IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Refining Multiparty Session Types

Fangyi Zhou

February 2024

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing at Imperial College London.

Statement of Originality

I declare that this thesis is my own work, and I have given appropriate reference and attribution to work of others in this thesis.

Fangyi Zhou

Abstract

A distributed system consists of various components working in coordination, usually following a specified protocol. *Multiparty session types (MPST)* are a typing discipline for distributed programming with message passing concurrency. The typing discipline provides a way to describe communication protocols as global types governing the overall communication structure. From global types, local types are derived for describing the behaviour of an individual component. This top-down design methodology provides a way to specify and implement multiparty communication protocols correctly.

Communication protocols often contain constraints on payload data. While the original MPST theory provides support for basic payload types and ensures the absence of payload type mismatches, there is no way to capture data constraints using basic types. To address this limitation, we propose to incorporate *refinement types* into multiparty session types. Refinement types provide a lightweight way to describe and verify properties on data, and integrating refinement types into MPST allows properties and constraints on data to be specified and verified.

In this thesis, we introduce a theory of refined multiparty session types, and a toolchain, SESSION*, for implementing refined protocols. On the theory side, we show that our extended theory retains desirable properties of MPST while improving expressivity. On the practical side, we target the F* programming language, and demonstrate a code generation approach that guarantees safety by construction.

Acknowledgements

This PhD has been a long and tough journey, especially in the light of the COVID-19 pandemic. I'm tremendously grateful to be supported by a number of wonderful people throughout this journey. I wouldn't be able to complete this PhD without you.

This PhD is supported by EPSRC grants EP/K034413/1 and EP/N027833/1. I would like to thank the EPSRC for the generous funding support.

I would like to first thank my supervisor Prof. Nobuko Yoshida for her advice and guidance. I would also like to thank my two second supervisors Prof. Sophia Drossopoulou, and Dr. Nicolas Wu for their advice and support, especially during difficult times. I would like to thank my examiners Prof. Dr. Peter Thiemann and Dr. Sergio Maffeis for the fruitful discussions during the viva, and for their valuable feedback and suggestions to improve this thesis. I would like to thank the departmental PhD administrator Dr. Amani El-Kholy for being always available to help during the entire PhD. In addition, I would like to thank Prof. Paul Kelly, my personal tutor during undergraduate times, who continued to support, encourage, and inspire me, since the beginning of my undergraduate time at Imperial, all way to the end of my PhD.

I would like to thank SIGPLAN-M, a mentorship programme set up by Talia Ringer, and my mentor, Stavros Aronis. The programme has been very helpful for me as a junior researcher, and I feel more connected to the community.

I would like to thank people who worked with me, collaborated with me, shared an office with me, or joined regularly at lunch or afternoon breaks: Adam, Alceste, Ameen, Anson, Assel, Benito, Chengsong, David, Domenico, Elena, Eva, Francesco, Francisco, Hailey, Julia, Keigo, Litao, Lorenzo, Luke, Martin, Nicholas, Nicolas, Panagiotis, Ping, Raymond, Rumyana, Sacha, Samuel, Shahin, Simon, Stephen, Sung-Shik, Uma, Vesna, Wen, Yaoda, and Zak. You made this journey fun and exciting!

I would like to thank my wonderful colleagues at my new job, for supporting me through the final months of thesis writing: Daniel, Evan, Franco, Horia, Ilina, Ioannis, Pauline, Sandro, Stefan, and Vlad. I wouldn't be able to finish my thesis in time without your encouragement and help, especially Franco Raimondi's careful proofreading and valuable advice!

I would like to thank my therapist Ellis Morgan who taught me the lesson to be assertive, which has turned out to be crucial throughout this journey.

I would like to thank my friends: Ashley, Husheng, Junfeng, Lidan, Lun, Martin, Peter, Rachel, Rob, Rory (Qianyi), Tianyi, Vincenzo, Wei, Wenjie, Yicheng, Yolanda, Yvette, and Zifan. Thanks for listening to my monologues about my PhD, and I wouldn't be able to make it to the end without you!

Finally, I would like to thank my family for their unconditional support and love.

Contents

List of Figures

List of Tables

List of Theorems

The Biggest Problem in Communication Is the Illusion That It Has Taken Place.

William H. Whyte (probably[1])

[1]See `https://quoteinvestigator.com/2014/08/31/illusion/`.

1 Introduction

1.1 Motivation and Goal

Starting from the beginning of the 21st century, computing becomes an important part of daily life. Computing takes place in a wide variety of locations, ranging from personal devices such as mobile phones and smart wearables, to large scale data centres and super computers. Moreover, computing often takes place across locations in *coordination*: e.g. a weather service may process complex weather models in a data centre, and then deliver the weather forecast, the output from the models, to personal devices when requested.

This coordinative approach to computing reflects two interconnected concepts in modern computing: *distributed* and *concurrent* computing. Distributed computing focuses on interactions between various components (which may be located at difference places), and how the entire system may still function in spite of failures of some components. Concurrent computing focuses on performing computations at an overlapping time period, where computations may involve communications between themselves. Both concepts are widely applied for building robust and scalable software, and the modern era of computing has benefited much from advancements in both areas.

Together with their benefits, distributed and concurrent computing also bring challenges, of which *correctness* is a big one. Traditional techniques in (sequential) program verification may not be directly applicable for concurrent programs, because concurrency may introduce non-determinism, race conditions, or simply so many possible interleavings to explore to a point where verification ceases to be feasible (also known as the state explosion problem).

Researchers tackle the challenges in different ways. Some choose to extend existing sequential program verification techniques to account for concurrency, e.g. separation logic [Rey02] and *concurrent* separation logic [Bro07], Kleene algebra with tests [Koz97] and *concurrent* Kleene algebra with tests [JM16]. Some choose to introduce paradigms or techniques for concurrent programming, known as *structured concurrency*, so that the behaviour of concurrent programs can be more easily reasoned about. This approach is adopted in many modern programming languages. Notable examples are `async` and `await` keywords, allowing programs to be executed in an asynchronous and non-blocking way. They appear in a number of modern programming languages: C#, F#, JavaScript, Python, Rust, and many other[1].

