathsf{trans}_\mathsf{p}(G_1^\mu)/\mathbf{t}]\big) && \text{By (40)} \\
&= \mathsf{trans}_\mathsf{p}(\mu \mathbf{t}'.G_2^\mu)[\mathsf{trans}_\mathsf{p}(G_1^\mu)/\mathbf{t}] && \text{By (41)}
\end{aligned}
$$

The last identity requires the following fact which we have not yet derived

$$\mathsf{guardedVar}(\mathbf{t}', \mathsf{trans}_\mathsf{p}(G_2^\mu[G_1^\mu/\mathbf{t}])) = \mathsf{True}$$

which due to the induction hypothesis reduces to

$$\mathsf{guardedVar}(\mathbf{t}', \mathsf{trans}_\mathsf{p}(G_2^\mu)[\mathsf{trans}_\mathsf{p}(G_1^\mu)/\mathbf{t}]) = \mathsf{True}$$

In order to prove this, we derive and use the following property:

$$\text{If } \mathbf{t}' \notin \mathsf{FV}(T_1^\mu) \text{ and } \mathsf{guardedVar}(\mathbf{t}', T^\mu) = \mathsf{True} \text{ then } \mathsf{guardedVar}(\mathbf{t}', T^\mu[T_1^\mu/\mathbf{t}']) = \mathsf{True}$$

After which, we must show that

$$\mathbf{t}' \notin \mathsf{FV}(\mathsf{trans}_\mathsf{p}(G_1^\mu)) \qquad \mathsf{guardedVar}(\mathbf{t}', \mathsf{trans}_\mathsf{p}(G_2^\mu)) = \mathsf{True}$$

The first premise holds because of the Barendregt convention, which ensures that $\mathbf{t}'$ is fresh for G_1^μ. The second fact holds by assumption. □

From distributivity of translation over substitution we may think that the identity

$$\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{trans}_\mathsf{p}(\mathsf{unfold}(G^\mu))$$

holds. Unfortunately, this is not the case since unfolding of a global type stops at the first appearance of a constructor that is not a μ-binder, e.g., $\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu$. Translation might erase such constructs, triggering further unfolding in G^μ.

Example 10 (Unfolding and translation)**.** *The following global type is a counterexample that disproves* $\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{trans}_\mathsf{p}(\mathsf{unfold}(G^\mu))$

$$\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\mu\mathbf{t}.\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U\rangle.\mathbf{t}$$

Unfolding after translation yields

$$\mathsf{unfold}(\mathsf{trans}_\mathsf{r}(\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\mu\mathbf{t}.\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U\rangle.\mathbf{t})) = !^\mu k'\langle U\rangle.\mu\mathbf{t}.!^\mu k'\langle U\rangle.\mathbf{t}$$

Translating after unfolding yields

$$\mathsf{trans}_\mathsf{r}(\mathsf{unfold}(\mathsf{p} \xrightarrow{\mu} \mathsf{q} : k\langle U\rangle.\mu\mathbf{t}.\mathsf{r} \xrightarrow{\mu} \mathsf{s} : k'\langle U\rangle.\mathbf{t})) = \mu\mathbf{t}.!^\mu k'\langle U\rangle.\mathbf{t} \tag{42}$$

The produced local types are not the same

$$!^\mu k'\langle U\rangle.\mu\mathbf{t}.!^\mu k'\langle U\rangle.\mathbf{t} \neq \mu\mathbf{t}.!^\mu k'\langle U\rangle.\mathbf{t}$$

The problem in Example 10 is that we need to apply a final unfolding to the local type in Equation (42). This informs us that if we unfold *after* applying translation, it is equivalent to unfolding both *before* and *after* translation, which leads to the following:

Lemma 14 (Unfolding translations)**.** *Let G^μ be an inductive global type and p a role. Then,* $\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}(G^\mu)))$.

Proof. The $\mathsf{unfold}(G^\mu)$ function is defined as $\mathsf{unfold}_1^{|G^\mu|}(G^\mu)$. To prove the theorem, it suffices to show the more general property for some natural number n.

$$\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}_1^n(G^\mu)))$$

We prove this by induction on n, where the case of $n = 0$ is immediate because $\mathsf{unfold}_1^0(G^\mu) = G^\mu$. For the case of $n = n' + 1$, we have the following induction hypothesis:

$$\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}_1^{n'}(G^\mu)))$$

And, we must show

$$\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}_1^{n'}(\mathsf{unfold}_1(G^\mu))))$$

Because of the induction hypothesis, the proof reduces to showing

$$\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}_1(G^\mu)))$$

The proof is by induction on G^μ, the interesting case being $G^\mu = \mu\mathbf{t}.G_1^\mu$, where we must show that

$$\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mu\mathbf{t}.G_1^\mu)) = \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}_1(\mu\mathbf{t}.G_1^\mu)))$$

We proceed by cases.
Case: $\mathsf{guardedVar}(\mathbf{t}, G_1^\mu) = \mathsf{True}$.

$$\begin{aligned}
\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mu\mathbf{t}.G_1^\mu)) &= \mathsf{unfold}(\mu\mathbf{t}.(\mathsf{trans}_\mathsf{p}(G_1^\mu))) \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G_1^\mu)[\mathsf{trans}_\mathsf{p}(\mu\mathbf{t}.G_1^\mu)/\mathbf{t}]) && \text{Lemma 12(1)} \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G_1^\mu[\mu\mathbf{t}.G_1^\mu/\mathbf{t}])) && \text{Lemma 13} \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}_1(G^\mu)))
\end{aligned}$$

Case: $\mathsf{guardedVar}(\mathbf{t}, G_1^\mu) = \mathsf{False}$.

$$\begin{aligned}
\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mu\mathbf{t}.G^\mu)) &= \mathsf{unfold}(\mathsf{end}^\mu) \\
&= \mathsf{end}^\mu \\
&= \mathbf{t}[\mathsf{end}^\mu/\mathbf{t}] \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G_1^\mu))[\mathsf{end}^\mu/\mathbf{t}] && \text{Lemma 12(4)} \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G_1^\mu)[\mathsf{end}^\mu/\mathbf{t}]) && \text{Lemma 12(3)} \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G_1^\mu)[\mathsf{trans}_\mathsf{p}(\mu\mathbf{t}.G_1^\mu)/\mathbf{t}]) \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G_1^\mu[\mu\mathbf{t}.G_1^\mu/\mathbf{t}])) && \text{Lemma 13} \\
&= \mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}_1(G^\mu)))
\end{aligned}$$

□

Lemma 15 (Translation as projection)**.** *Let G^μ be an inductive type, G^ν and T^ν coinductive types, and p a role. Then, if $G^\mu \mathrel{\mathcal{R}} G^\nu$ and $G^\nu \downarrow_\mathsf{p}^\nu T^\nu$ then $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = T^\nu$.*

Proof. By coinduction using the candidate relation $\{(\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)), T^\nu) \mid G^\nu \downarrow^\nu_\mathsf{p} T^\nu \wedge G^\mu \mathcal{R} G^\nu\}$. From $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$, derive $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu) \vee T^\nu = \mathsf{end}^\nu$ and proceed by cases. We sketch the two cases.
Case: $T^\nu = \mathsf{end}^\nu$.
Derive $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = \mathsf{end}^\nu$ from the facts

$$G^\nu \downarrow^\nu_\mathsf{p} \mathsf{end}^\nu \qquad G^\mu \mathcal{R} G^\nu \qquad \neg\mathsf{partOf}^\nu_\mathsf{p}(G^\mu)$$

Case: $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu)$.
Proceed by induction on $\mathsf{guarded}^\nu_\mathsf{p}(G^\nu)$. From the distinctive shape of G^ν in each of the cases, we invert $G^\mu \mathcal{R} G^\nu$ and learn the shape of $\mathsf{unfold}(G^\mu)$. For example if $G^\nu = \mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu_1$, we derive the following two facts:

$$\mathsf{unfold}(G^\mu) = \mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu_1 \qquad G^\mu_1 \mathcal{R} G^\nu_1$$

Due to idempotence of $\mathsf{unfold}(\cdot)$, it is the case that $\mathsf{tocoind}(\cdot)$ has the following property:

$$\mathsf{tocoind}(T^\mu) = \mathsf{tocoind}(\mathsf{unfold}(T^\mu))$$

Using this property, we proceed as

$$\begin{aligned}
\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) &= \mathsf{tocoind}(\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(G^\mu))) && (43)\\
&= \mathsf{tocoind}(\mathsf{unfold}(\mathsf{trans}_\mathsf{p}(\mathsf{unfold}(G^\mu)))) \quad \text{Lemma 14} && (44)\\
&= \mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu) && (45)
\end{aligned}$$

We proceed by inversion on $\mathsf{p}_1 \xrightarrow{\nu} \mathsf{p}_2 : k\langle U\rangle.G^\nu \downarrow^\nu_\mathsf{p} T^\nu$. This allows one to further simplify Equation (45). For example, if the last rule application was $[\mathsf{M1}\downarrow^\nu]$, then we know $\mathsf{p} = \mathsf{p}_1$ and $T^\nu =!^\nu k\langle U\rangle.T^\nu_1$ and can derive

$$\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(\mathsf{p}_1 \xrightarrow{\mu} \mathsf{p}_2 : k\langle U\rangle.G^\mu_1) =!^\nu k\langle U\rangle.\big(\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu_1))\big) \qquad (46)$$

In which case, it suffices to show

$$\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu_1)) = T^\nu_1$$

This follows by coinduction hypothesis. □

Theorem 16 (Completeness)**.** *Let G^μ be an inductive type, G^ν and T^ν coinductive types, and p a role. Then, if $G^\nu \downarrow^\nu_\mathsf{p} T^\nu$ and $G^\mu \mathcal{R} G^\nu$ then $\mathsf{proj}_\mathsf{p}(G^\mu)$ is defined and $\mathsf{proj}_\mathsf{p}(G^\mu) \mathcal{R} T^\nu$.*

Proof. From $G^\mu \mathcal{R} G^\nu$ we know, by Proposition 2, that G^μ is closed. This along with Lemma 11 shows that $\mathsf{trans}_\mathsf{p}(G^\mu)$ is closed and contractive. From Proposition 2, we thus have

$$\mathsf{trans}_\mathsf{p}(G^\mu) \mathcal{R} \mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu))$$

From Lemma 15 we have that $\mathsf{tocoind}(\mathsf{trans}_\mathsf{p}(G^\mu)) = T^\nu$ and thus

$$\mathsf{trans}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu$$

From Theorem 9 with $G^\mu\ \mathcal{R}\ G^\nu$, $\mathsf{trans}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu$ and $G^\nu \downarrow^\nu_\mathsf{p}\ T^\nu$ we have

$$G^\mu \downarrow^\mu_\mathsf{p}\ \mathsf{trans}_\mathsf{p}(G^\mu)$$

From Corollary 6 we have $\mathsf{projectable}_\mathsf{p}(G^\mu)$ and thus $\mathsf{trans}_\mathsf{p}(G^\mu) = \mathsf{proj}_\mathsf{p}(G^\mu)$ which lets us conclude

$$\mathsf{proj}_\mathsf{p}(G^\mu)\ \mathcal{R}\ T^\nu$$

□

7 Mechanisation

All of our results are mechanised in Coq [18] using SSReflect [19] for writing proofs, the Paco library [20] for defining coinductive predicates, the Equations package [21] for defining functions by well-founded recursion (such as $\mathsf{dec}_p(V, G^\mu, T^\mu)$), and Autosubst2 [22] to generate syntax of inductive global and local types with binders represented by de Bruijn indices [23].

The mechanisation uses coinductive extensional equivalence relations to equate coinductive terms. For presentation purposes, we use propositional equality to equate coinductive types. These two types of equality are consistent [24].

In this section, we cover how to create predicates and relations that are defined using both inductive and coinductive inference rules, like our unravelling relation from Figure 7. We discuss how to create an inversion principle that allows us to do case analysis on the property of G^μ that $G^\mu\ \mathcal{R}\ \mathsf{tocoind}(G^\mu)$. Finally, we show how we prove decidability of this property.

7.1 Custom inversion principles

Many proofs on inductive global types work up to unfolding. Unravelling, for instance, unravels a finite number of μ-binders at every step and our intermediate projection function $\downarrow^\mu_\mathsf{p}$ and `sat` procedure both work in a similar way. To abstract away from finite unfoldings we use the following `gInvPred` predicate 🐓.

```
Variant gInvPredF (P : gType -> Prop) : gType -> Prop :=
| HTM g a u : P g          -> gInvPredF P (GMsg a u g)
| HTB gs d  : Forall P es -> gInvPredF P (GBranch d gs)
| HTE       :                gInvPredF P GEnd
Definition unf g := (iter (mu_height g) unf1 g).
Variant UnfoldF (P : gType -> Prop) : gType -> Prop :=
 | UnfF1 g :  P (unf g) -> UnfoldF g.
Definition gInvPred : (gType -> Prop) := paco1 (UnfoldF \o gInvPredF) bot.
                                                    (*function composition*)
```

We define two generating functions `InvPredF` and `UnfoldF` and generate `InvPred` as the greatest fixed point of their composition. The function `unf` corresponds to `unfold`

from Section 2. `InvPredF` contains cases for all constructors of inductive global types except for $\mu\mathbf{t}$ and $\mathbf{t}$. `UnfoldF` unfolds the top-level μ-binders from a global type. The key insight is that $\mathsf{InvPred}(\mathsf{G}^\mu)$ is equivalent to asserting closedness and contractiveness of G^μ.

The inversion principle of `InvPred` is convenient for proving predicates P that are closed under unfolding of inductive global types, i.e. $\forall G.\ P\ \mu\mathbf{t}.G$ if and only if $P\ G[\mu\mathbf{t}.G]$, as any unfolding applied by inverting `UnfoldF` can similarly be applied in the goal.

7.2 Well-founded recursion

Proposition 2 shows that well formedness of G^μ types can be characterised in terms of unravelling by asserting that $G^\mu\ \mathcal{R}\ \mathsf{tocoind}(G^\mu)$ is derivable. Due to the equivalence this propositino states, $G^\mu\ \mathcal{R}\ \mathsf{tocoind}(G^\mu)$ is trivially decidable by checking that $\mathsf{closed}(G^\mu)$ and $\mathsf{contr}(G^\mu)$ holds.

An alternative way of deciding $G^\mu\ \mathcal{R}\ \mathsf{tocoind}(G^\mu)$ which is strikingly similar to how our decision procedure $\mathsf{projectable}_\mathsf{p}(G^\mu)$ is defined, is `gUnravelsb` ↯ which is defined by well-founded recursion. We study this procedure as it provides a simpler setting for presenting our approach to defining well-founded recursion on μ-types with enumerations in Coq.

For convenience we define unravelling in two equivatent ways as `gUnravel` ↯ and `gUnravel2` ↯, and the latter definition corresponds to our definition in Figure 7. We represent $G^\mu\ \mathcal{R}\ \mathsf{tocoind}(G^\mu)$ in Coq as `\gUnravel2 G (gtocoind g)` and due to the theorem below ↯ it suffices to decide `gInvPred` which we introduced in Section 7.1.

```
Lemma gInvPred_iff : forall g, gInvPred g <-> gUnravel2 g (gtocoind g).
```

We decide `gInvPred` by defining a predicate `invP` that checks a global type is neither a binder nor a variable. This predicate is repeatedly applied by `sat` on continuations of the initial global type until a global type previously seen appears.

```
Definition invP g :=
match unf g with | GRec _ | GVar _ => false | _ => true end.

Definition invpred g := sat nil invP g.
Theorem gInvPred_dec : forall g, InvPred g <-> invpred g = true
```

We use the Equations package to define `sat` ↯ by well-founded recursion on the decreasing measure `gmeasure g V` ↯ which is defined as the number of distinct global types in the enumeration of `g` after removing the global types in `V`.

```
Definition gmeasure (g : gType) (V : seq gType) :=
  size (rep_rem V (undup (enumg g))).
Lemma closed_enum  : forall g0 g1 g2, g1 \in nextg (full_unf g) ->
                                      g2 \in enumg g1  -> g2 \in enumg g.
Equations sat (V : seq  gType)  (P : gType -> bool)
                    (g : gType) : bool by wf (gmeasure g V) :=
 sat V P g with (dec (g \in V)) => {
  sat  _ _ _ in_left  := true;
  sat V P g  in_right :=  (P g) &&
```

```
10                    (foldInMap (nextg (full_unf g))
11                               (fun g' _ => sat (g::V) P g')) }.
```

Defining `sat` generates one obligation that must be proved to show termination. If we write `gmeasure g V` as $M(g, V)$, then we must show it is decreasing for arguments to the recursive call, i.e. that $M(g', \{g\} \cup V) < M(g, V)$

Using a variant of the familiar `map` on inductive lists called `foldInMap` our obligation is enriched with the assumption that `g' \in nextg (full_unf g)`. The boolean wrapper `dec` further enriches the obligation with the case of the if-statement, `g \notin V`.