One of the techniques of (concurrent) program verification is *type systems*. Type systems come in all shapes and forms with different guarantees, but their lightweight nature makes

[1]See `https://en.wikipedia.org/wiki/Async/await` for a non-exhaustive list of programming languages with support for `async` and `await`.

them popular and easy to adopt. In short, type systems assign *types* to *terms*. Types act like a classification of the terms according to the nature of the terms, e.g. functions are distinguished from numeric values due to their difference in applicability. The main benefit of type systems is best summarised in a slogan by Milner [Mil78]:

> Well-typed programs cannot 'go wrong'.
>
> (Milner [Mil78, p. 348])

When applying type systems in concurrency, the concept of *behavioural types* [ABB+16] is of great relevance: types are assigned to reflect the *observable behaviour* of the term, whatever the behaviour of interest may be. For example, in the context of message passing programs, we are particularly interested in the *communication* behaviour: how a process sends and receives messages can be reflected by their behaviour types.

Notably, the typing discipline of *session types*, proposed first in the seminal work by Honda et al. [HVK98], is a stepping stone for applying type system techniques for structured concurrency. Processes communicate by exchanging messages, and session types describe behaviours of communication channels used by those processes. Session type systems do not only ensure the absence of communication errors, but also guarantee (under certain conditions) deadlock freedom and liveness.

This thesis concerns a methodology for safe distributed programming with multiparty communication, using structured concurrency, via a behavioural type system known as *multiparty session types*. In this design methodology, a programmer first describes the overall communication behaviour of a system in a structured way, using *global types*. For the entire system to function according to the description, each participant in the system must act accordingly, following their respective *local types*, which are derived from the prescribed global type. Similarly, multiparty session types provide desirable safety and liveness guarantees. Moreover, the theory is applied, and proved to be useful, in the context of distributed and concurrent programming, as demonstrated by researchers [DHH+15; SDH+17; VCE+18; CHJ+19; VHE+21]. We provide a technical background of multiparty session types in § 2.

As multiparty session types are being applied into distributed and concurrent programming, the theory has shown potential for further extension. In practical programming scenarios, not all communication protocols can be expressed as global types, sometimes due to a lack of expressivity in the original multiparty session type theories. In particular, we are concerned about the limitation that *constraints* on the data exchanged in the protocols cannot accurately represented. For instance, we can specify in the protocol that the type of the payload data may be one of `int`, `string`, or `bool`, yet there is no way to represent constraints such as the `int` must be non-negative, the `string` must be of length between 6 and 18, or the `bool` must reflect the result of some validation. Moreover, these constraints may involve multiple messages, e.g. the `int` payloads must be increasing over time, to be used as a sequence number.

The goal of this research is to extend the existing multiparty session type theory and practice, so that the aforementioned limitation on data constraints can be overcome. Fortunately, a lightweight approach of predicating data types with constraints exists in sequential program verification, known as *refinement types*, initially proposed by Freeman and Pfenning [FP91]. We achieve this goal by incorporating this technique into multiparty session types, and create an extension that benefits from the advantages of both.

1.2 Contributions

refine (*verb*)

1. to make something pure or improve something, especially by removing unwanted material.
2. to improve an idea, method, system, etc. by making small changes.

(Cambridge Dictionary[2])

The title of this thesis is 'Refining Multiparty Session Types', and we invite readers to interpret the word 'refine' liberally. On one hand, we present a development of *refined* multiparty session types, which incorporates *refinement* types into multiparty session types. On the other hand, this new theory improves the expressivity of the typing discipline, and is thus a *refinement* of the original theory in a general sense.

In this thesis, we develop the theory and practice of *refined* multiparty session types, an extension of the original multiparty session types. This extension allows the communication protocols to be described and implemented in a 'refined' manner, with the ability to specify and validate data and control flow constraints in communication protocols, which would not be possible using the original theory.

Overview

We use a flavour of refinement types to allow predicates to be attached on data values [RKJ08]. For example, for an integer type `int`, a *refined* type may require an inhabitant x : `int` to be non-negative $x \geq 0$, to form a refined type $x : \texttt{int}\{x \geq 0\}$. By doing so, the set of inhabited values by the refined type is a subset of all possible values, and hence this is a *refinement*.

Moreover, we can describe predicates involving other data values: suppose we already have a non-negative number x of the aforementioned refined type, we can describe numbers greater than that number using the refined type $y : \texttt{int}\{y > x\}$. This empowers us to describe properties that may appear in communication protocols across different messages.

We present the theoretical developments in § 3, where we give the syntax and semantics of refined multiparty session types, incorporating refinement types that express constraints on the

[2]https://dictionary.cambridge.org/dictionary/english/refine

payload data. As mentioned, the refinement types are simple yet expressive, and allow us to specify constraints not only on a single message, but also across different messages. Following a top-down methodology, we show that global types and projected local types are related; and thus the guarantees from the original multiparty session type theory, such as safety and deadlock-freedom, are preserved.

We present the practical developments in § 4, where we show how refined multiparty protocols can be implemented. We demonstrate a toolchain, named SESSION*, to generate code from a refined multiparty protocol in SCRIBBLE. We choose F* [SHK+16] to be our target language, in order to utilise its refinement type system as well as its effect system. The refinement type system in F* utilises a Satisfiability Modulo Theories (SMT) solver, which discharges proof obligations on the refinements automatically, without the need for developers to manually construct proofs themselves.

Finally, we present a toolchain for multiparty protocol, named νSCR[3], in § 5. The toolchain, written in OCAML, is designed to be extensible by multiparty session type researchers, with the goal to allow them to implement their extensions as prototypes easily. As a case study, we demonstrate how to extend νSCR to add the refinement type extensions, to implement the theory in § 3. So far, we are glad to see νSCR being used in some recent works [GLS+22; CY23], and we encourage researchers to implement more extensions in νSCR.

1.2.1 Publications

Publications produced during this PhD are listed below, with a short explanation for each publication on the contribution made by the thesis author.

Publications with Corresponding Parts in This Thesis

[ZFH+20] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova and Nobuko Yoshida. 'Statically Verified Refinements for Multiparty Protocols'. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: `10.1145/3428216`

Thesis Author's Contribution: This paper presents theoretical and practical developments of refined multiparty session types. The theoretical development was thesis author's own work, with supervision and advice from co-authors. The SCRIBBLE extension was largely Hu's work, to which the thesis author also made contributions. The design of the generated code and the implementation of the code generator were largely the thesis author's own work, with supervision and advice from co-authors. §§ 3–7 of the paper were based on the thesis author's own writing with editing from co-authors; §§ 1–2 were based on drafts from co-authors, and the thesis author contributed by editing.

[3] ν is taken from Greek alphabet (nu), thus νSCR is alternatively written as NUSCR. The inspiration for this name comes from the ν binder (scope restriction) for communication channels in π-calculus.

Corresponding Parts: § 3 of this thesis is based on the theoretical development of this paper, but contains further improvements in expressivity. § 4 of this thesis is based on the practical development of this paper.