Termination is then ensured due to `closed_enum` (l. 3) . The lemma states that the enumerations of a global types continuations, will all be part of the initial global types enumeration. The proof of this lemma is short, less than 100 lines.

The full termination proof for `sat` is short (about 250 lines) and the approach is general. The mechanisation also proves termination of the decision procedure for $\mathsf{projectable}_{\mathsf{p}}(G^{\mu})$. This task only requires adapting the algorithm to pairs of terms. This termination proof is also short. The conciseness is due to the space of continuations being computed by structural recursion by enum_g and enum_l. This makes it straightforward to prove substitution properties about it by induction on syntax.

8 Related Work and Discussion

Related Work. Ghilezan et al. [7] are the first to introduce coinductive projection on coinductive global and local types. They use it to show soundness and completeness of synchronous multiparty session subtyping. A key difference is that whereas we represent the infinite unfolding of a μ-type as a coinductive type, they represent it as a partial function. Projection on μ-types is then defined indirectly in terms of the coinductive projection of their corresponding partial functions. Because of this indirect definition, their projection is not computable. Our intermediate projection $\downarrow^{\mu}$ is similar to their projection on μ-types. However, ours is defined with inference rules stated directly on the μ-types which is why we can decide membership and thus compute projection. Castro-Perez et al. [12] use coinductive projection to express their meta theory about multiparty session types. Their main result is trace equivalence between processes, coinductive local types and coinductive global types, which they mechanise in Coq. Like us, they show soundness of their projection on μ-types. Their projection is however not complete, which is what inspired us to investigate approaches to sound and complete projection. A consequence of their projection on μ-types not being complete, is that there are many inductive global types that have the trace equivalence property, but must be excluded since their projection is undefined. Jacob et al. [17] show deadlock and leak freedom of multiparty GV, an extension of the functional language GV [25, 26]. They use coinductive projection to define when local types are compatible and do not define a projection on μ-types. Other work has formalised the notion of projection in Coq. Cruz-Filipe et al. [27, 28] formalise syntax and semantics of tail-recursive choreographies and a projection that includes full merge. However, this work does not approach coinductive syntax and therefore does not show any soundness and completeness results.

Our algorithm from Section 5 implements a procedure proposed by Eikelder [6]. This work provides several algorithms for deciding recursive type equivalence. Also, our proof of termination is quite similar to theirs. However, they define the set of reachable states as set comprehension, whereas we constructively produce a list. Similarly, showing their set comprehension is finite, boils down to substitution lemmas. Unlike ours, their work has not been mechanised in a proof assistant/theorem prover. The idea of defining the space of continuations for global and local type as an explicit enumeration is inspired by Asperti [29] who mechanised a concise proof of regular expression equivalence in the Matita theorem prover [30]. They do this by a new construction called pointed regular expressions. Essentially, this adds marks to a regular expression, such that one can encode state transitions by moving marks. This makes computing reachable configurations as trivial as computing all markings.

The primary focus of this work is on global types. Scalas and Yoshida [31] propose a more general approach that shows that properties such as deadlock freedom can be derived directly on local types without the need for global types and the corresponding projection. However, their approach misses the main advantage provided by global types which is providing a specification (blueprint) of the used protocols.

Discussion and Future work. This work is part of the MECHANIST project that aims at mechanising the full theory of multiparty asynchronous session types [8]. Our next step is to mechanise a proof of semantic equivalence between global types and their projections to local types through proj_p. Semantic equivalence is a property similar to trace equivalence which Castro-Perez et al. [12] mechanised. However, there are some key differences in our objectives. Their main result is Zooid, a tool that extracts certified message-passing programs, which is why their process syntax differs significantly from the original syntax by Honda et al (e.g., no parallel composition). Instead, we aim at mechanising the exact process calculus presented by Honda et al.. As the meta theory in Castro-Perez et al. [12] is independent of their projection function, it would also be interesting future work to adapt proj_p to their setting. Finally, proj_p implements the restrictive plain merge but related work also uses full merge [7, 28]. It would be interesting to define a binder-agnostic projection using full merge.

9 Conclusions

Projection is a function that maps global types to local types. The projections found in the literature impose syntactic restrictions that make them incomplete with respect to coinductive projection. This work shows the existence of a decidable projection that is sound and complete. Our procedure works in two phases: first a decision procedure tests a soundness property and, if successful, a second procedure translates the global type to a local type. The latter is very similar to the existing projections in the literature. The novelty of our work is in the decision procedure. All results have been mechanised in Coq.

References

[1] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline

for structured communication-based programming. In: Proceedings of ESOP. LNCS, vol. 1381, pp. 122–138. Springer, ??? (1998). https://doi.org/10.1007/BFb0053567

[2] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of POPL, pp. 273–284. ACM, ??? (2008). https://doi.org/10.1145/1328438.1328472

[3] Session types in programming languages: A collection of implementations. http://www.simonjf.com/2016/05/28/session-type-implementations.html. Accessed: May 2023

[4] Pierce, B.C.: Types and Programming Languages. MIT Press, ??? (2002)

[5] Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In: Fernández, M., Kirchner, C. (eds.) Proceedings of the First International Workshop on Security and Rewriting Techniques, SecReT@ICALP 2006, Venice, Italy, July 15, 2006. Electronic Notes in Theoretical Computer Science, vol. 171, pp. 73–93. Elsevier, ??? (2006). https://doi.org/10.1016/J.ENTCS.2007.02.056 . https://doi.org/10.1016/j.entcs.2007.02.056

[6] Eikelder, H.: Some algorithms to decide the equivalence of recursive types. https://pure.tue.nl/ws/files/2150345/9211264.pdf. Accessed: May 2023 (1991)

[7] Ghilezan, S., Jaksic, S., Pantovic, J., Scalas, A., Yoshida, N.: Precise subtyping for synchronous multiparty sessions. Journal of Logical and Algebraic Methods in Programming **104**, 127–173 (2019) https://doi.org/10.1016/j.jlamp.2018.12.002

[8] Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. Journal of the ACM **63**(1), 9–1967 (2016) https://doi.org/10.1145/2827695

[9] The Coq development team: The Coq Proof Assistant. https://coq.inria.fr. Accessed: May 2023

[10] Tirore, D.L., Bengtson, J., Carbone, M.: A sound and complete projection for global types. In: Naumowicz, A., Thiemann, R. (eds.) 14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland. LIPIcs, vol. 268, pp. 28–12819. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, ??? (2023). https://doi.org/10.4230/LIPICS.ITP.2023.28 . https://doi.org/10.4230/LIPIcs.ITP.2023.28

[11] Barendregt, H.P.: The Lambda Calculus - Its Syntax and Semantics. Studies in logic and the foundations of mathematics, vol. 103. North-Holland, ??? (1985)

[12] Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty

processes. In: Proceedings of PLDI, pp. 237–251. ACM, ??? (2021). https://doi.org/10.1145/3453483.3454041

[13] Demangeon, R., Yoshida, N.: On the expressiveness of multiparty sessions. In: Proceedings of FSTTCS. LIPIcs, vol. 45, pp. 560–574 (2015). https://doi.org/10.4230/LIPIcs.FSTTCS.2015.560

[14] Bejleri, A., Yoshida, N.: Synchronous multiparty session types. In: Proceedings of PLACES. ENTCS, vol. 241, pp. 3–33. Elsevier, ??? (2008). https://doi.org/10.1016/j.entcs.2009.06.002

[15] Glabbeek, R., Höfner, P., Horne, R.: Assuming just enough fairness to make session types complete for lock-freedom. In: Proceedings of LICS, pp. 1–13. IEEE, ??? (2021). https://doi.org/10.1109/LICS52264.2021.9470531

[16] Jeannin, J.-B., Kozen, D., Silva, A.: Cocaml: Functional programming with regular coinductive types. Fundamenta Informaticae **150**, 347–377 (2017) https://doi.org/10.3233/FI-2017-1473

[17] Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: functional multiparty session types with certified deadlock freedom. Proceedings of the ACM on Programming Languages **6**(ICFP), 466–495 (2022) https://doi.org/10.1145/3547638

[18] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, ??? (2004). https://doi.org/10.1007/978-3-662-07964-5

[19] Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system (2016). https://inria.hal.science/inria-00258384

[20] Hur, C., Neis, G., Dreyer, D., Vafeiadis, V.: The power of parameterization in coinductive proof. In: Proceedings of POPL, pp. 193–206. ACM, ??? (2013). https://doi.org/10.1145/2429069.2429093

[21] Sozeau, M., Mangin, C.: Equations reloaded: High-level dependently-typed functional programming and proving in coq. Proceedings of the ACM on Programming Languages **3**(ICFP), 86–18629 (2019) https://doi.org/10.1145/3341690

[22] Stark, K., Schäfer, S., Kaiser, J.: Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In: Proceedings of CPP, pp. 166–180. ACM, ??? (2019). https://doi.org/10.1145/3293880.3294101

[23] de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae (Proceedings) **75**(5), 381–392 (1972) https://doi.org/10.1016/1385-7258(72)90034-0

[24] Coinductive types and corecursive functions. https://coq.inria.fr/refman/language/core/coinductive.html. Accessed: May 2023

[25] Gay, S., Vasconcelos, V.: Linear type theory for asynchronous session types. Journal of Functional Programming **20**(1), 19–50 (2010) https://doi.org/10.1017/S0956796809990268

[26] Wadler, P.: Propositions as sessions. In: Thiemann, P., Findler, R.B. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012, pp. 273–286. ACM, ??? (2012). https://doi.org/10.1145/2364527.2364568 . https://doi.org/10.1145/2364527.2364568

[27] Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in coq. In: Cohen, L., Kaliszyk, C. (eds.) Proceedings of ITP 2021. LIPIcs, vol. 193, pp. 15–11518. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, ??? (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.15 . https://doi.org/10.4230/LIPIcs.ITP.2021.15

[28] Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: Cerone, A., Ölveczky, P.C. (eds.) Proceedings of ICTAC 2021. Lecture Notes in Computer Science, vol. 12819, pp. 115–133. Springer, ??? (2021). https://doi.org/10.1007/978-3-030-85315-0_8 . https://doi.org/10.1007/978-3-030-85315-0_8

[29] Asperti, A.: A compact proof of decidability for regular expression equivalence. In: Proceedings of ITP. LNCS, vol. 7406, pp. 283–298. Springer, ??? (2012). https://doi.org/10.1007/978-3-642-32347-8_19

[30] Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: The Matita interactive theorem prover. In: Proceedings of CADE. LNCS, vol. 6803, pp. 64–69. Springer, ??? (2011). https://doi.org/10.1007/978-3-642-22438-6_7

[31] Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30–13029 (2019) https://doi.org/10.1145/3290343

Appendix C

Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction

Multiparty Asynchronous Session Types: A Mechanised Proof of Subject Reduction

Abstract. Session types is a typing discipline for statically ensuring that the communication behaviour of the components in a concurrent system conforms to a specification given as a session type. Session types were initially introduced in a binary setting, specifying communication patterns between two components. With the advent of multiparty session types (MPST), the typing discipline was extended to arbitrarily many components. In MPST, communication patterns are given in terms of global types, an Alice-Bob notation that gives a global view of how components interact. A central theorem of MPST is subject reduction: a well-typed system remains well-typed after reduction. The literature contains formulations of MPST with proofs of subject reduction that have later been shown to be incorrect. In this paper, we show that the subject reduction proof of the original formulation of MPST by Honda et al. contains some flaws. Additionally, we provide a restriction to the theory and show that, for this fragment, subject reduction does indeed hold. All of our proofs are mechanised using the Coq proof assistant.

Keywords: Multiparty Session Types · Mechanisation · Coq.

1 Introduction

Session types provide a type-based framework for specifying how participants in a concurrent communication-based system exchange messages. These communication patterns, specified in a protocol language, are used for verifying the concurrent programs that implement them. The framework guarantees properties such as type safety (well-typed programs never go wrong), session fidelity (processes behave according to protocol descriptions), and in-session deadlock-freedom (protocols never get stuck). Originally, Honda et al. [26, 27] proposed *binary session types* where types (protocol specifications) describe only how pairs of processes exchange messages. Inspired by choreographic languages [8], Honda et al. later extended this concept to *multiparty session types* [28, 29], which generalise the framework to handle protocols with arbitrarily many participants and asynchronous communication. This seminal work has driven significant progress in both the theory of communication-based systems [15, 9, 35, 6, 45] and their implementation in mainstream programming languages [1, 34, 16, 2].

Except from some cases [44, 19, 25, 10], the correctness results of the theory of multiparty session types rely on paper proofs, which necessitate the omission of numerous details. Known pitfalls, such as incorrect results caused by issues with binders [17, 49], or unsound projection functions [46], often arise from overlooking technical details that are nearly impossible to catch in lengthy paper proofs.

A common way to make sure that all of these details are found, and to ensure that all proofs are correct, is to use interactive proof assistants like Isabelle [40], Coq [37], or Lean [38], to have all proofs machine-checked by a computer. Proof assistants have proven immensely successful and have been used to guarantee the correctness of a wide variety of foundational results in both mathematics [21–24] and computer science [32, 5]. Nevertheless, attractive as these mechanisations are, they are also very difficult to implement. The original multiparty session types paper [28], recipient of the Most Influential Paper POPL 2008 award [36], should by all means be a prime candidate for mechanisation but has, largely because of the difficulty of the proofs, mostly been left untouched.

In this paper, we mechanise the subject reduction theorem for the theory of multiparty asynchronous session types, as stated in the journal version of the original paper [29]. Subject reduction states that well-typed programs remain well-typed throughout their execution, as defined by their operational semantics. Subject reduction is one of the core theorems in the meta-theory, supporting key results such as type safety, session fidelity, and in-session deadlock-freedom. Although the proofs of the original paper are quite detailed, they are also lengthy and intricate. During our mechanisation efforts, however, we discovered that subject reduction, as presented in the original work, does not hold. In this paper, we provide a counterexample to the original theorem, and we present a new type system for which subject reduction does in fact hold. Our subject reduction proof is fully machine-checked using the Coq proof assistant.

Multiparty Session Types. Multiparty session types use *global types*, a protocol specification language for describing the interaction patterns among multiple communicating participants. Interactions feature a sender and a receiver, generally called *roles*. For example, the following global type specifies the communication patterns between roles p, q, and r:

$$G = \mathsf{p} \to \mathsf{q}\colon 1\left\{\begin{array}{l} l_1 \colon \mathsf{q} \to \mathsf{r} : 2\langle \mathsf{bool}\rangle.\mathsf{end} \\ l_2 \colon \mathsf{p} \to \mathsf{r} : 2\langle \mathsf{bool}\rangle.\mathsf{end} \end{array}\right\} \tag{1}$$

This global type describes a protocol where role p internally decides and informs role q over channel 1, whether to continue in branch l_1 or l_2. A channel in the protocol represents a private FIFO queue, shared between the three roles for communication. If p picks branch l_1, role q, after receiving this label, must send a boolean to r over channel 2. Conversely, if q picks branch l_2, p must then send a boolean to r over the same channel 2. In both branches, the protocol terminates after p or q communicate with r. A key observation is that global types describe asynchronous protocols, meaning communications happen asynchronously. In this example, role q must first receive label l_1 from p before sending a boolean on channel 2. However, if p chooses branch l_2, it can immediately send a boolean without waiting for q to read from channel 1. Global types are equipped with a *reduction semantics* that captures this asynchronous behaviour.