[YZF21] Nobuko Yoshida, Fangyi Zhou and Francisco Ferreira. 'Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types'. In: *Fundamentals of Computation Theory*. Ed. by Evripidis Bampis and Aris Pagourtzis. Cham: Springer International Publishing, 2021, pp. 18–35. ISBN: 978-3-030-86593-1. DOI: `10.1007/978-3-030-86593-1_2`

Thesis Author's Contribution: This paper summarises the connection between session types and communicating finite state machines, and presents a new multiparty session type toolchain νSCR. Ferreira and the thesis author designed the toolchain, and the implementation of the toolchain was largely the thesis author's own work, with assistance from Ferreira. §§ 2–5 of the paper were based on the thesis author's own writing with editing from co-authors.

Corresponding Parts: § 5 of this thesis describes the toolchain νSCR, and is based on the material from this paper.

Publications without Corresponding Parts in This Thesis

[MFY+21] Anson Miu, Francisco Ferreira, Nobuko Yoshida and Fangyi Zhou. 'Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types'. In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. CC 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 94–106. ISBN: 9781450383257. DOI: `10.1145/3446804.3446854`

Thesis Author's Contribution: This paper presents a theory of routed multiparty session types, and an toolchain for communication-safe web applications in TYPESCRIPT. The thesis author contributed to the theoretical and practical developments along with other co-authors, but majority of work was done by Miu. §§ 1–2 of the paper were based on the thesis author's own writing with editing from co-authors; the rest of the paper were based on drafts from co-authors, and the thesis author contributed by editing.

[BSY+22] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida and Fangyi Zhou. 'Generalised Multiparty Session Types with Crash-Stop Failures'. In: *33rd International Conference on Concurrency Theory (CONCUR 2022)*. Ed. by Bartek Klin, Sławomir Lasota and Anca Muscholl. Vol. 243. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 35:1–35:25. ISBN: 978-3-95977-246-4. DOI: `10.4230/LIPIcs.CONCUR.2022.35`

Thesis Author's Contribution: This paper presents a generalised multiparty session type system with crash-stop failures. The thesis author contributed to the development of the theory and prototype implementation, but much of the work were done by Barwell and Scalas. § 1 was based on the thesis author's own writing with editing from co-authors; the rest of the paper were based on drafts from co-authors, and the thesis author contributed by editing.

[BHY+23] Adam D. Barwell, Ping Hou, Nobuko Yoshida and Fangyi Zhou. 'Designing Asynchronous Multiparty Protocols with Crash-Stop Failures'. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 1:1–1:30. ISBN: 978-3-95977-281-5. DOI: `10.4230/LIPIcs.ECOOP.2023.1`. URL: `https://drops.dagstuhl.de/opus/volltexte/2023/18194`

Thesis Author's Contribution: This paper presents a multiparty session type system with crash-stop failures, using the traditional top-down design approach; as well as a prototype implementation to implement them in SCALA. The theory was jointly developed by Hou and the thesis author. §§ 1, 3–5 were based on the thesis author's own writing with editing from co-authors; the rest of the paper were based on drafts from co-authors, and the thesis author contributed by editing.

1.3 Structure of the Thesis

§ 2 gives a brief overview of the development of the theory of multiparty session types (MPST) from the literature, on which the rest of the thesis is based.

§ 3 introduces a theory of *refined* multiparty session types (RMPST). We integrate refinement types into the multiparty session type theory, and show how they can be used to specify multiparty protocols with data and control flow constraints.

§ 4 describes how to implement refined multiparty protocols. We introduce a toolchain SESSION*, for implementing multiparty protocols with refinements specified using the SCRIBBLE description language in the F* programming language.

§ 5 introduces an extensible toolchain, νSCR, for multiparty protocols. νSCR is designed for researchers of MPST to implement their extensions easily, and we show how refinement type extension is implemented in νSCR.

§ 6 concludes this thesis, where we summarise the contributions and propose future work.

1.4 Funding Acknowledgements

This PhD is funded by the following grants from the Engineering and Physical Sciences Research Council (EPSRC):

- From Data Types to Session Types—A Basis for Concurrency and Distribution
  Grant Number: EP/K034413/1

- Turtles: Protocol-Based Foundations for Distributed Multiagent Systems

  Grant Number: EP/N027833/1

The thesis author would like to kindly thank the EPSRC for their generous funding support during this PhD.

2 Background: Multiparty Session Types

Multiparty Session Types (MPST) are originally proposed by Honda et al. [HYC08], as a typing discipline for distributed message-passing processes. The typing discipline has been under active development since its initial publication, with simplifications, generalisations, and extensions proposed.

We provide a technical background of a core MPST theory in this chapter, upon which the rest of the thesis builds.

2.1 Intuitions of Session Types

Session types, as originally introduce by Honda et al. [HVK98], provide a way to describe *structured* communication. A *session* describe a single unit of structured communication, e.g. an interaction between a customer and a bank. In this typing discipline, session types are assigned to channels, through which concurrent processes communicate; a session type describes how the channel should be used, usually as a sequence of communication actions (e.g. sending an integer, receiving a string).

Without going into too much details, the theory ensures well-typed processes enjoy desirable properties, which we summarise as *the Session Theorems*: type safety (processes communicate without errors), protocol conformance (also known as *session fidelity*, processes behave according to their types), deadlock-freedom/progress (processes do not get stuck), and liveness (input/output actions eventually succeed).

A key concept in the session type theory is the *duality* of session types, i.e. the dual of a sending action is a receiving action, and vice versa. Intuitively, when two ends of a channel have dual types, the communication shall succeed without failures, nor would there be any deadlocks. However, such guarantee often holds on the condition that there is only a *single* session. When there are more sessions, the interleaving of actions may render progress impossible, i.e. processes are unable to reduce while not reaching completion, e.g. in the case where all processes are in a circular wait.

Suppose a group of n people ($n > 2$) sitting in a circle, where they are only permitted to listen to the person to their left, and speak to the person to their right. This situation can be modelled by n processes P_i, and n sessions s_i ($i \in \{1..n\}$). For a process P_i, it first receives via s_i, and then sends via s_{i+1} (with modular arithmetic). When we view the usage of a channel from each end, i.e. the session type of s_i from the view of P_i and P_{i-1}, they are indeed dual of each other, since one side is sending and the other is receiving. However, when we view the processes composing together, there is *no* progress, since everyone is waiting for the person on their left to say anything, and no one says anything.

In order to have progress, we can designate a conversation starter, who speaks to the person to their right first, then listens to the person to their left. A special process P'_k (for some $k \in \{1..n\}$) sends via s_{k+1} first, follows by receiving via s_k. This change does not affect duality, since the actions performed on each channel do not change. Yet, we can see that the composition of the special process P'_k and ordinary processes P_i (for $i \in \{1..n\}, i \neq k$) has progress.

When there are more than two communicating processes, it is hard to avoid having more than a single communication channel, which in turns invalidates single-session progress guarantees. The multiparty session type theory is introduced to overcome this deficiency, by effectively expanding the concept of a session to include multiple participants.

A *multiparty* session describes the structured communication between a number of participants, instead of two ends of a channel. The global communication pattern is described by *global types*, from a bird's-eye view. The concept of 'session types', describing the behaviour of a single participant or endpoint, transforms to *local types*. A local type can be obtained via an operation known as *projection* from a global type, following a top-down design methodology.

In the rest of this background chapter, we give the syntax and semantics of global and local types, and show how they are related via projection.

2.2 Syntax of Global and Local Types

Many forms of syntax have been proposed for multiparty session types, with differences in their expressivity. In this chapter, we present a syntax in a standard style used in many recent works [SY19; CFG+21]. We base the main development of this thesis on this syntax.

Definition 2.1 (Global and Local Type Syntax)**.** Let *base types* (also known as *sorts*) be ranged over by $\mathtt{S}$, which characterise the values being transmitted in messages. We use G, G' to range over *global* types, and L, L' to range over *local* types, given by the following syntax:

$$\begin{array}{rclll}
\mathtt{S} & ::= & \mathtt{int} \mid \mathtt{bool} \mid \ldots & & \text{Base Types} \\
G & ::= & & & \text{Global Types} \\
 & \mid & \mathbf{p} \to \mathbf{q}\{\ell_i(\mathtt{S}_i).G'_i\}_{i \in I} & & \text{Message} \\
 & \mid & \mu\mathbf{t}.G' & & \text{Recursion} \\
 & \mid & \mathbf{t} \mid \mathtt{end} & & \text{Type Var., End} \\
L & ::= & & & \text{Local Types} \\
 & \mid & \mathbf{p}\&\{\ell_i(\mathtt{S}_i).L'_i\}_{i \in I} & & \text{Receiving} \\
 & \mid & \mathbf{p}\oplus\{\ell_i(\mathtt{S}_i).L'_i\}_{i \in I} & & \text{Sending} \\
 & \mid & \mu\mathbf{t}.L' & & \text{Recursion} \\
 & \mid & \mathbf{t} \mid \mathtt{end} & & \text{Type Var., End}
\end{array}$$

where $\mathbf{p}, \mathbf{q}$ are *participants* (also called *roles*) taken from a fixed set; ℓ are *message labels* taken from a fixed set. ⌟

We first discuss the constructors other than recursion and type variable:

- a transmission from a participant $\mathbf{p}$ to another participant $\mathbf{q}$ (where $\mathbf{p} \neq \mathbf{q}$) is presented in a global type as $\mathbf{p} \to \mathbf{q}\{\ell_i(S_i).G'_i\}_{i \in I}$, where a non-empty, finite set of message labels ℓ_i are available, each carrying a payload of base type S_i. The labels must be mutually distinct ($\ell_i = \ell_j$ iff $i = j$), and the global type continues as the corresponding continuation G'_i. When there is only a single choice (i.e. $|I| = 1$), we omit the curly braces and write the global type as $\mathbf{p} \to \mathbf{q} : \ell(S).G'$.

- a sending action towards a participant $\mathbf{p}$ is presented in a local type as $\mathbf{p}\oplus\{\ell_i(S_i).L'_i\}_{i \in I}$; and a receiving action from a participant $\mathbf{p}$ is presented in a local type as $\mathbf{p}\&\{\ell_i(S_i).L'_i\}_{i \in I}$, where similar restrictions on labels apply. When there is only a single choice (i.e. $|I| = 1$), we omit the curly braces and write the local type as $\mathbf{p}\oplus\ell(S).L'$ or $\mathbf{p}\&\ell(S).L'$.

- a terminated global type is presented as $\texttt{end}$, and a terminated local type as $\texttt{end}$, where no actions remain to be taken.

We use μ-types to represent recursions [Pie02, § 21.8] at the level of global and local types. The operator μ binds a type variable $\mathbf{t}$ in a type $\mu\mathbf{t}.G$, and usual definitions of free and bound type variables arise.

Notation 2.2 (Omission of Terminated Types). The terminated types $\texttt{end}$ (global) and $\texttt{end}$ (local) are sometimes omitted when appearing as a continuation where unambiguous. For example, a global type $\mathbf{A} \to \mathbf{B} : \mathsf{Num}(\texttt{int}).\texttt{end}$ may appear as $\mathbf{A} \to \mathbf{B} : \mathsf{Num}(\texttt{int})$. ⌟

Notation 2.3 (Omission of Payload Types). Where the payload types S are not a topic of particular interest, we may omit the payload types for convenience and clarity. For example, a global type $\mathbf{A} \to \mathbf{B} : \mathsf{Num}(\texttt{unit}).\texttt{end}$ may appear as $\mathbf{A} \to \mathbf{B} : \mathsf{Num}.\texttt{end}$. ⌟

Notation 2.4 (Substitution). We write $G[G'/\mathbf{t}]$ to substitute free occurrences of the type variable $\mathbf{t}$ for a global type G' within a global type G; and $L[L'/\mathbf{t}]$ to substitute free occurrences of the type variable $\mathbf{t}$ for a local type L' within a local type L. ⌟

2.2.1 Notes on Recursive Types

With regards to recursion, usually an equi-recursive interpretation [Pie02, § 20.1] is taken, i.e. a recursive type $\mu\mathbf{t}.G'$ is always considered equal to its *unfolding* (i.e. replacing the type variable in the inner type with the entire recursive type) $G'[\mu\mathbf{t}.G'/\mathbf{t}]$, giving rise to an infinite type. In order for a recursive type to be well-formed, the type needs to be *contractive* [Pie02, § 21.8], i.e. successive unfoldings of a recursive type must give a constructor other than recursion. The usual example of a non-contractive type is $\mu\mathbf{t}.\mathbf{t}$, whose unfolding equals to itself. The requirement of contractivity is sometimes also known as *guardedness* [Mil80, § 5.4], in the sense that a type variable must be appear *guarded* under a constructor [Nak00].