While global types provide a holistic view of a protocol, the behaviour of its individual roles can be described as *local types*. Local types provide a local view of the protocol for each of its roles and can be obtained from global types by an

operation known as *projection*. For example, the global type from Equation (1) can be expressed as the following collection of local types:

$$\mathsf{p} : 1 \oplus \left\{ \begin{array}{l} l_1 : \mathsf{end} \\ l_2 : !2\langle \mathsf{bool} \rangle.\mathsf{end} \end{array} \right\} \qquad \mathsf{q} : 1 \& \left\{ \begin{array}{l} l_1 : !2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2 : \mathsf{end} \end{array} \right\} \qquad \mathsf{r} : ?2\langle \mathsf{bool} \rangle.\mathsf{end} \tag{2}$$

The local type for p says that label l_1 or label l_2 must be communicated over channel 1 and if label l_2 is chosen then a boolean must be sent over channel 2. Dually, role q must be ready to receive label l_1 or l_2 on channel 1, and if label l_1 is selected, a boolean must be sent over channel 2. Finally, role r is ready to receive a boolean from channel 2. Similar to global types, collections of local types such as the one seen in Equation (2), are equipped with a semantics that captures the various communications that can be performed.

Protocols described as global types are executed as *sessions*. A session is an instance of a protocol with private channels where each role is implemented as an executable process, written in the multiparty session calculus, a model of computation for concurrent system based on the π-calculus [39]. The *core idea of multiparty session types* is that global types can be projected into local types, which are then used by a type system to verify that a given implementation conforms to the specification provided by the global types. For example, multiparty session types ensure that the following parallel composition of processes can participate in a session that implements the global type from Equation (1):

$$\begin{array}{l} \overline{\mathsf{a}}[\mathsf{p}]_2(\mathfrak{s}).\overbrace{\mathfrak{s}[1] \lhd l_2; \mathfrak{s}[2]!\langle \mathsf{true} \rangle; \mathbf{0}}^{P} \quad | \\ \mathsf{a}[\mathsf{q}](\mathfrak{s}).\underbrace{\mathfrak{s}[1] \rhd \{l_1 : \mathfrak{s}[2]!\langle \mathsf{false} \rangle;\ \mathbf{0},\quad l_2 : \mathbf{0}\}}_{Q} \quad | \quad \mathsf{a}[\mathsf{r}](\mathfrak{s}).\underbrace{\mathfrak{s}[2]?(x); \mathbf{0}}_{R} \end{array} \tag{3}$$

The term above reads as follows: three processes are about to initiate a 3-party protocol on a, where a is a *shared name* denoting the name of the protocol in the calculus. The first process, prefixed with $\overline{\mathsf{a}}[\mathsf{p}]_2(\mathfrak{s})$, is requesting the initiation of a session featuring two queues (instantiating the channels specified in the global type), acting as p while the second and third process are performing a session accept, joining the session as roles q and r. Process P denotes a process that, playing role p, first puts a label l_2 in the queue at channel 1 and then puts the message true in queue at channel 2. Process Q awaits for a label on queue at channel 1 that determines its continuation. Finally, process R, playing role r, receives a value on the queue at channel 2. Initially, we synchronise over the prefixes $\overline{\mathsf{a}}[\mathsf{p}]_2(\mathfrak{s})$, $\mathsf{a}[\mathsf{q}](\mathfrak{s})$, and $\mathsf{a}[\mathsf{r}](\mathfrak{s})$, yielding the term

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s}) \left(\begin{array}{l} \overbrace{\mathsf{s}^{\mathsf{p}}[1] \lhd l_2; \mathsf{s}^{\mathsf{p}}[2]!\langle \mathsf{true} \rangle; \mathbf{0}}^{P[\mathsf{s}^{\mathsf{p}}/\mathfrak{s}]} \quad | \quad \overbrace{\mathsf{s}^{\mathsf{q}}[1] \rhd \{l_1 : \mathsf{s}^{\mathsf{q}}[2]!\langle \mathsf{false} \rangle; \mathbf{0}, l_2 : \mathbf{0}\}}^{Q[\mathsf{s}^{\mathsf{q}}/\mathfrak{s}]} \quad | \\ \underbrace{\mathsf{s}^{\mathsf{r}}[2]?(x); \mathbf{0}}_{R[\mathsf{s}^{\mathsf{r}}/\mathfrak{s}]} \quad | \quad \mathsf{s}[1] :: \emptyset \quad | \quad \mathsf{s}[2] :: \emptyset \end{array} \right) \tag{4}$$

The operation generates the new session identifier s (the syntax $(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})$ is a restriction binder), which we use to replace the binder $\mathfrak{s}$ with *located session identifiers*

s^{p}, s^{q} and s^{r} for roles p, q, and r respectively. The two communication queues, represented by $\mathsf{s}[1]$ and $\mathsf{s}[2]$, where messages can be enqueued and dequeued, are shared between all participants and are initially empty. Having role p, implemented by process $P[\mathsf{s}^{\mathsf{p}}/\mathit{s}]$, execute its outputs results in the following term:

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\Big(\mathbf{0} \quad | \quad Q[\mathsf{s}^{\mathsf{q}}/\mathit{s}] \quad | \quad R[\mathsf{s}^{\mathsf{r}}/\mathit{s}] \quad | \quad \mathsf{s}[1]::l_2{}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2]::\mathsf{true}^{\mathsf{p}} \cdot \emptyset\Big) \tag{5}$$

where the label $l_2{}^{\mathsf{p}}$ and the value $\mathsf{true}^{\mathsf{p}}$, both annotated with the sending role p, have been enqueued on $\mathsf{s}[1]$ and $\mathsf{s}[2]$ respectively.

A Counter-Example to Subject Reduction. Subject reduction states that any term typed by a collection of local types (referred to as the *typing environment*), derived from a global type through projection, will remain typable after the process reduces. Since local types are equipped with a reduction semantics, the subject reduction theorem also states that the term obtained after reduction can be typed either by the same typing environment or by a modified environment, reflecting a reduction of local types. The original statement of subject reduction by Honda et al. can be informally stated as follows:

> *If P is well typed in a typing environment Δ, and P reduces to P' then either P' is well-typed in the original typing environment Δ, or in another environment Δ' such that Δ can reduce to Δ'*

The reduction on the environment Δ is crucial because it establishes a connection between the reductions performed by the processes and the protocol specification defined by the local types (which are derived from global types). Proving other results, such as session fidelity (processes behave according to protocol descriptions), heavily relies on the relationship between the reductions of processes and types. Unfortunately, the original theory by Honda et al. does not satisfy subject reduction. The key issue lies in the fact that some well-typed processes can reduce to well-typed terms in a way where their typing environments are not the same nor does the initial environment reduce to the latter . We demonstrate this with our running example.

Even though the process in Equation (3) has been reduced to the one in (4) and then to (5), this new process is still typable by the same local types in Equation (2), derived by means of projection from the global type in Equation (1). This is because the semantics of environments tracks interactions, which matches an output action on a channel with an input action on the same channel. So far only outputs have been performed, and the environment is thus unchanged. However, allowing our process to reduce one more step into

$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\Big(\mathbf{0} \quad | \quad Q[\mathsf{s}^{\mathsf{q}}/\mathit{s}] \quad | \quad \mathbf{0} \quad | \quad \mathsf{s}[1]::l_2{}^{\mathsf{p}} \cdot \emptyset \quad | \quad \mathsf{s}[2]::\emptyset\Big) \tag{6}$$

reveals a problem. Here, the process $\mathsf{s}^{\mathsf{r}}[2]?(x);\mathbf{0}$, acting as role r, has consumed the boolean $\mathsf{true}^{\mathsf{p}}$ from the $\mathsf{s}[2]$ channel while $l_2{}^{\mathsf{q}}$ is still in the queue. In other words, the branching interaction between p and q described by the global type in Equation (1) has only been partially performed, since q has not yet received l_2, but the communication of a boolean from p to r in the l_2 branch has been

completely performed. Below (left) we show a global type that describes the remaining behaviour of the session, and we show (right) the environment derived from it.

$$\mathsf{p} \to \mathsf{q} : 1\left\{\begin{matrix} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{matrix}\right\} \qquad \mathsf{p} : 1 \oplus \left\{\begin{matrix} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{matrix}\right\}, \quad \mathsf{q} : 1 \& \left\{\begin{matrix} l_1 : \mathsf{end} \\ l_2 : \mathsf{end} \end{matrix}\right\}, \quad \mathsf{r} : \mathsf{end} \tag{7}$$

This environment types the process in Equation (6). The problem is that the two environments in Equations (2) and (7) are neither the same, nor can they be related by the semantics of local types, therefore breaking subject reduction.

A reasonable question to ask at this point is why, given how influential the work by Honda et al. is, has this not been noticed before? The answer is most likely twofold. Firstly, the proofs are very complex and have never been mechanised. Secondly, the original formulation of multiparty session types differs from most later work. In particular, they use explicit channels where all roles in a session share queues, as opposed to implicit channels, introduced by Bettini et al. [4], where each ordered pair of roles has a dedicated queue. Our counterexample hinges on the use of explicit channels since it would not have been possible to derive it with implicit channels.

Contributions and Structure. The following is a detailed description of the contributions of this paper:

- We identify a counterexample to subject reduction, highlighting a flaw in the original meta-theory of multiparty asynchronous session types (Section 1)
- We prove a projection theorem (establishing the correspondence between the semantics of global and local types) for a novel definition of projection [46] (Section 3)
- We prove the decidability of linearity for global types, a well-formedness condition on global types first introduced in the original multiparty session types paper [28, 29] (Section 3)
- We present a complete revision of the meta-theory of asynchronous multiparty session types that includes a new type system for the asynchronous multiparty session calculus, based on a more general projection function [46], for which we prove subject reduction (Sections 4 and 5)
- All proofs have been machine-checked in the Coq proof assistant (see supplementary material), marking this as the first mechanisation of subject reduction for asynchronous multiparty session types.

2 Multiparty Asynchronous Session Processes

Our process language is an extension of the π-calculus with the notion of session which forms the foundation for inter-process communication.

Syntax. Let $\mathscr{P}$ be a finite subset of the natural numbers which denotes a set of roles (ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \mathsf{t}$), let $\mathscr{L}$ be a set of labels (ranged over by l) and let I, J denote index sets of form $\{1, \ldots, n\}$ for any natural number n. Moreover, let indices in an index set be ranged over by indices i, j. We identify two distinct

a shared name	x value variable	a shared name non-terminal
s^{p} located session	$\mathfrak{s}$ located session variable	s located session non-terminal
s session identifier	k channel	

Fig. 1. Naming of Key Elements

sets of atoms, *shared names*, ranged over by $\mathsf{a}, \mathsf{b}, \mathsf{c}$, and *session identifiers*, ranged over by $\mathsf{s}, \mathsf{t}, \mathsf{r}$. Note that both shared names and session identifiers are in sans-serif font. Multiparty asynchronous session processes are defined as follows:

$$
\begin{array}{lll}
a ::= x \mid \mathsf{a} & \text{(shared names)} & s ::= \mathfrak{s} \mid \mathsf{s}^{\mathsf{p}} \quad \text{(session channels)} \\
v ::= \mathsf{a} \mid \mathsf{true} \mid \mathsf{false} & \text{(values)} & e ::= x \mid v \mid e \ \mathsf{and}\ e' \quad \text{(expressions)} \\
h ::= l^{\mathsf{p}} \mid v^{\mathsf{p}} \mid (\mathsf{s}^{\mathsf{q}})^{\mathsf{p}} & \text{(message)} & D ::= \{X_i(x\tilde{\mathfrak{s}}) = P_i\}_{i \in I} \quad \text{(declarations)}
\end{array}
$$

$$
\begin{array}{lll}
P ::= \overline{a}\,[\mathsf{n}]_k(\mathfrak{s}).P & \text{(session request)} & \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \quad \text{(conditional branch)} \\
\mid a[\mathsf{p}](\mathfrak{s}).P & \text{(session acceptance)} & \mid P \mid Q \quad \text{(parallel composition)} \\
\mid s[k]!\langle e\rangle; P & \text{(value sending)} & \mid \mathbf{0} \quad \text{(inaction)} \\
\mid s[k]?(x); P & \text{(value reception)} & \mid (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \quad \text{(name hiding)} \\
\mid s[k]!\langle\!\langle t\rangle\!\rangle; P & \text{(session delegation)} & \mid (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \quad \text{(session hiding)} \\
\mid s[k]?((\mathfrak{s})); P & \text{(session reception)} & \mid X\langle e, \tilde{s}\rangle \quad \text{(process call)} \\
\mid s[k] \lhd l; P & \text{(label selection)} & \mid \mathsf{s}[k] :: \tilde{h} \quad \text{(message queue)} \\
\mid s[k] \rhd \{l_i\colon P_i\}_{i \in I} & \text{(label branching)} &
\end{array}
$$

A shared name non-terminal $a, b, c, \ldots$ can be either a value variable, denoted by $x, y, z, \ldots$, or a shared name. Located session non-terminals, denoted by $s, t, r, \ldots$, can be either a located session variable, denoted by $\mathfrak{s}, \mathfrak{t}, \mathfrak{r}, \ldots$, or a located session, denoted by $\mathsf{s}^{\mathsf{p}}, \mathsf{t}^{\mathsf{p}}, \mathsf{r}^{\mathsf{p}}$. Figure 1 gives an overview of the various names.

A multi-cast session request $\overline{a}\,[\mathsf{n}]_k(\mathfrak{s}).P$ requests to start a session on a, with roles $\{\mathsf{0}, ..., \mathsf{n}\}$ and the requester acting as n. When a session eventually is initiated on a concrete shared name a, a fresh session identifier s is created. The requester acts as n by referencing variable channel $\mathfrak{s}$ in P which, during the execution, is substituted for a located session s^{n}. The natural number k declares how many channels (queues) will be created when the session is initiated and each queue is addressed as $\mathsf{s}[i]$ for $i \in \{1, ..., k\}$. The term $a[\mathsf{p}](\mathfrak{s}).P$ denotes the dual process that accepts a request on a, playing role p.

The construct $s[k]!\langle e\rangle; P$ denotes the sending of expression e (after its evaluation to a value) to the queue of session s addressed at $\mathsf{s}[k]$. The construct $s[k]?(x); P$ reads from $\mathsf{s}[k]$, with variable x binding the received value in the continuation P. The construct $s[k]!\langle\!\langle t\rangle\!\rangle; P$ denotes sending a session channel, an operation called delegation. The construct $s[k]?((\mathfrak{s})); P$ denotes receiving a session channel, an operation called session reception. In session reception, we bind a located session variable $\mathfrak{s}$ in the continuation P, similarly to the request and accept constructs. Branching communications are captured by $s[k] \lhd l; P$ and $s[k] \rhd \{l_i\colon P_i\}_{i \in I}$. The former selects a branch by communicating label l, while the latter reacts to a branch that another process has chosen (external choice).

The terms $(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P$ and $(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P$ respectively bind shared names and session identifiers in P. Free shared names and free session identifiers, denoted by $\mathsf{fn}(Q)$ and $\mathsf{fs}(S)$ respectively, are defined in a standard way. The process call $X\langle e, \tilde{s}\rangle$ allows recursive definitions by calling procedures stored in the declarations D. Process definitions have shape $X(x\tilde{\jmath}) = P$, with parameters x and $\tilde{\jmath}$ instantiated respectively as a value and a list of distinct located sessions. Definitions may contain process variables and this allows recursion. The terminated process is $\mathbf{0}$.

A queue addressed at $\mathsf{s}[k]$ with messages $\tilde{h}$ is denoted by $\mathsf{s}[k] :: \tilde{h}$, and a message h is either a label, a value, or a session name. A message contains an annotation of the role that has sent the message, e.g., $\mathsf{s}[k] :: \mathsf{true}^{\mathsf{p}} \cdot \tilde{h}$ contains a message from p. Syntactically, variables cannot be messages in a queue. This is sensible because the process semantics introduced next enforces that all variables are substituted for concrete terms before being put on a queue.

We define two types of *substitution*: one on value variables and one on located session variables. Both are denoted by the standard syntax $P[\mathsf{a}/x]$ and $P[\mathsf{s}^{\mathsf{p}}/t]$.

Reduction Semantics. We now give a reduction semantics for processes. First, we must define the structural congruence relation.

Definition 1 (Structural Congruence). *Structural congruence, denoted by $P \equiv Q$, is the smallest equivalence relation that satisfies*

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})\mathbf{0} \equiv \mathbf{0} \qquad ((\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P) \mid Q \equiv (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})(P \mid Q) \text{ when } \mathsf{a} \notin \mathsf{fn}(Q)$$
$$(\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\mathbf{0} \equiv \mathbf{0} \qquad ((\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P) \mid Q \equiv (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})(P \mid Q) \text{ when } \mathsf{s} \notin \mathsf{fs}(Q)$$

The first line makes processes a commutative monoid over parallel composition with $\mathbf{0}$ as the unit element. The next lines give the structural rules for name restriction, allowing the binder to be dropped when its scope is a terminated process, and allowing the scope to be extruded when this is capture-free.

The reduction semantics for processes, denoted by $P \rightarrow_D P'$, is defined by the rules given in Figure 2. Rule [Link] initiates a session between $\{0, ..., \mathsf{n}\}$ roles. The request is made by n and the remaining processes accept on the same shared name a, declaring with $[\mathsf{i}]$ which role of the session they will perform as. The requester indicates by k how many queues will be initiated for the session. This information does not need to be shared with the other roles because the type system will ensure queue addresses $\mathsf{s}[k]$ are used correctly. The parallel composition of the accepting processes is defined using $\prod$ to denote parallel composition of a set of processes. We assume that the natural number n is greater than 0, i.e., there are at least two processes engaging in a session.

Rule [Send] puts a value at the end of a queue, annotating the value with the role p of the located session s^{p} it was sent on, and [Recv] reads from the front of the queue. Rules [Deleg] and [SRec] are the corresponding rules for delegation and session reception while [Label] and [Branch] are the corresponding rules for labels. Rules [IfT] and [IfF] are standard semantics for if-statements. Rule [Def] replaces process variable X with its declaration in D, substituting x for the evaluation of e and substituting $\tilde{\jmath}$ for a list of located sessions. Rules

$$[\textsc{Link}]\ \Big(\prod_{i=0}^{n-1} \mathsf{a}[\mathsf{i}](\vec{s}).P_i\Big) \mid \overline{\mathsf{a}}\,[\mathsf{n}]_k(\vec{s}).P_{\mathsf{n}} \ \to_D\ (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})\Big(\prod_{i=0}^{n-1} P_i[\mathsf{s}^{\mathsf{i}}/\vec{s}] \mid \prod_{j=1}^{k} \mathsf{s}[j] :: \epsilon\Big)$$

$$[\textsc{Send}]\ \mathsf{s}^{\mathsf{p}}[k]!\langle e\rangle; P \mid \mathsf{s}[k] :: \tilde{h} \ \to_D\ P \mid \mathsf{s}[k] :: \tilde{h} \cdot v^{\mathsf{p}} \qquad \boxed{e \downarrow v}$$

$$[\textsc{Recv}]\ \mathsf{s}^{\mathsf{p}}[k]?(x); P \mid \mathsf{s}[k] :: v^{\mathsf{q}} \cdot \tilde{h} \ \to_D\ P[v/x] \mid \mathsf{s}[k] :: \tilde{h}$$

$$[\textsc{Deleg}]\ \mathsf{s}^{\mathsf{p}}[k]!\langle\langle \mathsf{t}^{\mathsf{q}}\rangle\rangle; P \mid \mathsf{s}[k] :: \tilde{h} \ \to_D\ P \mid \mathsf{s}[k] :: \tilde{h} \cdot (\mathsf{t}^{\mathsf{q}})^{\mathsf{p}}$$

$$[\textsc{SRec}]\ \mathsf{s}^{\mathsf{p}}[k]?((\vec{s})); P \mid \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{r}})^{\mathsf{q}} \cdot \tilde{h} \ \to_D\ P[\mathsf{t}^{\mathsf{r}}/\vec{s}] \mid \mathsf{s}[k] :: \tilde{h}$$

$$[\textsc{Label}]\ \mathsf{s}^{\mathsf{p}}[k] \lhd l; P \mid \mathsf{s}[k] :: \tilde{h} \ \to_D\ P \mid \mathsf{s}[k] :: \tilde{h} \cdot l^{\mathsf{p}}$$

$$[\textsc{Branch}]\ \mathsf{s}^{\mathsf{p}}[k] \rhd \{l_i : P_i\}_{i\in I} \mid \mathsf{s}[k] :: l_j{}^{\mathsf{q}} \cdot \tilde{h} \ \to_D\ P_j \mid \mathsf{s}[k] :: \tilde{h} \qquad \boxed{j \in I}$$

$$[\textsc{IfT}]\ \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \ \to_D\ P \qquad \boxed{e \downarrow \mathsf{true}}$$

$$[\textsc{IfF}]\ \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \ \to_D\ Q \qquad \boxed{e \downarrow \mathsf{false}}$$

$$[\textsc{Def}]\ X\langle e, \tilde{\mathsf{s}^{\mathsf{p}}}\rangle \ \to_D\ P[v/x][\tilde{\mathsf{s}^{\mathsf{p}}}/\tilde{s}] \qquad \boxed{e \downarrow v,\ X(\tilde{x s}) = P \in D}$$

$$[\textsc{Res}_{\boldsymbol{\nu}^{\mathsf{s}}}]\ P \ \to_D\ P' \ \Rightarrow\ (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P \ \to_D\ (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s})P'$$

$$[\textsc{Res}_{\boldsymbol{\nu}^{\mathsf{a}}}]\ P \ \to_D\ P' \ \Rightarrow\ (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P \ \to_D\ (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a})P'$$

$$[\textsc{Par}]\ P \ \to_D\ P' \ \Rightarrow\ P \mid Q \ \to\ P' \mid Q$$

$$[\textsc{Str}]\ P \equiv P' \text{ and } P' \ \to_D\ Q' \text{ and } Q' \equiv Q \ \Rightarrow\ P \ \to_D\ Q$$

Fig. 2. Process semantics

[Res$_{\boldsymbol{\nu}^{\mathsf{s}}}$], [Res$_{\boldsymbol{\nu}^{\mathsf{a}}}$] and [Par] allow reductions to occur under restriction binders and parallel composition. Rule [Str] handles structural congruence.

Remark 1 (Differences with the original work). We annotate session identifiers s with a role p, yielding a located session s^{p}. Such annotations have become a common practice in session types papers. It was introduced as *polarities* for binary session types [18, 49], which address unintended limitations of process semantics. As in previous work [4, 13], we annotate messages with the sending role to indicate the sender of the message. This allows to compare the specification of p against the messages it has put on queues and its remaining actions.

3 Global Types, Local Types, and Projection

Syntax. The syntax of global and local types is given as follows:

$$G \ ::=\ \mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \mid \mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j\in J} \mid \mu\mathbf{t}.G \mid \mathbf{t} \mid \mathsf{end}$$

$$T \ ::=\ !k\langle U\rangle.T \mid ?k\langle U\rangle.T \mid k \oplus \{l_i : T\}_{i\in I} \mid k\ \&\ \{l_i : T_i\}_{i\in I} \mid \mu\mathbf{t}.T \mid \mathbf{t} \mid \mathsf{end}$$

$$U \ ::=\ \mathsf{bool} \mid \mathsf{int} \mid G \mid T \qquad \hat{\Delta} \ ::=\ \emptyset \mid \hat{\Delta},\ \mathsf{p} : T$$

The global type $\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G$ specifies a session with an interaction where p sends a message of type U to q via channel k, and G specifies the remaining session. The type $\mathsf{p} \to \mathsf{q} : k\{l_j : G_j\}_{j\in J}$ specifies a branching interaction

where p sends a label l_i to q via k, and G_i specifies the remaining session. The constructs $\mu\mathbf{t}.G$ and $\mathbf{t}$ are used to write tail-recursive specifications such as $\mu\mathbf{t}.\mathsf{p} \to \mathsf{q} : k\langle U\rangle.\mathbf{t}$. Finally, end specifies the terminated session. A message type U is either a boolean, an integer[1], a global type or a local type. Global types are used as message types when communicating shared names and local types are used when communicating session channels.

The local type $!k\langle U\rangle.T$ (resp. $?k\langle U\rangle.T$) specifies a role that sends (resp. receives) a message of type U over channel k, with T representing the remaining actions. The type $k \oplus \{l_i : T\}_{i\in I}$ (resp. $k \,\&\, \{l_i : T_i\}_{i\in I}$) specifies sending (resp. receiving) a label l_i, with T_i being the subsequent actions. As for global types, $\mu\mathbf{t}.T$ and $\mathbf{t}$ allow tail-recursive specifications. Lastly, end is a terminated role.

We deal with recursive variables in a standard way and write capture-avoiding substitution as $G_1[G_2/\mathbf{t}]$ (resp. $T_1[T_2/\mathbf{t}]$). We assume that types are closed and contractive. Global and local types are contractive if, for any of its sub-expressions with shape $\mu\mathbf{t}_0.\mu\mathbf{t}_1...\mu\mathbf{t}_n.\mathbf{t}$, the body $\mathbf{t}$ is not $\mathbf{t}_0$ [43].

A *local type environment* $\hat{\Delta}$ is a map from roles to local types. For every p in the domain of $\hat{\Delta}$ (denoted by $\mathsf{dom}(\hat{\Delta})$), there is a unique entry $\mathsf{p} : T$ for some T.

Projection. Projection is a partial function which, given a global type and a role, attempts to generate a local type that captures the specification of this role in the global type. When the global type specifies behavior that the role cannot implement, projection is undefined. Our mechanisation uses a variant projection function $\mathsf{proj}_{\mathsf{p}}(G)$ by Tirore et al. [46], which is defined in Figure 3. The projection function $\mathsf{proj}_{\mathsf{p}}(G)$ is defined in terms of two auxiliary definitions: a translation function, denoted by $\mathsf{trans}_{\mathsf{p}}(G)$, and a projectability predicate, denoted by $\mathsf{projectable}_{\mathsf{p}}(G)$. The translation function is a total function which produces a local type representing the local behavior of the translated role. The projectability predicate checks that the global type specifies behavior that is possible to implement by the projected role.

Translation is defined such that $\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U\rangle.\ G)$ produces the output $!k\langle U\rangle$ (resp. input $?k\langle U\rangle$) when the translated role is p_1 (resp. p_2), and translates the remaining global type as $\mathsf{trans}_{\mathsf{p}}(G)$. If the translated role does not occur in the interaction, translation simply ignores the interaction proceeds as $\mathsf{trans}_{\mathsf{p}}(G)$. The translation of a branching global type $\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j\in J})$ is similar when p occurs in the interaction. When p does not occur, translation chooses the first branch j_1 and proceeds as $\mathsf{trans}_{\mathsf{p}}(G_1)$. The auxiliary predicate $\mathsf{gVar}(\mathsf{t}, G)$ is used in the translation of $\mu\mathbf{t}.G$, and checks if $\mu\mathbf{t}$ can safely be added to $\mathsf{trans}_{\mathsf{p}}(G)$, without producing non-contractive local types. In particular, this predicate ensures that trans never produces non-contractive types like $\mu\mathbf{t}.\mathbf{t}$. Finally, $\mathsf{trans}_{\mathsf{p}}(\mathbf{t})$ (resp. $\mathsf{trans}_{\mathsf{p}}(\mathsf{end})$) returns $\mathbf{t}$ (resp. end).

[1] Our mechanization excludes the integer message type, as the expressiveness of the expression language is orthogonal to our focus. We include the integer type here solely to simplify examples with distinct message types.

$$\mathsf{gVar}(\mathsf{t}, G) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{gVar}(\mathsf{t}, G_1) \ \wedge \ \mathbf{t} \neq \mathbf{t}' & \text{if } G = \mu \mathbf{t}'.G_1 \\ \mathbf{t} \neq \mathbf{t}' & \text{if } G = \mathbf{t}' \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle.G) \stackrel{\mathsf{def}}{=} \begin{cases} !k\langle U \rangle.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ ?k\langle U \rangle.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G) & \text{if } \mathsf{p} \notin \{\mathsf{p}_1, \mathsf{p}_2\} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathsf{p}_1 \to \mathsf{p}_2 : k\{l_j : G_j\}_{j \in J}) \stackrel{\mathsf{def}}{=} \begin{cases} k \oplus \{l_j : \mathsf{trans}_{\mathsf{p}}(G_j)\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_1 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ k \ \& \ \{l_j : \mathsf{trans}_{\mathsf{p}}(G_j)\}_{j \in J} & \text{if } \mathsf{p} = \mathsf{p}_2 \text{ and } \mathsf{p}_1 \neq \mathsf{p}_2 \\ \mathsf{trans}_{\mathsf{p}}(G_1) & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mu \mathbf{t}.G) \stackrel{\mathsf{def}}{=} \begin{cases} \mu \mathbf{t}.(\mathsf{trans}_{\mathsf{p}}(G)) & \text{if } \mathsf{gVar}(\mathsf{t}, G) \\ \mathsf{end} & \text{otherwise} \end{cases}$$

$$\mathsf{trans}_{\mathsf{p}}(\mathbf{t}) \stackrel{\mathsf{def}}{=} \mathbf{t} \qquad \mathsf{trans}_{\mathsf{p}}(\mathsf{end}) \stackrel{\mathsf{def}}{=} \mathsf{end}$$

$$\mathsf{proj}_{\mathsf{p}}(G) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{trans}_{\mathsf{p}}(G) & \text{if } \mathsf{projectable}_{\mathsf{p}}(G) \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

$$\mathsf{proj\_roles}(G, \ \{\mathsf{p}_1, \ \cdots, \ \mathsf{p}_n\}) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{p}_1 : \mathsf{proj}_{\mathsf{p}_1}(G), \ \cdots, & \text{if } \forall i \in \{1, \ \cdots, \ n\}. \\ \mathsf{p}_n : \mathsf{proj}_{\mathsf{p}_n}(G) & \mathsf{proj}_{\mathsf{p}_i}(G) \text{ is defined} \\ \mathit{undefined} & \mathit{otherwise} \end{cases}$$

$$\mathsf{full\_proj}(G) \stackrel{\mathsf{def}}{=} \mathsf{proj\_roles}(G, \ \mathsf{roles}(G))$$

Fig. 3. Projection

On its own, the translation function is too permissive. E.g., the global type

$$\mathsf{p} \to \mathsf{q} : 1\left\{ \begin{array}{l} l_1 : \mathsf{r} \to \mathsf{s} : 2\langle \mathsf{bool} \rangle.\mathsf{end} \\ l_2 : \mathsf{r} \to \mathsf{s} : 2\langle \mathsf{int} \rangle.\mathsf{end} \end{array} \right\}$$

specifies that p sends l_1 or l_2 to q and depending on this choice, r has to send a boolean or an integer to s. The problem is that r cannot know which choice was made, and its specification is thus not possible to implement. Projection on r should therefore be undefined for this global type. As a total function, translation naively ignores this and translates the l_1 branch, producing the local type $!2\langle \mathsf{bool} \rangle.\mathsf{end}$. If p chooses l_2, r will incorrectly behave as if l_1 was chosen.