Definition 2.5 (One-Time and Successive Unfoldings)**.** For a global type G, we define the *one-time* and *successive* unfoldings of G, written $\mathrm{unf}^1(G)$ and $\mathrm{unf}(G)$ respectively, as:

$$\mathrm{unf}^1(G) = \begin{cases} G'[\mu\mathbf{t}.G'/\mathbf{t}] & \text{if } G = \mu\mathbf{t}.G' \\ G & \text{otherwise} \end{cases} \qquad \mathrm{unf}(G) = \begin{cases} \mathrm{unf}(G'[\mu\mathbf{t}.G'/\mathbf{t}]) & \text{if } G = \mu\mathbf{t}.G' \\ G & \text{otherwise} \end{cases}$$

$\mathrm{unf}^1(L)$ and $\mathrm{unf}(L)$ is defined similarly for a local type L.

Furthermore, we define $\mathrm{unf}^n(G) = \mathrm{unf}^1\left(\mathrm{unf}^{n-1}(G)\right)$ (where $n > 1$) as the *n-time* unfolding of G. Similar definitions also apply for local types L. ⌟

Fact 2.6 (Contractivity and Unfolding). Fix a closed global type G (or a closed local type L, equivalently). $\mathrm{unf}(G)$ is defined whenever there exists a number $n > 0$ such that $\mathrm{unf}^n(G)$ is of a constructor other than recursion $\mu\mathbf{t}.G$ or type variable $\mathbf{t}$. Moreover, $\mathrm{unf}(G)$ is defined for any contractive global type G by the definition of contractivity (cf. [Pie02, § 21.8]) ⌟

The syntax of global and local types are sometimes represented in a *coinductive* way, as infinite trees, as done in [CFG+21; GJP+19; DGD23]. In general, recursive types using μ-types can be interpreted as regular infinite trees that have finitely many sub-trees [Pie02, §§ 21.7, 21.8].

Remark 2.7 (Degenerate Recursions). A recursive type can sometimes be represented in a more 'canonical' way with some recursion declarations removed, without modifying the substantive recursion body. We refer to these recursions as *degenerate*.

A form of degenerate recursive type $\mu\mathbf{t}.G'$ occurs when the type variable $\mathbf{t}$ does not appear in the inner type G'. The type is equivalent to the type G' without recursion.

In addition, a 'nested' recursive type $\mu\mathbf{t}.\mu\mathbf{t}'.G'$ is also considered degenerate. Nested recursion can be merged into a single one, as the type $\mu\mathbf{t}'.G'[\mathbf{t}'/\mathbf{t}]$, since both $\mathbf{t}$ and $\mathbf{t}'$ recurse to the same point. Some authors consider these forms of recursions as not guarded [vGHH21].

However, we do not consider a recursion as degenerate if it can be presented in a simplified form, e.g. $\mu\mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell.\mathbf{A} \to \mathbf{B} : \ell.\mathbf{t}$ can be simplified into a more compact form $\mu\mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell.\mathbf{t}$, because they unfold to the same infinite tree. Such simplification does not involve the removal of any recursion declarations. ⌟

2.2.2 Participants in a Global Type

We write $\mathrm{pts}(G)$ as the set of participants in a global type G. A participant $\mathbf{p}$ participates in G, if $\mathbf{p}$ occurs in a communication $\mathbf{p} \to \mathbf{q}$ or $\mathbf{q} \to \mathbf{p}$ (for some $\mathbf{q}$), we sometimes write $\mathbf{p} \in G$.

Definition 2.8 (Participant Set)**.** The set of participants of a global type G, written $\text{pts}(G)$, is defined as follows:

$$\begin{array}{rcl} \text{pts}(\texttt{end}) & = & \emptyset \\ \text{pts}(\mathbf{t}) & = & \emptyset \\ \text{pts}\Big(\mathbf{p} \to \mathbf{q}\{\ell_i(\mathsf{S}_i).G'_i\}_{i \in I}\Big) & = & \{\mathbf{p}, \mathbf{q}\} \cup \bigcup_{i \in I} \text{pts}(G'_i) \\ \text{pts}(\mu\mathbf{t}.G') & = & \text{pts}(G')\,. \end{array}$$

⌟

Lemma 2.9 (Participant Set after Substitution)**.** *Substitution does not remove any participants, and potentially adds participants in the global type that is substituted for:* $\text{pts}(G) \subseteq \text{pts}(G[G'/\mathbf{t}]) \subseteq \text{pts}(G) \cup \text{pts}(G')$. ⌟

Proof. By induction on G:

- Case $G = \texttt{end}$:

  $\texttt{end}$ remains $\texttt{end}$ after substitution, and $\text{pts}(\texttt{end}) = \emptyset$, trivial.

- Case $G = \mathbf{t}'$:

  If $\mathbf{t}' = \mathbf{t}$, then the type after substitution is G'. We then have $\text{pts}(\mathbf{t}[G'/\mathbf{t}]) = \text{pts}(G') \subseteq \text{pts}(G) \cup \text{pts}(G')$, as required. Moreover, $\text{pts}(\mathbf{t}') = \emptyset$, which satisfies the left part.

  If $\mathbf{t}' \neq \mathbf{t}$, then the type after substitution remains $\mathbf{t}'$, and $\text{pts}(\mathbf{t}') = \emptyset$, trivial.

- Case $G = \mu\mathbf{t}'.G'$:

  If $\mathbf{t}' = \mathbf{t}$, then no substitution occurs, trivial.

  If $\mathbf{t}' \neq \mathbf{t}$, then the type after substitution is $\mu\mathbf{t}'.G'[G'/\mathbf{t}]$. By definition $\text{pts}(\mu\mathbf{t}'.G') = \text{pts}(G')$, and the desired result follows from inductive hypothesis.

- Case $G = \mathbf{p} \to \mathbf{q}\,\{\ell_i(\mathsf{S}_i).G'_i\}$:

  By inductive hypothesis. □

Proposition 2.10 (One-Time Unfolding does not Change the Participant Set)**.** $\text{pts}(\mu\mathbf{t}.G') = \text{pts}(G'[\mu\mathbf{t}.G'/\mathbf{t}])$. ⌟

Proof. By definition, we know that $\text{pts}(\mu\mathbf{t}.G') = \text{pts}(G')$.

By Lem. 2.9, we know that $\text{pts}(G') \subseteq \text{pts}(G'[\mu\mathbf{t}.G'/\mathbf{t}]) \subseteq \text{pts}(G') \cup \text{pts}(\mu\mathbf{t}.G')$.