The problem in the example above is solved by letting projection, which we denote $\mathsf{proj}_{\mathsf{p}}(G)$, be defined only when a projectability predicate $\mathsf{projectable}_{\mathsf{p}}(G)$ holds, in which case, the returned local type is computed as $\mathsf{trans}_{\mathsf{p}}(G)$. This predicate depends on a coinductive specification of projection that will not be

$$\frac{}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle U\rangle} G}\ [\text{GR1}] \qquad \frac{j \in I}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle l_j\rangle} G_j}\ [\text{GR2}]$$

$$\frac{G_1 \xrightarrow{\ell} G_2 \qquad \mathsf{q} \notin \ell}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G_1 \xrightarrow{\ell} \mathsf{p} \to \mathsf{q} : k\langle U\rangle.G_2}\ [\text{GR3}]$$

$$\frac{\forall i \in I.\ G_i \xrightarrow{\ell} G_i' \qquad \mathsf{q} \notin \ell}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \xrightarrow{\ell} \mathsf{p} \to \mathsf{q} : k\{l_i : G_i'\}_{i\in I}}\ [\text{GR4}] \qquad \frac{G[\mu \mathbf{t}.G/t] \xrightarrow{\ell} G'}{\mu \mathbf{t}.G \xrightarrow{\ell} G'}\ [\text{GR5}]$$

Fig. 4. Semantics of Global Types

introduced in this paper. For the definition of the projectable predicate and the coinductive specification we refer the reader to Tirore et al. [46].

The bottom of Figure 3 defines the function full_proj which projects a global type over all its roles, and generates a local type environment $\hat{\Delta}$.

Remark 2 (Differences with the original work.). Projection is notoriously complex to define correctly. Our mechanisation uses the projection by Tirore et al.[46], which, unlike that of Honda et al., handles μ-binders correctly and has been proven sound and complete with respect to the coinductive projection specification by Ghilezan et al.[20].

Semantics of Global Types, Local Types, and Environments. We define the semantics of global types as a labeled transition system, denoted by $G \xrightarrow{\ell} G'$. We call the label ℓ for an *interaction*, formally defined as:

$$\mathcal{U} ::= U \mid l \qquad\qquad \ell ::= \mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$$

The syntactic category $\mathcal{U}$ is either a message type U or a label l, and an interaction $\mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$ is either the exchange of a message type or a label.

The semantics of global types is defined by the rules given in Figure 4. Rules [GR1] and [GR2] allow a global type to reduce by its first interaction. Rules [GR3] and [GR4] allow a global type to reduce by an interaction ℓ that occurs nested in one of its continuations, which makes the semantics asynchronous by allowing p to occur in ℓ. On the process level, this corresponds to p putting a message on the queue to be read by q, followed by p completing interaction ℓ with a different role before q receives the message. The completed interaction ℓ may not involve q because inputs are blocking. Rule [GR5] unfolds a μ-binder.

The semantics of local types is a two-level labeled transition system: the first level gives semantics to local types T, denoted by $T \xrightarrow{\zeta} T'$, and the second level gives semantics to local type environments $\hat{\Delta}$, denoted by $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. We call the label ζ for an *action*, formally defined as:

$$\zeta ::= !k\langle \mathcal{U}\rangle \mid ?k\langle \mathcal{U}\rangle$$

$$\frac{}{!k\langle U\rangle.T \xrightarrow{!k\langle U\rangle} T}\ [\text{LR1}] \qquad \frac{}{?k\langle U\rangle.T \xrightarrow{?k\langle U\rangle} T}\ [\text{LR2}]$$

$$\frac{j \in I}{k \oplus \{l_i : T\}_{i\in I} \xrightarrow{!k\langle l_j\rangle} T_j}\ [\text{LR3}] \qquad \frac{j \in I}{k \,\&\, \{l_i : T_i\}_{i\in I} \xrightarrow{?k\langle l_j\rangle} T_j}\ [\text{LR4}]$$

$$\frac{T \xrightarrow{\zeta} T' \quad \mathsf{ch}(\zeta) \neq k}{!k\langle U\rangle.T \xrightarrow{\zeta} !k\langle U\rangle.T'}\ [\text{LR5}] \qquad \frac{\forall i \in I.\, T_i \xrightarrow{\zeta} T_i' \quad \mathsf{ch}(\zeta) \neq k}{k \oplus \{l_i : T\}_{i\in I} \xrightarrow{\zeta} k \oplus \{l_i : T_i'\}_{i\in I}}\ [\text{LR6}]$$

$$\frac{T[\mu \mathbf{t}.T/\mathbf{t}] \xrightarrow{\zeta} T'}{\mu \mathbf{t}.T \xrightarrow{\zeta} T'}\ [\text{LR7}] \qquad \frac{T_1 \xrightarrow{!k\langle U\rangle} T_1' \qquad T_2 \xrightarrow{?k\langle U\rangle} T_2'}{\hat{\Delta},\ \mathsf{p} : T_1,\ \mathsf{q} : T_2 \xrightarrow{\mathsf{p}\to\mathsf{q}:k\langle U\rangle} \hat{\Delta},\ \mathsf{p} : T_1',\ \mathsf{q} : T_2'}\ [\text{LEnv}]$$

Fig. 5. Semantics of Local Types and Local Type Environments

An action is the sending or receiving of a message type or a label. We write $\mathsf{ch}(\zeta)$ (resp. $\mathsf{ch}(\ell)$) to extract the channel of a local label ζ (resp. global label ℓ).

Formally, we define the semantics of local types and typing environments by the rules given in Figure 5. Rules [LR1], [LR2], [LR3] and [LR4] allow a local type to reduce by its first action. Similar to the global type rules [GR3] and [GR4], rules [LR5] and [LR6] allow a local type to reduce by an action ζ that occurs nested in one of its continuations. These rules make the local type semantics asynchronous. A condition for reducing by a nested action ζ, is that ζ uses a different channel than the first action, i.e., $\mathsf{ch}(\zeta) \neq k$. The intuition is that the first action, which is an output, already has occurred and it has resulted in a value being put on channel k; but before this value has been received by another participant, another interaction has in the meantime completed on another channel k'. Interactions between local types is captured by the semantics of local type environments $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$, given by the single rule [LEnv]. An environment may reduce by $\mathsf{p} \to \mathsf{q} : k\langle \mathcal{U}\rangle$ when it maps p and q to local types that can reduce by $!k\langle \mathcal{U}\rangle$ and $?k\langle \mathcal{U}\rangle$.

Remark 3 (Differences with the original work). We define the semantics of local types differently from Honda et al. Unlike their semantics, ours includes the asynchronous rules [LR5] and [LR6]. Instead, Honda et al. achieve asynchrony by defining a relation that relates local types with permuted outputs; for example, $!k\langle U\rangle.!k'\langle U'\rangle.T$ is related to $!k'\langle U'\rangle.!k\langle U\rangle.T$. Similar to our asynchronous rules, this permutation is only allowed if $k \neq k'$. However, this is a more restrictive semantics, as it misses cases where an output is preceded by a nested input on a different channel, thereby permitting the input to be placed in front of the output. For instance, with the types $!k\langle U\rangle; ?k'\langle U'\rangle; T$ and $?k'\langle U'\rangle; !k\langle U\rangle; T$, for $k \neq k'$, our rule [LR5] with $\zeta = ?k'\langle U\rangle$ allows the output to be preceded by the input. This ability to precede an output with an input is essential for proving an equivalence between the semantics of global types and local type environments, a property known as the *Projection Theorem*. Consequently, their restrictive se-

mantics means that their projection theorem (Lemma 5.11 [29]) does not hold. At the end of this section, we present a projection theorem for our semantics.

Linearity. The Honda et al. formulation of multiparty session types began an entire line of research, but to the best of our knowledge, all future work has adopted the *implicit* channels introduced by Bettini et al. [4]. Although they are more expressive, explicit channels are more complicated, allowing global types to contain races. For example, the global type:

$$\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int} \rangle.\ \mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool} \rangle.\mathsf{end} \tag{8}$$

is problematic because the two interactions $\mathsf{p} \to \mathsf{q} : k\langle \mathsf{int} \rangle$ and $\mathsf{r} \to \mathsf{q} : k\langle \mathsf{bool} \rangle$ share the same explicit channel k which introduces a race between p and r. If r wins the race, a type error occurs when q receives bool while expecting int.

Linearity is a property that rules out global types with ordered interactions that introduce races. Two interactions are ordered if they occur along the same trace, formally defined as:

Definition 2 (Trace). *A trace ρ of a global type G is a (possibly empty) sequence of interactions such that*

$$\frac{\mathsf{trace}\ \rho\ G}{\mathsf{trace}\ (\mathsf{p} \to \mathsf{q} : k\langle U \rangle \cdot \rho)\ (\mathsf{p} \to \mathsf{q} : k\langle U \rangle.G)} \qquad \frac{\mathsf{trace}\ \rho\ (G[\mu \mathbf{t}.G\ \mathsf{t}])}{\mathsf{trace}\ \rho\ (\mu \mathbf{t}.G)}$$

$$\frac{\mathsf{trace}\ \rho\ G_i}{\mathsf{trace}\ (\mathsf{p} \to \mathsf{q} : k\langle l_i \rangle \cdot \rho)\ (\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i \in I})} \qquad \frac{}{\mathsf{trace}\ \epsilon\ G}$$

Intuitively, a trace records the interactions of a finite walk through the global type, starting from the first interaction and unfolding μ-binders when they occur. We say $\mathsf{p} \to \mathsf{q} : k$ and $\mathsf{r} \to \mathsf{s} : k$ are ordered G if we for some ρ and ρ' can derive:

$$\mathsf{trace}\ (\rho \cdot \mathsf{p} \to \mathsf{q} : k\langle \mathcal{U} \rangle \cdot \rho' \cdot \mathsf{r} \to \mathsf{s} : k\langle \mathcal{U}' \rangle)\ G \tag{9}$$

We omit $\mathcal{U}$ and $\mathcal{U}'$ from the interactions as they provide no causal information. Linearity asserts that ordered interactions that share the same channel have ordered inputs and outputs, a property which is asserted by deriving input and output dependencies, formally defined as:

Definition 3 (Input and Output Dependencies). *The interaction dependencies* $\mathsf{IO}, \mathsf{II}, \mathsf{OO}, \mathsf{IOOO} \subseteq \ell \times \ell$ *and judgments* $\mathsf{input\ dep}\ \rho$ *and* $\mathsf{output\ dep}\ \rho$ *are defined as:*

$$\mathsf{p} \to \mathsf{q} : k\ \mathsf{IO}\ \mathsf{s} \to \mathsf{r} : k' \stackrel{\mathsf{def}}{=} \mathsf{q} = \mathsf{s} \qquad \mathsf{p} \to \mathsf{q} : k\ \mathsf{II}\ \mathsf{r} \to \mathsf{s} : k' \stackrel{\mathsf{def}}{=} \mathsf{q} = \mathsf{s}$$

$$\mathsf{p} \to \mathsf{q} : k\ \mathsf{OO}\ \mathsf{s} \to \mathsf{r} : k' \stackrel{\mathsf{def}}{=} \mathsf{p} = \mathsf{s} \wedge k = k' \qquad \ell_1\ \mathsf{IOOO}\ \ell_2 \stackrel{\mathsf{def}}{=} \ell_1\ \mathsf{IO}\ \ell_2\ \vee\ \ell_1\ \mathsf{OO}\ \ell_2$$

$$\text{[I-Base]}\ \frac{\ell_1\ \mathsf{II}\ \ell_2}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2} \qquad \text{[I-Skip]}\ \frac{\mathsf{input\ dep}\ \ell_1 \cdot \rho \cdot \ell_3}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3} \qquad \text{[I-Tail]}\ \frac{\mathsf{input\ dep}\ \ell_2 \cdot \rho \cdot \ell_3 \qquad \ell_1\ \mathsf{IO}\ \ell_2}{\mathsf{input\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3}$$

$$\text{[O-Base]}\ \frac{\ell_1\ \mathsf{IOOO}\ \ell_2}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2} \qquad \text{[O-Skip]}\ \frac{\mathsf{output\ dep}\ \ell_1 \cdot \rho \cdot \ell_3}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3} \qquad \text{[O-Tail]}\ \frac{\mathsf{output\ dep}\ \ell_2 \cdot \rho \cdot \ell_3 \qquad \ell_1\ \mathsf{IOOO}\ \ell_2}{\mathsf{output\ dep}\ \ell_1 \cdot \ell_2 \cdot \rho \cdot \ell_3}$$

Input and output dependencies are composed of interaction dependencies. Interaction dependencies relate interactions by asserting different ways the same role can occur in both interactions, such as IO (input-output) meaning the input role of the first interaction is also the output role of the second interaction. Likewise we have OO (output-output) and II (input-input) dependencies and we chain interaction dependencies together as input and output dependencies. An input dependency is a sequence of IO dependencies that ends in II while an output dependency is a sequence of IOOO dependencies. We use these chains to show causal orderings between the roles in the first and last interactions of the chain.

We start by considering for each rule of input dependency, why they assert and preserve the ordering of inputs. The shortest input dependency is built by [I-Base]. The two inputs are by the same role and therefore ordered. Rule [I-Skip] captures that chains need not use all interactions of the trace. Finally rule [I-Tail] extends an existing input dependency between ℓ_2 and ℓ_3 by adding ℓ_1 in front of ℓ_2 when it satisfies ℓ_1 IO ℓ_2. We can assume that the inputs of ℓ_2 and ℓ_3 are ordered and this is also the case between ℓ_1 and ℓ_3 if it is the case for ℓ_1 and ℓ_2. If the channels of ℓ_1 and ℓ_2 are different then the inputs are ordered because the input of ℓ_2 cannot intercept the output of ℓ_1. If the channels are the same, then there must exist input and output dependencies between ℓ_1 and ℓ_2, ensuring in particular that their inputs are ordered. More generally, input and output dependencies show that a specific pair of ordered interactions, with the same channel, have ordered inputs and outputs, by relying on this property to hold for all other ordered interactions with the same channel.

Next, we consider why the rules of output dependency preserve ordering of outputs. We start by the shortest output dependency built by [O-Base], relating the interactions by either OO or IO. The former ensures the ordering by having the same role in both outputs. IO preserves output dependency by blocking the output of the second interaction. Rule [O-Skip] is the corresponding skip. Finally, rule [O-Tail] allows output dependencies to be extended either by OO or IO, extending the ordering of the outputs between ℓ_2 and ℓ_3 by adding ℓ_1 in front, as long as the outputs of ℓ_1 and ℓ_2 are ordered as well. The argument for why output ordering is preserved is similar to that of rule [O-Base].

Linearity asserts dependencies between any ordered interactions with the same channel. Unlike Honda et al., we define linearity in terms of the weaker property, $\mathsf{linearHead}(G)$, which simplifies the decision procedure. This formulation only asserts the existence of dependencies when the first of the two interactions is also the first interaction in the trace:

$$\mathsf{trace}\ (\mathsf{p} \to \mathsf{q} : k\langle U \rangle \cdot \rho \cdot \mathsf{r} \to \mathsf{s} : k\langle U' \rangle)\ G \tag{10}$$

$$|G| \stackrel{\text{def}}{=} \begin{cases} 1 + |G_1| & \text{if } G = \mu \mathbf{t}.G_1 \\ 0 & \text{otherwise} \end{cases} \qquad \mathsf{unfold\_once}(G) \stackrel{\text{def}}{=} \begin{cases} T_1[\mu \mathbf{t}.G_1/t] & \text{if } G = \mu \mathbf{t}.G_1 \\ T & \text{otherwise} \end{cases}$$

$$\mathsf{unf}(G) \stackrel{\text{def}}{=} \mathsf{unfold\_once}(|G|)G$$

Fig. 6. Calculating μ-height and unfolding global types. Corresponding operations exist on local types.

We define linearity of a global type, $\mathsf{linear}(G)$, in terms of $\mathsf{linearHead}(G)$ in Figure 7. Intuitively, $\mathsf{linearHead}(G)$ checks if the first interaction is free of races and $\mathsf{next}(G)$ proceeds to the continuation of the global type. Due to global types having only finitely many distinct subterms, a consequence of μ-types being regular [43], only finitely many applications of the coinductive rule of linear is necessary before a previously encountered global type inevitably reappears, at which point the proof can circularly be closed.

The definition of the function $\mathsf{next}(G)$ uses the unfold operation $\mathsf{unf}(G)$ which we define in Figure 6. The definition uses μ-height, denoted by $|G|$, which is the number of top-level μ-binders occurring before another construct appears. For example, $|\mu \mathbf{t}.\mathsf{end}| = 1$ and $|\mathsf{p} \to \mathsf{q} : k\langle U \rangle.G| = 0$. We create an auxiliary function $\mathsf{unfold\_once}(\cdot)$ which unfolds a single binder by turning $\mu \mathbf{t}.G$ into $G[\mu \mathbf{t}.G/\mathbf{t}]$ and define the unfolding operation for global types, denoted by $\mathsf{unf}(G)$, as the repeated application of $\mathsf{unfold\_once}(G)$ μ-height times. This operation can similarly be defined on local types, denoted by $\mathsf{unf}(T)$, and we elide its definition.

$$\mathsf{linearHead}(G) \stackrel{\text{def}}{=} \forall \ell_1\ \rho\ \ell_2.\ \mathsf{trace}\ (\ell_1 \cdot \rho \cdot \ell_2)\ G \to \mathsf{ch}(\ell_1) = \mathsf{ch}(\ell_2) \to \mathsf{input\ dep}\ \ell_1 \cdot \rho \cdot \ell_2 \wedge \mathsf{output\ dep}\ \ell_1 \cdot \rho \cdot \ell_2$$

$$\mathsf{next}(G) \stackrel{\text{def}}{=} \begin{cases} \{G_1\} & \text{if } \mathsf{unf}(G) = \mathsf{p}_1 \to \mathsf{p}_2 : k\langle U \rangle.G_1 \\ \{G_i \mid i \in I\} & \text{if } \mathsf{unf}(G) = \mathsf{p}_1 \to \mathsf{p}_2 : k\{l_i : G_i\}_{i \in I} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\frac{\mathsf{linearHead}(G) \quad \forall G' \in \mathsf{next}(G).\ \mathsf{linear}((G'))}{\mathsf{linear}(G)}$$

Fig. 7. Linearity of global types

Theorem 1 (Decidability of Linearity). $\mathsf{linear}(G)$ *is decidable.*

Decidability of linearity can be reduced to a reachability problem on a graph induced by a global type with $\mathsf{next}(\cdot)$ as adjacency list.

Projection Theorem. We have seen that interactions in a protocol can be described by $G \xrightarrow{\ell} G'$ and $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. Naturally, we want to establish a corre-

spondence between these two semantics with respect to the projection operation. Intuitively, if $\hat{\Delta}$ and $\hat{\Delta}'$ are derived from G and G', respectively, then $G \xrightarrow{\ell} G'$ holds if and only if $\hat{\Delta} \xrightarrow{\ell} \hat{\Delta}'$. However, this result is too strong due to recursion, which may unfold some terms during reduction, making projection slightly different. The issue lies in syntactic equality on local types being too restrictive to prove the projection theorem. Local types, however, can be more loosely related by a coinductive equality on their continuously unfolded terms:

Definition 4 (Coinductive equality). *Coinductive equality is a relation on local types defined by the following rules:*

$$\frac{\mathit{unf}(T) = \mathsf{end} \quad \mathit{unf}(T)' = \mathsf{end}}{T \approx T'} \text{ [Eq-End]}$$

$$\frac{\mathit{unf}(T) = !k\langle U\rangle.T'' \quad \mathit{unf}(T') = !k\langle U\rangle.T''' \quad T'' \approx T'''}{T \approx T'} \text{ [Eq-!]}$$

$$\frac{\mathit{unf}(T) = ?k\langle U\rangle.T'' \quad \mathit{unf}(T') = ?k\langle U\rangle.T''' \quad T'' \approx T'''}{T \approx T'} \text{ [Eq-?]}$$

$$\frac{\mathit{unf}(T) = k \oplus \{l_j : T_j\}_{j\in J} \quad \mathit{unf}(T') = k \oplus \{l_j : T'_j\}_{j\in J} \quad \forall j \in J.\ T_j \approx T'_j}{T \approx T'} \text{ [Eq-}\oplus\text{]}$$

$$\frac{\mathit{unf}(T) = k \,\&\, \{l_j : T_j\}_{j\in J} \quad \mathit{unf}(T') = k \,\&\, \{l_j : T'_j\}_{j\in J} \quad \forall j \in J.\ T_j \approx T'_j}{T \approx T'} \text{ [Eq-\&]}$$

This relation checks the shape of the unfolded local types, and checks whether the same actions are specified, before proceeding to the continuations. Because of unfolding, it might require a circular proof to derive the equality. With an abuse of notation, $\hat{\Delta}_1 \approx \hat{\Delta}_2$ is the point-wise extension of $\approx$ to local type environments. Coinductive equality on continuously unfolded μ-types is decidable [43], and this has been mechanised [14]. Our mechanisation also includes a proof of the decidability of the relation $\approx$, which, to the best of our knowledge, is the first mechanisation for session types. Using this relation, we can state the following:

Theorem 2 (Projection theorem). *Let G be a global type such that* $\mathsf{linear}(G)$ *holds and* $\mathsf{full\_proj}(G)$ *is defined, then*

1. *If $G \xrightarrow{l} G'$ then $\hat{\Delta}$ exists such that* $\mathsf{full\_proj}(G) \xrightarrow{l} \hat{\Delta}$ *and* $\mathsf{proj\_roles}(G', \mathsf{roles}(G)) \approx \hat{\Delta}$
2. *If* $\mathsf{full\_proj}(G) \xrightarrow{l} \hat{\Delta}$ *then G' exists such that $G \xrightarrow{l} G'$ and* $\mathsf{proj\_roles}(G', \mathsf{roles}(G)) \approx \hat{\Delta}$.

Remark 4 (Differences with the original work). The projection theorem of Honda et al. does not take into account that $\mathsf{roles}(G')$ may be a strict subset of $\mathsf{roles}(G)$. The set of roles in a global type can decrease after reduction if the reduced interaction involves a role that only occurs in that interaction. This results in

an invalid environment reduction because the domain shrinks, exemplifying an innocent error that is hard to catch on paper.

4 Queue Types, Environment Decomposition, and Coherence

Queue types. Honda et al. handle the typing of queues using type contexts $\mathcal{T}$, which are local types with a hole $[\cdot]$. They decompose a local type T into a context $\mathcal{T}$ and another local type T', such that the applied context $\mathcal{T}[T']$ is related to T by a subtyping relation (see Definition 5.1 [29]). The type checking of a process against an environment $\hat{\Delta}$, may involve multiple of these decompositions. Mechanising this approach is non-trivial, and for convenience, we take a different approach: we type processes with an environment that contains local types that are decomposed *a priori*. We refer to this as a decomposed environment, denoted by $\hat{\Delta};\hat{\mathcal{Q}}$. We formally define queue environments $\hat{\mathcal{Q}}$, and the queue types Q it contains, as:

Definition 5 (Queue type and Queue Environment). Queue types*, denoted by* $\mathtt{Q}$*, and* queue environments*, denoted by* $\hat{\mathcal{Q}}$*, are defined as:*

$$\mathtt{Q} ::= \emptyset \mid (k, \mathcal{U}) \cdot \mathtt{Q} \qquad\qquad \hat{\mathcal{Q}} ::= \emptyset \mid \hat{\mathcal{Q}},\ \mathsf{p} : \mathtt{Q}$$

A queue environment $\hat{\mathcal{Q}}$ is a map from roles to queue types, and a queue type is a sequence of pairs. In each pair, the left component is always a channel, and the right component is either a message type U or a label l. A pair $(k, \mathcal{U})$ in the queue type models the presence of a message $\mathcal{U}$ in the queue at channel k, and the order of these pairs indicate the order in which these messages were sent. Queue types are given with respect to roles in the environment $\hat{\mathcal{Q}}$, and this environment describes the content of all queues in the session. Queue types are derived from local types using the following notion of path:

Definition 6 (Path). *A path, denoted by* σ*, is a (possibly empty) sequence of elements from the set* $\mathcal{L} \cup \{\bot\}$*.*

A path is similar to a trace, it represents a walk through the μ-type. Unlike traces, we use paths to decompose local types.

Definition 7 (Decomposition of Local Types). *The decomposition of local types, denoted by* $\mathsf{decomp} : \sigma \to T \to T \times Q$*, is defined as:*

$$\mathsf{decomp}_\sigma\ T \stackrel{\mathsf{def}}{=} \begin{cases} (T', (k, U) \cdot \mathtt{Q}) & \mathsf{unf}\ T = !k\langle U\rangle.T_1 \ \wedge\ \sigma = \bot \cdot \sigma' \ \wedge\ \mathsf{decomp}_\sigma\ T_1 = (T', \mathtt{Q}) \\ (T', (k, l_j) \cdot \mathtt{Q}) & \mathsf{unf}\ T = k \oplus \{l_i : T\}_{i \in I} \ \wedge\ \sigma = {}_j \cdot \sigma' \ \wedge \\ & \mathsf{decomp}_\sigma\ T_j = (T', \mathtt{Q}) \quad \wedge\ j \in I \\ (\mathsf{unf}\ T, \emptyset) & \sigma = \emptyset \\ undefined & otherwise \end{cases}$$

With respect to a path, decomposition splits a local type into a pair consisting of a residual local type and a queue type. We overload notation, denoting the

point-wise extension of decomposition onto environments as $\mathsf{decomp}_f\ \hat{\Delta}$, which splits the environment $\hat{\Delta}$, with respect to a path function $f : \mathscr{P} \to \sigma$, into a residual environment $\hat{\Delta}'$ and a queue environment $\hat{\mathcal{Q}}$.

Definition 8 (Decomposition of Environments). *Let the path function f be such that $\forall i \in \{1, \ldots, n\}.\ \mathsf{decomp}_{f(\mathsf{p}_i)}\ T_i = (T_i', \mathtt{Q}_i)$. Environment decomposition, denoted by $\mathsf{decomp} : (\mathscr{P} \to \sigma) \to \hat{\Delta} \to \hat{\Delta} \times \hat{\mathcal{Q}}$, is defined as:*

$$\mathsf{decomp}_f\ (\mathsf{p}_1 : T_1, ..., \mathsf{p}_n : T_n) \quad \stackrel{\mathsf{def}}{=} \quad \mathsf{p}_1 : T_1', ..., \mathsf{p}_n : T_n';\ \ \mathsf{p}_1 : \mathtt{Q}_1, ..., \mathsf{p}_n : \mathtt{Q}_n$$

Example 1. The path function tracks, within a session, the messages that are in transit. Recall that the initial state of our running example from Section 1, Equation (4), has two empty queues. Because the queues are empty, the decomposition of the local types that type the three roles, should all produce the empty queue type $\emptyset$. This is achieved with the path function: $f(\mathsf{p}) = \emptyset$. After p has put a message on each of the two queues in Equation (5), the local type of p must be decomposed in a way that reflects the two outputs have been performed, and that the remaining roles have performed no outputs. We achieve this with the path function: $f(\mathsf{p}') = \mathsf{if}\ \mathsf{p}' = \mathsf{p}\ \mathsf{then}\ l_1 \cdot \bot\ \mathsf{else}\ \emptyset$. The decomposition of $G \upharpoonright_{\mathsf{p}}$ with respect to $f(\mathsf{p})$, where G is the global type in our running example, yields the residual local type end and queue type $(1, l_2) \cdot (2, \mathsf{bool})$.

Coherence. Honda et al. use a well-formedness condition on global types called coherence, which they define as a global type being linear and projectable, i.e., its projection is defined for all roles. However, coherence alone does not rule out our counterexample. For this reason, we introduce an additional property to our definition of coherence, called unstuckness and denoted by $\mathsf{unstuck}\ (G)$.

The idea behind unstuckness is the following. The global type which specifies the counterexample is stuck because the only relevant rule, [GR4], is not applicable due to its restrictive premise, which requires universal quantification $\forall i \in I.\ G_i \xrightarrow{\ell} G_i'$. If this premise was relaxed to existential quantification $\exists i \in I.\ G_i \xrightarrow{\ell} G_i'$, then the global type would not be stuck. It is however not an option to relax the rule because it would break the Projection theorem. Instead, we define $\mathsf{unstuck}\ (G)$, which captures global types where the universally quantified premise implies the existentially quantified premise. The global type in the counterexample does not satisfy this property and is thus ruled out.

In order to define the $\mathsf{unstuck}$ predicate, we use the notion of barb, denoted by $G \downarrow^1 \ell$, formally defined by the rules in Figure 8. Intuitively, $G \downarrow^1 \ell$ holds whenever it is possible to derive that G can reduce by ℓ, in the relaxed setting where the existentially quantified premise is used in [GR4]. This lets us quantify over the labels for which reduction derivations must exist. This property must also be preserved by reduction, so we additionally assert that the reduced global type G' satisfies the property as well. For recursive global types, a derivation will not be finite, and the predicate is therefore coinductively defined:

$$\frac{}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \downarrow^1 \mathsf{p} \to \mathsf{q} : k\langle U\rangle}\ [\text{GB-1}] \qquad \frac{j \in I}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \downarrow^1 \mathsf{p} \to \mathsf{q} : k\langle l_j\rangle}\ [\text{GB-2}]$$

$$\frac{\mathsf{q} \notin \ell \quad G \downarrow^1 \ell}{\mathsf{p} \to \mathsf{q} : k\langle U\rangle.G \downarrow^1 \ell}\ [\text{GB-3}] \qquad \frac{\mathsf{q} \notin \ell \quad j \in I \quad G_j \downarrow^1 \ell}{\mathsf{p} \to \mathsf{q} : k\{l_i : G_i\}_{i\in I} \downarrow^1 \ell}\ [\text{GB-4}] \qquad \frac{G[\mu \mathbf{t}.G/t] \downarrow^1 \ell}{\mu \mathbf{t}.G \downarrow^1 \ell}\ [\text{GB-5}]$$

Fig. 8. Barb Relation for Global Types

Definition 9 (Unstuck). *We say a global type is unstuck if it satisfies the predicate* $\mathsf{unstuck}\ (G)$*, coinductively defined as:*

$$\frac{\forall \ell.\ G \downarrow^1 \ell \implies \exists G'.G \xrightarrow{\ell} G' \wedge \mathsf{unstuck}\ (G')}{\mathsf{unstuck}\ (G)}$$

We can finally introduce our stricter notion of coherence:

Definition 10 (Coherence). *The coherence of global, local types and decomposed environments, denoted respectively by* $\mathsf{coherent}(G)$*,* $\mathsf{coherent}(L)$ *and* $\mathsf{coherent}(\hat{\Delta}; \hat{\mathcal{Q}})$*, is defined as:*

- $\mathsf{coherent}(G)$ *holds iff* G *is projectable, linear and unstuck.*
- $\mathsf{coherent}(\hat{\Delta})$ *holds iff there exists a coherent* G *whose full projection is* $\hat{\Delta}$*.*
- $\mathsf{coherent}(\hat{\Delta}; \hat{\mathcal{Q}})$ *holds iff there exists a coherent* $\hat{\Delta}'$ *and a function* f *such that* $\mathsf{decomp}_f\ \hat{\Delta}' = \hat{\Delta}; \hat{\mathcal{Q}}$*.*

5 Typing System and Subject Reduction

This section introduces the type system for processes, connecting the concepts we have covered so far. This is expressed through the following typing judgement:

$$\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \mathcal{Q};\ \Delta$$

The intuitive meaning of the judgement is that process P is well-typed with values typed by the unrestricted environment Γ, including shared names which Γ maps to global types. Ongoing sessions are typed by the two linear environments Δ and $\mathcal{Q}$. Process variables are typed by $\mathcal{D}$, and the linear environment $\mathcal{C}$ ensures that each queue address is unique. An entry $\mathsf{s}[k]$ in $\mathcal{C}$ is like a token uniquely assigned to a queue. We use $\mathcal{C}$ to prevent processes where queue addresses are not unique, e.g., $\mathsf{s}[k] :: \emptyset \mid \mathsf{s}[k] :: \emptyset$. Formally, we define these environments as

$$\text{(Unrestricted)} \quad \Gamma ::= \emptyset \mid \Gamma, a : G \mid \Gamma, x : U \qquad \mathcal{D} ::= \emptyset \mid \mathcal{D}, X : (U, \boldsymbol{T})$$

$$\text{(Linear)} \quad \mathcal{C} ::= \emptyset \mid \mathcal{C}, \mathsf{s}[k] \qquad \Delta ::= \emptyset \mid \Delta, s : T \qquad \mathcal{Q} ::= \emptyset \mid \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q$$

The environment $\mathcal{Q}$ associates a queue type with each located session, recording what actions have been performed, while Δ associates a local type with each session channel, indicating what actions remain. $\mathcal{C}$ ensures that all queue addresses

are disjoint. Note that the environments $\mathcal{Q}$ and Δ may specify multiple sessions, while $\hat{\mathcal{Q}}$ and $\hat{\Delta}$ specifies a single session.