By expanding the definition and simplifications, we have that $\text{pts}(G') \subseteq \text{pts}(G'[\mu\mathbf{t}.G'/\mathbf{t}]) \subseteq \text{pts}(G')$, thus $\text{pts}(G'[\mu\mathbf{t}.G'/\mathbf{t}]) = \text{pts}(G') = \text{pts}(\mu\mathbf{t}.G')$, as required. □

$$
\begin{array}{rcll}
\text{Protocol Declarations} \quad P & ::= & \texttt{global protocol}\ p\ (\texttt{role}\ \mathbf{r}_1, \ldots, \texttt{role}\ \mathbf{r}_n)\{G\} & \\
\text{Protocol Constructs} \quad G & ::= & \ell(\mathtt{S})\ \texttt{from}\ \mathbf{r}_1\ \texttt{to}\ \mathbf{r}_2; G' & \text{Single Message} \\
& \mid & \texttt{choice at}\ \mathbf{r}\ \{G_1\}\ \texttt{or}\ \ldots\ \texttt{or}\ \{G_n\} & \text{Branches} \\
& \mid & \texttt{rec}\ \mathbf{X}\ \{G'\} \mid \texttt{continue}\ \mathbf{X} & \text{Recursion / Var.} \\
& \mid & \texttt{end}\ \text{(omitted in practice)} & \text{Termination} \\
& \mid & \texttt{do}\ p(\mathbf{r}_1, \ldots, \mathbf{r}_n) & \text{Protocol Call} \\
\text{Base Types} \quad \mathtt{S} & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots &
\end{array}
$$

Figure 2.1: Syntax of Core SCRIBBLE Language

2.2.3 Programmatic Representation in SCRIBBLE

We make a brief digression here from the theory, to introduce a protocol description language SCRIBBLE [YHN+14; NY19]. The primary aim of SCRIBBLE is to describe choreographies in web services, in the form of high-level multiparty communication. The first version of SCRIBBLE slightly predates the MPST theory [YHN+14, § 1], but later versions are largely based on MPST and its extensions. In addition to protocol description, SCRIBBLE also provides a toolchain for multiparty protocols, allowing developers to implement the protocol via the top-down methodology. We defer the description of the toolchain part to later sections.

We show the syntax of the core SCRIBBLE language in Fig. 2.1. A SCRIBBLE module (i.e. a unit of compilation) consists of multiple protocol declarations P. In the syntax, protocol names are represented by p, role names by $\mathbf{r}$, label names by ℓ, and recursion variable names by $\mathbf{X}$. The four kinds of names range over string identifiers.

We observe similarities between SCRIBBLE global protocols and global types (Def. 2.1), namely recursion declarations $\texttt{rec}\ \mathbf{X}\ \{G'\}$ vs. $\mu\mathbf{X}.G'$; recursion variables $\texttt{continue}\ \mathbf{X}$ vs. $\mathbf{X}$; singleton transmissions $\ell(\mathtt{S})\ \texttt{from}\ \mathbf{r}_1\ \texttt{to}\ \mathbf{r}_2; G'$ vs. $\mathbf{r}_1 \to \mathbf{r}_2 : \ell(\mathtt{S}).G'$; and the terminated type $\texttt{end}$ vs. $\texttt{end}$. Yet, we note that the plural ($|I| > 1$) form of message transmission $\mathbf{r}_1 \to \mathbf{r}_2\{\ell_i(\mathtt{S}_i).G'_i\}_{i \in I}$ is denoted by a $\texttt{choice at}\ \mathbf{r}_1$, where each branch is a single message $\ell_i(\mathtt{S}_i)\ \texttt{from}\ \mathbf{r}_1\ \texttt{to}\ \mathbf{r}_2; G'_i$. On the other hand, the $\texttt{choice}$ construct in SCRIBBLE is more flexible, e.g. messages sent from the same sender towards different receivers, and may not always have a corresponding global type (as shown in Def. 2.1).

2.3 Projecting a Global Type

In the standard top-down methodology of multiparty session type theory, a protocol *designer* only needs to define the global type, which governs the communication over all participants. For a developer to implement a participant, they need to know the local type of that participant, which is obtained via *projection*. If all the participants are implemented with regards to their projected local types, then the concurrent system is able to run *autonomously* (without a central coordinator) and *correctly*.

Local types describe the behaviour from the perspective of a single participant. Fix a participant **r** as our current role, and a global type G. The *projection* of G on **r**, written $G \upharpoonright \mathbf{r}$, describes the actions to be taken by **r** in order to realise the protocol described by G. We give some intuition of projection followed by its definition.

Intuition

Suppose a message is to be sent from a participant **p** to another participant **q**. The projection of that interaction depends on the view of our current role **r**, i.e. the participant that we project onto. In essence, projection of a global type onto a role **r** describes the relevant interactions involving that role (and erases the irrelevant interactions).

If **r** is the sender ($\mathbf{r} = \mathbf{p}$), then the current role **r** shall send a message to **q** to play their part in the global type, thus the projection on **r** is a sending action. Similarly, if **r** is the receiver ($\mathbf{r} = \mathbf{q}$), then our current role **r** shall receive a message from **p** to play their part, thus the projection on **r** is a receiving action.

The projection is more intriguing if our current role **r** is *neither* the sender **p** *nor* the receiver **q**, sometimes called a *non-choice participant*. In the simple case, if the message from **p** to **q** does not involve any choice (i.e. $|I| = 1$), then the initial message can be simply ignored. The projection on **r** is the same as that of the continuation that follows after the message.

Otherwise, our current role **r** needs to 'reconcile' projections of a number of continuations (since $|I| > 1$): this reconciliation is known as *merging* in the literature. It is not always possible to merge local types, which makes the projection a *partial* function (i.e. not defined for all global types). In the subsequent texts, we discuss two merge operators in the literature: the *plain* and the *full* merge operator, and explains differences in expressivity.

We give formal definitions of projection and merging in Def. 2.11.