Definition 11 (Session Filtering and Lifting). *The filtering of an environment by session identifier* s*, denoted respectively by* $\Delta(\mathsf{s})$ *and* $\mathcal{Q}(\mathsf{s})$*, yields environments* $\hat{\Delta}$ *and* $\hat{\mathcal{Q}}$ *respectively, containing the local types and queue types of that session. This operation is always defined and returns the empty environment when the session is not present.*
We define a lifting operation, denoted respectively by $[\hat{\Delta}]_{\mathsf{s}}$ *and* $[\hat{\mathcal{Q}}]_{\mathsf{s}}$*, which yields environments* Δ *and* $\mathcal{Q}$*, corresponding to the initial environments where roles* p *have been replaced with located sessions* s^{p}*.*

Using session filtering, we can extend the definition of coherence to $\Delta; \mathcal{Q}$.

Definition 12 (Coherence of $\Delta; \mathcal{Q}$). *The coherence of the environment* $\Delta; \mathcal{Q}$*, denoted by* $\mathsf{coherent}(\Delta; \mathcal{Q})$*, holds iff for any session identifier* s*,* $\Delta(\mathsf{s}); \mathcal{Q}(\mathsf{s})$ *is coherent.*
When we, in stating the coherence of $\Delta; \mathcal{Q}$*, want to refer to the underlying coherent environment* Δ'*, whose decomposition yields* $\Delta; \mathcal{Q}$*, we write* $\mathsf{coherent}(\Delta; \mathcal{Q})$ *as* Δ'*.*

The environments Δ, $\mathcal{Q}$ and $\mathcal{C}$ are linear, and this means different things for each. For Δ and $\mathcal{C}$, linearity carries the standard meaning: each entry in the environment must be used exactly once. In other words, entries cannot be duplicated, such that a local type in Δ is used to type two different processes. Linearity of $\mathcal{Q}$, is based on the partitioning of a queue type, which we explain next:

Example 2 (Partioning a queue). Consider two queues with messages from the same role p

$$\mathsf{s}[k] :: \mathsf{true}^{\mathsf{p}} \mid \mathsf{s}[k'] :: \mathsf{true}^{\mathsf{p}}$$

The queue type of p is

$$(k, \mathsf{bool}) \cdot (k', \mathsf{bool})$$

We partition the queue type into two queue types, one for each queue.

$$(k, \mathsf{bool}) \qquad (k', \mathsf{bool})$$

In the following definition, let $\mathsf{filter}\ f\ Q$ denote the queue type obtained by filtering out all those entries in the channels of Q that do not satisfy the predicate $f : k \to \{\mathsf{True}, \mathsf{False}\}$. Partitioning is then defined as:

Definition 13 (Partitioning). *Partitioning of a queue type by a predicate* $f : k \to \mathsf{bool}$*, denoted by* $\mathsf{partition} : (k \to \{\mathsf{True}, \mathsf{False}\}) \to Q \to Q \times Q$*, is defined as:*

$$\mathsf{partition}^f(Q) = (\mathsf{filter}\ f\ Q, \mathsf{filter}\ (\lambda k.\ \neg f(k))\ Q)$$

We write $(Q_0, Q_1) \hookrightarrow Q$ *when there exists a predicate* f *such that* $\mathsf{partition}^f(Q) = (Q_0, Q_1)$ *and we point-wise extend this to environments, abusing notation by writing* $(\mathcal{Q}_0, \mathcal{Q}_1) \hookrightarrow \mathcal{Q}$*.*

$$\frac{\Gamma(x) = S}{\Gamma \vdash x : S} \qquad \Gamma \vdash b : \mathsf{bool} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e' : \mathsf{bool}}{\Gamma \vdash e \text{ and } e' : \mathsf{bool}}$$

Fig. 9. Typing rules for expressions

$$\frac{\Gamma \vdash a : G \quad \begin{array}{l}\mathsf{roles}(G) = \{0, \ldots, \mathsf{n}\} \\ \mathsf{channels}(G) = \{1, \ldots, k\}\end{array} \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, \mathfrak{s} : \mathsf{unf}(G\restriction_{\mathsf{n}});\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} \overline{a}\,[\mathsf{n}]_k(\mathfrak{s}).P \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-MCast}]$$

$$\frac{\Gamma \vdash a : G \quad \mathsf{p} \in \mathsf{roles}(G) \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, \mathfrak{s} : \mathsf{unf}(G\restriction_{\mathsf{p}});\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} a[\mathsf{p}](\mathfrak{s}).P \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-MAcc}]$$

$$\frac{\Gamma \vdash e : U \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]!\langle e\rangle; P \triangleright_{\emptyset} \Delta, s :!k\langle U\rangle.T;\ \mathcal{Q}}\ [\text{T-Send}] \qquad \frac{\Gamma, x : U \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]?(x); P \triangleright_{\emptyset} \Delta, s :?k\langle U\rangle.T;\ \mathcal{Q}}\ [\text{T-Rcv}]$$

$$\frac{T_1 \approx T_2 \quad s \neq s' \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]!\langle\!\langle s'\rangle\!\rangle; P \triangleright_{\emptyset} \Delta, s :!k\langle T_1\rangle.T, s' : T_2;\ \mathcal{Q}}\ [\text{T-Deleg}] \qquad \frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T), t : \mathsf{unf}(T');\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k]?((t)); P \triangleright_{\emptyset} \Delta, s :?k\langle T'\rangle.T;\ \mathcal{Q}}\ [\text{T-SRec}]$$

$$\frac{i \in I \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T_i);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k] \lhd l_i; P \triangleright_{\emptyset} \Delta, s : k \oplus \{l_j : T_j\}_{j\in I};\ \mathcal{Q}}\ [\text{T-Sel}]$$

$$\frac{\forall i \in I \quad \Gamma \vdash_{\mathscr{D}} P_i \triangleright_{\emptyset} \Delta, s : \mathsf{unf}(T_i);\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} s[k] \rhd \{l_j : P_j\}_{j\in I} \triangleright_{\emptyset} \Delta, s : k \,\&\, \{l_j : T_j\}_{j\in I};\ \mathcal{Q}}\ [\text{T-Branch}]$$

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\emptyset} \Delta;\ \mathcal{Q} \quad \Gamma \vdash_{\mathscr{D}} Q \triangleright_{\emptyset} \Delta;\ \mathcal{Q} \quad \Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash_{\mathscr{D}} \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-If}]$$

$$\frac{\mathsf{ends}(\Delta_0) \quad \Gamma \vdash e : U \quad X : (U, \tilde{T}) \in \mathscr{D} \quad \Delta_1 \approx \tilde{s} : \tilde{T}}{\Gamma \vdash_{\mathscr{D}} X\langle e, \tilde{s}\rangle \triangleright_c \Delta_0, \Delta_1;\ \mathcal{Q}}\ [\text{T-Var}] \qquad \frac{\mathsf{ends}(\Delta)}{\Gamma \vdash_{\mathscr{D}} \mathbf{0} \triangleright_{\emptyset} \Delta;\ \mathcal{Q}}\ [\text{T-Inact}]$$

Fig. 10. Typing rules: All rules implicitly include the premise $\mathsf{ends}(\mathcal{Q})$

Terminated process $\mathbf{0}$ and the empty queue $\mathsf{s}[k] :: \emptyset$ are typed using the following definitions for terminated environments:

Definition 14 (Terminated Environments). *A local type environment is terminated, denoted by* $\mathsf{ends}(\Delta)$ *iff* $\forall(\mathsf{p} : T) \in \Delta.\ \mathit{unf}(T) = \mathsf{end}$. *A queue environment is terminated, denoted by* $\mathsf{ends}(\mathcal{Q})$ *iff* $\forall(\mathsf{s}^{\mathsf{p}} : Q) \in \mathcal{Q}.\ Q = \epsilon$

Typing Rules. The rules defining our type system can be found in Figures 9, 10, and 11. Figure 9 contains the typing rules for expressions. The typing rules for processes are split into two parts: the first part is presented in Figure 10, where all rules include the premise $\mathsf{ends}(\mathcal{Q})$, which is omitted in the figure to conserve space. The remaining rules are in Figure 11, where the premises of all rules are explicitly stated. [T-MCast] and [T-MAcc] type session request and accept, respectively. Roles are totally ordered, and the largest role of G is n which is assigned to the requesting process. The unfolding of its projection is

$$\frac{\Gamma \vdash_{\mathscr{D}} P \triangleright_{\mathcal{C}_0} \Delta_0;\ \mathcal{Q}_0 \quad \Gamma \vdash_{\mathscr{D}} Q \triangleright_{\mathcal{C}_1} \Delta_1;\ \mathcal{Q}_1 \quad (\mathcal{Q}_0, \mathcal{Q}_1) \hookrightarrow \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} P \mid Q \triangleright_{\mathcal{C}_0, \mathcal{C}_1} \Delta_0, \Delta_1;\ \mathcal{Q}} \text{ [T-Conc]}$$

$$\frac{\mathsf{ends}(\mathcal{Q}) \quad \mathsf{ends}(\Delta)}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \emptyset \triangleright_{\mathsf{s}[k]} \Delta;\ \mathcal{Q}} \text{ [T-QNil]} \qquad \frac{\mathsf{coherent}(G) \quad \Gamma, \mathsf{a} : G \vdash_{\mathscr{D}} P \triangleright_{C} \Delta;\ \mathcal{Q}}{\Gamma \vdash_{\mathscr{D}} (\boldsymbol{\nu}^{\mathsf{a}}\mathsf{a}) P \triangleright_{\emptyset} \Delta;\ \mathcal{Q}} \text{ [T-NRes]}$$

$$\frac{\mathsf{coherent}(\hat{\Delta}; \hat{\mathcal{Q}}) \quad \Gamma \vdash_{\mathscr{D}} P \triangleright_{(\mathcal{C}, \{\mathsf{s}[k_i]\}_{i \in I})} \Delta, [\hat{\Delta}]_{\mathsf{s}};\ \mathcal{Q}, [\hat{\mathcal{Q}}]_{\mathsf{s}}}{\Gamma \vdash_{\mathscr{D}} (\boldsymbol{\nu}^{\mathsf{s}}\mathsf{s}) P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}} \text{ [T-CRes]}$$

$$\frac{\Gamma \vdash v : U \quad \Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: v^{\mathsf{p}} \cdot \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, U) \cdot Q} \text{ [T-QVal]} \qquad \frac{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: l^{\mathsf{p}} \cdot \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, l) \cdot Q} \text{ [T-QSel]}$$

$$\frac{T_0 \approx T_1 \quad \Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: \tilde{h} \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : Q}{\Gamma \vdash_{\mathscr{D}} \mathsf{s}[k] :: (\mathsf{t}^{\mathsf{q}})^{\mathsf{p}} \cdot \tilde{h} \triangleright_{\mathcal{C}} \Delta, \mathsf{t}^{\mathsf{q}} : T_0;\ \mathcal{Q}, \mathsf{s}^{\mathsf{p}} : (k, T_1) \cdot Q} \text{ [T-QSess]}$$

Fig. 11. Typing rules: There are no implicit premises for these rules

inserted into Δ. [T-Send] and [T-Rcv] are the rules for sending and receiving. The session channel s is mapped to a local type in Δ and checked to correspond with the action of the process. In the typing of the subterm P, the s entry in Δ is updated to the unfolded continuation $\mathsf{unf}\ T$. [T-Deleg] and [T-SRec] type session delegation and reception. Similarly to the rules for send and receive, they perform the delegation of session channel s', which requires the type T_2 for s' in the environment to be coinductively equal to the carried type T_1 in the type of s. Here, coinductive equality is necessary to ensure the type system is closed under $\approx$. The rules [T-Sel] and [T-Branch] are similar to [T-Send] and [T-Rcv]. [T-IF] is the rule for if statements, which has no effect on the type environments. [T-Inact] and [T-QNil] are the rules for inaction and the empty queue, using the predicates $\mathsf{ends}(\Delta)$ and $\mathsf{ends}(\mathcal{Q})$ to check that the respective environments are terminated. [T-Conc] types parallel composition and implements the split of the environments as discussed above. [T-NRes] and [T-CRes] are the rules for name and channel restriction. [T-NRes] introduces a coherent global type into Γ. [T-CRes] adds the decomposed environment $\hat{\Delta}; \hat{\mathcal{Q}}$ into $\Delta; \mathcal{Q}$ and extends $\mathcal{C}$ by a disjoint set of unique tokens $\{\mathsf{s}[k_i]\}_{i \in I}$. The tokens in $\mathcal{C}$ are used to ensure the uniqueness of queue addresses such that if $\mathsf{s}_k \in \mathsf{dom}\ \mathcal{C}$, then exactly one queue of shape $\mathsf{s}[k] :: \tilde{h}$ exists in the typed process. [T-QVal], [T-QSess], and [T-QSel] are type queues. A message is annotated by the role p that sent the message, which is used to identify the queue type in $\mathcal{Q}$ associated with the located session s^{p}, checking that k and $\mathcal{U}$ in the queue type correspond to the channel of the queue and the message it contains. [T-Var] is the rule for procedure call, calling procedure X with an expression and a list of distinct session channels. Here, we write $\tilde{s} : \tilde{T}$ to mean then environment with entries $(\tilde{s}_i : \tilde{T}_i)$ for $i \in \{1, .., |\tilde{s}|\}$, where $|\tilde{s}| = |\tilde{T}|$.

Our semantics uses a fixed set of definitions D in $\rightarrow_D$. We now show how to type these definitions.

Definition 15 (Typing of declarations). *Declarations D are typed by the environment $\mathcal{D}$ whenever* $\mathsf{dom}\ D = \mathsf{dom}\ \mathcal{D}$ *and*

$$\forall X(x\tilde{\jmath}) = P \in D. \quad X : (U, \tilde{T}) \in \mathcal{D} \text{ implies } x : U \vdash_{\mathcal{D}} P \triangleright_{\emptyset} \tilde{\jmath} : \tilde{T};\ \emptyset$$

Example 3 (Typing process definitions). Let D contain the single declaration $X(x\jmath) = \jmath[k]!\langle x\rangle; X\langle x, \jmath\rangle$, which is typed by the environment $X : \mathsf{bool} \times \mathsf{unf}(T)$, where $T = \mu\mathbf{t}.!k\langle\mathsf{bool}\rangle.\mathbf{t}$. We name this environment $\mathcal{D}$ and to assert that D is typed by $\mathcal{D}$, we must show:

$$x : \mathsf{bool} \vdash_{\mathcal{D}} \jmath[k]!\langle x\rangle; X\langle x, \jmath\rangle \triangleright_{\emptyset} \jmath : \mathsf{unf}(T);\ \emptyset$$

Which is derivable by [T-Send] and and [T-Var].

The example above highlights an important aspect of the type system. All local types in Δ are unfolded. Note for example that we put the unfolded local type in $\mathcal{D}$. This is necessary because all rules assume local types are unfolded, and we preserve this property by only introducing unfolded local types into Δ.

Properties of the type system. We now state some properties of the type system, starting with the substitution lemma, which is required by the Subject Reduction Theorem to derive typings for value substituted processes $P[v/x]$ and located session substituted processes $P[\mathsf{s}^{\mathsf{p}}/t]$.

Lemma 1 (Substitution).

1. *If $\Gamma, x : U \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$ and $\Gamma \vdash v : U$ then $\Gamma \vdash_{\mathcal{D}} P[v/x] \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$.*
2. *If $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \Delta, \jmath : T;\ \mathcal{Q}$ then $\Gamma \vdash_{\mathcal{D}} P[\mathsf{s}^{\mathsf{p}}/\jmath] \triangleright_{\mathcal{C}} \Delta, \mathsf{s}^{\mathsf{p}} : T;\ \mathcal{Q}$.*

The lemma states that processes remain well-typed after variables are substituted for values and located session variables are substituted for located session identifiers. This lemma is necessary to prove the cases of subject reduction dealing with value and session reception.

We are now ready to state subject congruence and reduction:

Theorem 3 (Subject congruence and reduction).

1. *If $P \equiv Q$ and $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$ then $\Gamma \vdash_{\mathcal{D}} Q \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$*
2. *If $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\mathcal{C}} \Delta;\ \mathcal{Q}$ and $\mathsf{coherent}(\Delta; \mathcal{Q})$ as Δ_0 and $P \rightarrow_D P'$ then there exists $\Delta_1, \Delta', \mathcal{Q}'$ s.t $\mathsf{coherent}(\Delta'; \mathcal{Q}')$ as Δ_1 and $\Gamma \vdash_{\mathcal{D}} P' \triangleright_{\mathcal{C}} \Delta';\ \mathcal{Q}'$ and either $\Delta_0 = \Delta_1$ or there exists ℓ s.t. $\Delta_0 \xrightarrow{\ell} \Delta_1$*
3. *If $\Gamma \vdash_{\mathcal{D}} P \triangleright_{\emptyset} \emptyset;\ \emptyset$ and $P \rightarrow_D P'$ then $\Gamma \vdash_{\mathcal{D}} P' \triangleright_{\emptyset} \emptyset;\ \emptyset$*

The theorem contains three statements: (1) typing is preserved by congruence; (2) typing is preserved by process reductions, relating environments either by equality or reduction; and, (3) typing is preserved by closed processes with no queues, which is a special case of (2). The proof of (1) is by structural induction on the congruence relation, while the proof of (2), the core subject reduction result, is by induction on process reductions.

Remark 5 (Differences with the original work). The type system enforces linear resource use by defining smaller resources as splits of larger ones, contrasting with Honda et al., who define larger resources as compositions of smaller ones. Our focus on splitting rather than composing is evident in several aspects of the type system. Firstly, the environment $\Delta; \mathcal{Q}$ provides an intuitive temporal split: queue types in $\mathcal{Q}$ indicate completed actions, while residual local types in Δ indicate remaining actions. Keeping these concepts distinct simplifies the definition of environment splitting, as Δ and $\mathcal{Q}$ split orthogonally. Secondly, queue types and residual local types are obtained by decomposition. Our definition allows us to avoid using the subtyping relation, unlike Honda et al.. This is significant because introducing subtyping through a non-structural subsumption rule complicates reasoning about type derivations, particularly in proof assistants.

6 Related Work and Discussion

Mechanisation of multiparty session types. Castro-Perez et al. [10] mechanise in Coq a trace equivalence between processes, coinductive local types and coinductive global types. Their processes are single-session without delegation, therefore there is no multi-cast session request nor session acceptance and a process is always typed by a single global type. Their process language is a domain specific language embedded in Coq. This allows a user to write a correct-by-construction process in Coq for each role of a session, and use Coq's extraction feature to extract executable OCaml code. The generated code has a transport API that allows interaction with external code. Although their setting is more applied than ours, they also propose an elegant approach to representing session types as coinductive types that are coinductively related to unfolded μ-types. Castro-Perez et al. prove their definition of projection sound with respect to a coinductive specification of projection by Ghilezan et a. [20]. For this same specification, Tirore et al. [46] propose a projection for which both soundness and completeness holds. This is the projection that we adopted in this paper.

Jacobs et al. [31] mechanise multiparty GV in Coq using the Iris framework [33]. Multiparty GV is an extension of the linear lambda calculus with multiparty asynchronous session types. Their language features an n-ary fork that combines session initiation and process spawning, ensuring an acyclic topology. This is unlike our setting where already existing processes initiate sessions on shared names. Their methodology is significantly different from ours, and is based on separation logic [41, 42] and connectivity graphs [30].

Mechanisation of binary session types. Using the Iris framework, Hinrichsen et al. [25] introduce semantic typing for a variant of binary session types using logical relations. Gay et al. [19] compare duality definitions in an Agda mechanisation of binary session types. Duality is the binary notion of compatibility used for binary session types, that is a special case of the multiparty notion of coherence. Gay et al. demonstrate that some definitions are unsound when type variables can appear in messages. Like them, our work does not take the equi-recursive view and we explicitly state when syntactic equality of μ-types is

used and thus when to trigger unfoldings and when we use coinductive equality modulus unfolding. Castro-Perez et al. [11] mechanise in Coq a subject reduction theorem for the binary session type meta-theory given by Yoshida and Vasconcelos [49]. They use the Locally Nameless representation for variables by Charguéraud [12]. Like us, Castro-Perez et al. distinguish between variable and value session channels. Unlike our setting, their session types do neither include recursion nor process call and definitions. Instead of process call they use replication $!P$. Recently, Sano et al. [44] mechanised a version of binary session types that are in a Curry-Howard correspondence with linear logic [47]. They use linearity predicates, which enable the separate checking of linearity properties in the type system, allowing the typing judgement to focus solely on non-linear contexts.
Session types on paper. There is a vast literature on multiparty session types that builds on the original paper. An introduction was given by Yoshida and Gheri [48] and a comprehensive overview was given by Scalas and Yoshida [45] where they argue the inconvenience of using global types to define compatibility. It should be noted that to the best of our knowledge, all later works use implicit channels. The counter example therefore does not invalidate these results. However, we argue that explicit channels are important in a setting where there is a limited number of resources available that must be shared in a session, e.g., AMQP [3].

Future work and discussion. Subject reduction is used to prove session fidelity and type safety, the former expressing that communication within a session will progress as specified by the global type, and the latter expressing that well typed programs do not go wrong in the standard sense. Future work includes proving these properties. Another important result of the original paper is progress, which states that a well-typed single-session process never deadlocks. It would be interesting to investigate the implications of the counterexample on progress as well as proving progress for our meta-theory.

The counterexample highlighted the restrictiveness of the global type semantics. Later work on multiparty session types [45, 10] introduces a more flexible global type semantics that, in the context of implicit channels, is unrelated to the counterexample. This work suggests semantic definitions that could allow the unstuck predicate to be removed from the meta-theory.

The mechanisation revealed some inconveniences in the meta-theory. For example, the representation of a finished session channel is ambiguous; it can be represented either as not being contained in the environment or as being contained but mapped to end. We conjecture that it would simplify proofs to resolve this ambiguity by choosing only the non-presence representation in conjunction with a closure operation on sessions, as done by Caires et al. [7] and Wadler [47].

Another inconvenience is the use of global types as the basis for multiparty compatibility. Scalas and Yoshida [45] pointed out that more general notions of compatibility exist beyond global types. This broader perspective may not only provide a more encompassing definition of compatibility but also simplify proofs by eliminating the need for two specification languages in the meta-theory.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Scribble: Describing Multy Party Protocols, http://www.scribble.org
2. Session Types in Programming Languages: A Collection of Implementations, http://groups.inf.ed.ac.uk/abcd/session-implementations.html
3. Advanced Message Queuing Protocol. http://www.iona.com/opensource/amqp/ (2015)
4. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5201, pp. 418–433. Springer (2008). https://doi.org/10.1007/978-3-540-85361-9_33, https://doi.org/10.1007/978-3-540-85361-9_33
5. Boldo, S., Jourdan, J., Leroy, X., Melquiond, G.: A formally-verified C compiler supporting floating-point arithmetic. In: Nannarelli, A., Seidel, P., Tang, P.T.P. (eds.) 21st IEEE Symposium on Computer Arithmetic. pp. 107–115. IEEE Computer Society (2013). https://doi.org/10.1109/ARITH.2013.30
6. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of asynchronous session subtyping. Inf. Comput. **256**, 300–320 (2017). https://doi.org/10.1016/J.IC.2017.07.010, https://doi.org/10.1016/j.ic.2017.07.010
7. Caires, L., Pfenning, F., Toninho, B.: Linear logic propositions as session types. Math. Struct. Comput. Sci. **26**(3), 367–423 (2016). https://doi.org/10.1017/S0960129514000218, https://doi.org/10.1017/S0960129514000218
8. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: Nicola, R.D. (ed.) Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4421, pp. 2–17. Springer (2007). https://doi.org/10.1007/978-3-540-71316-6_2, https://doi.org/10.1007/978-3-540-71316-6_2
9. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: A logical explanation of multiparty session types. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.33, https://doi.org/10.4230/LIPIcs.CONCUR.2016.33
10. Castro-Perez, D., Ferreira, F., Gheri, L., Yoshida, N.: Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In: Proceedings of PLDI. pp. 237–251. ACM (2021). https://doi.org/10.1145/3453483.3454041
11. Castro-Perez, D., Ferreira, F., Yoshida, N.: EMTST: engineering the meta-theory of session types. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12079, pp. 278–285.

Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_17, https://doi.org/10.1007/978-3-030-45237-7_17
12. Charguéraud, A.: The locally nameless representation. J. Autom. Reason. **49**(3), 363–408 (2012). https://doi.org/10.1007/S10817-011-9225-2, https://doi.org/10.1007/s10817-011-9225-2
13. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. MSCS **760**, 1–65 (2015)
14. Danielsson, N.A., Altenkirch, T.: Subtyping, declaratively. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6120, pp. 100–118. Springer (2010). https://doi.org/10.1007/978-3-642-13321-3_8, https://doi.org/10.1007/978-3-642-13321-3_8
15. Deniélou, P., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7211, pp. 194–213. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_10, https://doi.org/10.1007/978-3-642-28869-2_10
16. Fowler, S.: An erlang implementation of multiparty session actors. In: Bartoletti, M., Henrio, L., Knight, S., Vieira, H.T. (eds.) Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016. EPTCS, vol. 223, pp. 36–50 (2016). https://doi.org/10.4204/EPTCS.223.3, https://doi.org/10.4204/EPTCS.223.3
17. Gay, S.J., Hole, M.: Types and subtypes for client-server interactions. In: Swierstra, S.D. (ed.) Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1576, pp. 74–90. Springer (1999). https://doi.org/10.1007/3-540-49099-X_6, https://doi.org/10.1007/3-540-49099-X_6
18. Gay, S.J., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**(2-3), 191–225 (2005). https://doi.org/10.1007/S00236-005-0177-Z, https://doi.org/10.1007/s00236-005-0177-z
19. Gay, S.J., Thiemann, P., Vasconcelos, V.T.: Duality of session types: The final cut. In: Balzer, S., Padovani, L. (eds.) Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020. EPTCS, vol. 314, pp. 23–33 (2020). https://doi.org/10.4204/EPTCS.314.3, https://doi.org/10.4204/EPTCS.314.3
20. Ghilezan, S., Jaksic, S., Pantovic, J., Scalas, A., Yoshida, N.: Precise subtyping for synchronous multiparty sessions. Journal of Logical and Algebraic Methods in Programming **104**, 127–173 (2019). https://doi.org/10.1016/j.jlamp.2018.12.002
21. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: Kapur, D. (ed.) Computer Mathematics, 8th Asian Symposium, ASCM. Lecture Notes in Computer Science, vol. 5081, p. 333. Springer (2007). https://doi.org/10.1007/978-3-540-87827-8_28
22. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S.L., Mahboubi, A., O'Connor, R., Biha, S.O., Pasca, I., Rideau, L., Solovyev,

A., Tassi, E., Théry, L.: A machine-checked proof of the odd order theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving - 4th International Conference, ITP. LNCS, vol. 7998, pp. 163–179. Springer (2013). https://doi.org/10.1007/978-3-642-39634-2_14
23. Hales, T.C., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Hoang, T.L., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J.M., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the kepler conjecture. CoRR **abs/1501.02155** (2015), http://arxiv.org/abs/1501.02155
24. Han, J.M., van Doorn, F.: A formal proof of the independence of the continuum hypothesis. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 353–366. ACM (2020). https://doi.org/10.1145/3372885.3373826
25. Hinrichsen, J.K., Louwrink, D., Krebbers, R., Bengtson, J.: Machine-checked semantic session typing. In: Hritcu, C., Popescu, A. (eds.) CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021. pp. 178–198. ACM (2021). https://doi.org/10.1145/3437992.3439914, https://doi.org/10.1145/3437992.3439914
26. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings. Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer (1993). https://doi.org/10.1007/3-540-57208-2_35, https://doi.org/10.1007/3-540-57208-2_35
27. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Proceedings of ESOP. LNCS, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
28. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proceedings of POPL. pp. 273–284. ACM (2008). https://doi.org/10.1145/1328438.1328472
29. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. Journal of the ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
30. Jacobs, J., Balzer, S., Krebbers, R.: Connectivity graphs: a method for proving deadlock freedom based on separation logic. Proc. ACM Program. Lang. **6**(POPL), 1–33 (2022). https://doi.org/10.1145/3498662, https://doi.org/10.1145/3498662
31. Jacobs, J., Balzer, S., Krebbers, R.: Multiparty GV: functional multiparty session types with certified deadlock freedom. Proceedings of the ACM on Programming Languages **6**(ICFP), 466–495 (2022). https://doi.org/10.1145/3547638
32. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D.A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an os kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proceedings of the 22nd ACM Symposium on Operating Systems Principles. pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596
33. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 205–217. ACM (2017). https://doi.org/10.1145/3009837.3009855, https://doi.org/10.1145/3009837.3009855

34. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 1137–1148. ACM (2018)
35. Lange, J., Yoshida, N.: On the undecidability of asynchronous session subtyping. In: Esparza, J., Murawski, A.S. (eds.) Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10203, pp. 441–457 (2017). https://doi.org/10.1007/978-3-662-54458-7_26, https://doi.org/10.1007/978-3-662-54458-7_26
36. SIPLAN selection committee: Most Influential POPL Paper Award (2018). https://www.sigplan.org/Awards, accessed: July 2024
37. The Coq development team: The Coq Proof Assistant. https://coq.inria.fr, accessed: July 2024
38. The Lean development team: The Lean Proof Assistant. https://lean-lang.org, accessed: July 2024
39. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. Information and Computation **100**(1), 1–40,41–77 (Sep 1992)
40. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9, https://doi.org/10.1007/3-540-45949-9
41. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symb. Log. **5**(2), 215–244 (1999). https://doi.org/10.2307/421090, https://doi.org/10.2307/421090
42. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001). https://doi.org/10.1007/3-540-44802-0_1, https://doi.org/10.1007/3-540-44802-0_1
43. Pierce, B.C.: Types and programming languages. MIT Press (2002)
44. Sano, C., Kavanagh, R., Pientka, B.: Mechanizing session-types using a structural view: Enforcing linearity without linearity. Proc. ACM Program. Lang. **7**(OOPSLA2), 374–399 (2023). https://doi.org/10.1145/3622810, https://doi.org/10.1145/3622810
45. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019). https://doi.org/10.1145/3290343, https://doi.org/10.1145/3290343
46. Tirore, D.L., Bengtson, J., Carbone, M.: A sound and complete projection for global types. In: Naumowicz, A., Thiemann, R. (eds.) 14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland. LIPIcs, vol. 268, pp. 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ITP.2023.28, https://doi.org/10.4230/LIPIcs.ITP.2023.28
47. Wadler, P.: Propositions as sessions. In: Proceedings of ICFP. pp. 273–286. ACM (2012). https://doi.org/10.1145/2364527.2364568

48. Yoshida, N., Gheri, L.: A very gentle introduction to multiparty session types. In: Hung, D.V., D'Souza, M. (eds.) Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings. Lecture Notes in Computer Science, vol. 11969, pp. 73–93. Springer (2020). https://doi.org/10.1007/978-3-030-36987-3_5, https://doi.org/10.1007/978-3-030-36987-3_5
49. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. In: Fernández, M., Kirchner, C. (eds.) Proceedings of the First International Workshop on Security and Rewriting Techniques, SecReT@ICALP 2006, Venice, Italy, July 15, 2006. Electronic Notes in Theoretical Computer Science, vol. 171, pp. 73–93. Elsevier (2006). https://doi.org/10.1016/J.ENTCS.2007.02.056, https://doi.org/10.1016/j.entcs.2007.02.056