Definition 2.11 (Projection of Global Types, Merge Operator)**.** Projection ($\upharpoonright$) is partial function from a global type and a participant to a local type:

$$(\mathbf{p} \to \mathbf{q}\{\ell_i(\mathsf{S}_i).G_i\}_{i\in I}) \upharpoonright \mathbf{r} = \begin{cases} \mathbf{q}\oplus\{\ell_i(\mathsf{S}_i).L_i\}_{i\in I} & \text{if } \mathbf{r} = \mathbf{p} \\ \mathbf{p}\&\{\ell_i(\mathsf{S}_i).L_i\}_{i\in I} & \text{if } \mathbf{r} = \mathbf{q} \\ \sqcup_{i\in I} L_i & \text{if } \mathbf{r} \notin \{\mathbf{p},\mathbf{q}\} \end{cases} \qquad (\text{where } L_i = G_i \upharpoonright \mathbf{r})$$

$$(\mu\mathbf{t}.G') \upharpoonright \mathbf{r} = \begin{cases} \mu\mathbf{t}.L' & \text{if } L' \neq \mu\widetilde{\mathbf{t}'}.\mathbf{t} \\ \mathtt{end} & \text{if } L' = \mu\widetilde{\mathbf{t}'}.\mathbf{t} \end{cases} \qquad (\text{where } L' = G' \upharpoonright \mathbf{r}\text{, and } \widetilde{\mathbf{t}'} \text{ denotes 0 or more type variables})$$

$$\mathbf{t} \upharpoonright \mathbf{r} = \mathbf{t}$$

$$\mathtt{end} \upharpoonright \mathbf{r} = \mathtt{end}$$

where $\sqcup$ is a *merge* operator defined on local types, and the notation $\sqcup_{i\in I} L_i$ is a shorthand for

$L_1 \sqcup L_2 \sqcup \ldots \sqcup L_n$ where $\{1 \ldots n\} = I$.

We define the *plain* merge operator as:

$$L \sqcup L = L \qquad \text{and undefined otherwise.}$$

We define the *full* merge operator as:

$$\begin{gathered}
\mathbf{p}\&\{\ell_i(\mathsf{S}_i).L'_i\}_{i\in I} \sqcup \mathbf{p}\&\{\ell_j(\mathsf{S}'_j).L''_j\}_{j\in J} = \\
\mathbf{p}\&\{\ell_k(\mathsf{S}_k).(L'_k \sqcup L''_k)\}_{k\in I\cap J} \cup \{\ell_i(\mathsf{S}_i).L'_i\}_{i\in I\setminus J} \cup \left\{\ell_j(\mathsf{S}'_j).L''_j\right\}_{j\in J\setminus I} \\
\text{where } \forall k \in I \cap J : \mathsf{S}_k = \mathsf{S}'_k; \\
\mathbf{p}\oplus\{\ell_i(\mathsf{S}_i).L'_i\}_{i\in I} \sqcup \mathbf{p}\oplus\{\ell_i(\mathsf{S}_i).L''_i\}_{i\in I} = \mathbf{p}\oplus\{\ell_i(\mathsf{S}_i).(L'_i \sqcup L''_i)\}_{i\in I} \\
\mu\mathbf{t}.L' \sqcup \mu\mathbf{t}.L'' = \mu\mathbf{t}.(L' \sqcup L'') \qquad \mathbf{t} \sqcup \mathbf{t} = \mathbf{t} \qquad \mathtt{end} \sqcup \mathtt{end} = \mathtt{end} \qquad \text{and undefined otherwise.}
\end{gathered}$$

⌟

We say a global type G is *projectable* onto a participant $\mathbf{r}$, if the projection $G \upharpoonright \mathbf{r}$ is defined. Usually, we consider global types G that are projectable onto all participants $\mathrm{pts}(G)$ as *well-formed*. As a result, the partial projection function rules out unrealisable global types.

The substantive clauses of the definition follows the intuition: the sender and receiver of a message gets a sending and receiving local type respectively. The more intriguing part is the *merge* operator, where we give definition of *plain* merge and *full* merge.

Plain merge was proposed in the first MPST paper [HYC08] (although not explicitly mentioned), where a non-choice participants must have a *uniform* behaviour regardless of the choice. This is useful for ruling out unrealisable global types, such as $G = \mathbf{A} \to \mathbf{B} \begin{Bmatrix} \ell_1.\mathbf{C} \to \mathbf{B} : \ell_1 \\ \ell_2.\mathbf{C} \to \mathbf{B} : \ell_2 \end{Bmatrix}$. In order to realise such a global type, there must be a 'covert' channel between $\mathbf{A}$ and $\mathbf{C}$ to relay the choice of label ℓ_1 or ℓ_2.

Full merge was initially proposed in [YDB+10], introduced as *mergeability* and *injection*.

> The mergeability relation states that two types are identical up to their branching types where only branches with distinct labels are allowed to be different.
>
> (Yoshida et al. [YDB+10, p. 136])

The mergeability relation induces the full merge operator, as shown in the formal definition. As a consequence, a non-choice participant may behave differently if they are informed of that choice *indirectly*. For example, taking a similar global type from the previous example, $G' = \mathbf{A} \to \mathbf{B} \begin{Bmatrix} \ell_1.\mathbf{B} \to \mathbf{C} : \ell_1 \\ \ell_2.\mathbf{B} \to \mathbf{C} : \ell_2 \end{Bmatrix}$. The participant $\mathbf{C}$ is able to learn the choice made by $\mathbf{A}$ via the relayed message from $\mathbf{B}$, giving them a receiving local type:

$$G' \upharpoonright \mathbf{C} = \mathbf{B}\&\ell_1 \sqcup \mathbf{B}\&\ell_2 = \mathbf{B}\& \begin{Bmatrix} \ell_1 \\ \ell_2 \end{Bmatrix}.$$

Note that the projection is not defined under plain merge.

The expressivity of different merge operators is studied and compared in a recent work by Stutz [Stu23, § 3]. In particular, they propose a modification of definition with regards to the recursive type, to allow merging of recursive types with different type variables:

$$\mu\mathbf{t}.L' \sqcup \mu\mathbf{t}'.L'' = \mu\mathbf{t}.(L' \sqcup L''[\mathbf{t}/\mathbf{t}']).$$

The remaining definitions of projection are the administrative clauses: type variables $\mathbf{t}$ and terminated global type $\texttt{end}$ are converted to their local type counterparts $\mathbf{t}$ and $\texttt{end}$; recursions $\mu\mathbf{t}.G'$ are converted inductively, but we need to take caution to ensure that the projected type $\mu\mathbf{t}.(G' \upharpoonright \mathbf{r})$ remains contractive. Therefore, if the result of projection might be non-contractive: either $\mu\mathbf{t}.\mathbf{t}$, or $\mu\mathbf{t}.\mu\mathbf{t}'_1.\cdots.\mu\mathbf{t}'_n.\mathbf{t}$, (which we collectively denote as $\mu\mathbf{t}.\mu\widetilde{\mathbf{t}'}.\mathbf{t}$ in the definition), a terminated type $\texttt{end}$ is produced instead of the non-contractive type[1].

Projection plays an important role in the multiparty session type theory, as it enables a top-down approach for protocol design and implementations. Here, projection is defined syntactically, but we will show later that projection also provides a connection between the semantics of global and local types.

2.4 Semantics of Global and Local Types

In this section, we give semantics for global and local types, using Labelled Transition System (LTS) [Kel76]. When a global type is implemented by a collection of participants via their projected local types, we would like to ensure that the behaviour of the projected collection matches what we model using a global type. This result is often called operational correspondence, and is crucial to correctness of the MPST theory.

We first introduce the definition of labelled transition systems, which we use to model semantics of global and local types.

Definition 2.12 (Labelled Transition System)**.** A *labelled transition system* is a tuple (S, A, δ), where S is a set of *states*, A is a set of labels, and $\delta \subseteq S \times A \times S$ is a set of transitions.

For any state $s, s' \in S$, and any label $\alpha \in A$, we write $s \xrightarrow{\alpha} s'$ whenever $(s, \alpha, s') \in \delta$. ⌟

Notation 2.13 (Related Reduction Notations). Fix a labelled transition system (S, A, δ).

For any state $s \in S$, and action $\alpha \in A$, we write $s \xrightarrow{\alpha}$, whenever there exists a state $s' \in S$ such that $s \xrightarrow{\alpha} s'$; we write $s \not\xrightarrow{\alpha}$, whenever there exists no state $s' \in S$ such that $s \xrightarrow{\alpha} s'$.

For any state $s \in S$, we write $s \rightarrow$, whenever there exists an action $\alpha \in A$ such that $s \xrightarrow{\alpha}$; we write $s \not\rightarrow$, whenever there exists no action $\alpha \in A$ such that $s \xrightarrow{\alpha}$.

We write $\rightarrow^*$ as the reflexive and transitive closure of $\rightarrow$. ⌟

We introduce a *synchronous* (*rendez-vous*) semantics of global and local types: there are no intermediate states for messages in transit, i.e. after the message is sent, before it is received.

[1]In this definition, the projected type may contain degenerate recursion even if the global type does not contain degenerate recursion.

The alternative is an *asynchronous* (*message passing*) semantics, where messages are placed in an intermediate buffer after sent. The asynchronous semantics and related results are presented by Deniélou and Yoshida [DY13]. We opt to show a synchronous version for simplicity, adapted from the asynchronous version. Similar adaptations can be found in [KY14; GJP+19].

Definition 2.14 (Transition Label)**.** An action $\alpha \in A$ is generated from the grammar $\alpha ::= \mathbf{p} \to \mathbf{q} : \ell(\mathsf{S})$, where $\mathbf{p} \neq \mathbf{q}$.

The *subject*s of an action α, written $\mathrm{subj}(\alpha)$, are defined as: $\mathrm{subj}(\mathbf{p} \to \mathbf{q} : \ell(\mathsf{S})) = \{\mathbf{p}, \mathbf{q}\}$. ⌟

We use a single form of action in the definition: this is due to the synchronous nature of our semantics. Consequently, the subject of an action involves two participants, namely *both* the sender and the receiver. In contrast, an asynchronous semantics, such as [DY13], uses separate sending and receiving actions, and each has a single subject.

In subsequent subsections, we define the transitions for global types and local types respectively, and explain the connection between them via projection.

2.4.1 Global Type Semantics

Definition 2.15 (Global Type Transition)**.** The transition relation of global types ($\xrightarrow{\alpha}$) is inductively defined as follows:

$$\frac{j \in I}{\mathbf{p} \to \mathbf{q}\{\ell_i(\mathsf{S}_i).G_i\}_{i\in I} \xrightarrow{\mathbf{p}\to\mathbf{q}:\ell_j(\mathsf{S}_j)} G_j}\ \text{[G-Pfx]} \qquad \frac{G[\mu\mathbf{t}.G/\mathbf{t}] \xrightarrow{\alpha} G'}{\mu\mathbf{t}.G \xrightarrow{\alpha} G'}\ \text{[G-Rec]}$$

$$\frac{\{\mathbf{p}, \mathbf{q}\} \cap \mathrm{subj}(\alpha) = \varnothing \quad \forall i \in I : G_i \xrightarrow{\alpha} G_i'}{\mathbf{p} \to \mathbf{q}\{\ell_i(\mathsf{S}_i).G_i\}_{i\in I} \xrightarrow{\alpha} \mathbf{p} \to \mathbf{q}\{\ell_i(\mathsf{S}_i).G_i'\}_{i\in I}}\ \text{[G-Cnt]}$$

⌟

We define transitions on *closed* global types (with regards to type variables). If the global type is a communication $\mathbf{p} \to \mathbf{q}\{\ell_i(\mathsf{S}_i).G_i\}_{i\in I}$, there are two possible rules that can engage: rule [G-Pfx] for a reduction in the prefix position and rule [G-Cnt] for a reduction in the continuation position.

The rule [G-Pfx] concerns a reduction in the *prefix* position of the communication: sender $\mathbf{p}$ communicates a message labelled ℓ_j with payload type S_j to receiver $\mathbf{q}$, provided that the message is one of those prescribed by the global type. After that, the global type continues as the selected continuation G_j.

Alternatively, the rule [G-Cnt] concerns a reduction in the *continuation* position. If an action α can be taken in all of the continuations, and the subjects of the action are disjoint from the prefix participants, the reduction may happen under the prefix. This rule is alternatively known as the 'permutation rule' [KY14], or the 'pass rule' [GLS+22]. The significance of this rule is to allow non-causally-related reductions.

It is important to note that the sequencing in global types are *weak* sequencing [RW94]. Take a global type $G = \mathbf{A} \to \mathbf{B} : \ell_1 . \mathbf{C} \to \mathbf{D} : \ell_2 . \mathtt{end}$. There is no causal relation between the message from $\mathbf{A}$ to $\mathbf{B}$ labelled ℓ_1 and the message from $\mathbf{C}$ to $\mathbf{D}$ labelled ℓ_2; therefore, the global type reductions allow both to take place, via rules [G-Pfx] and [G-Cnt], respectively.

The final rule [G-Rec] is an administrative rule to unfold a recursive global type one time $\mu \mathbf{t}.G$ into $G[\mu \mathbf{t}.G/\mathbf{t}]$ before reduction (i.e. $\mathrm{unf}^1(\mu \mathbf{t}.G)$). We would not need this rule if we considered global types as infinite trees [GJP+19; CFG+21].

Proposition 2.16 (Reductions under Unfolding)**.** *Unfolding a global type G (one time or successively) does not change its reductions:*

1. $G \xrightarrow{\alpha} G'$ if and only if $\mathrm{unf}^1(G) \xrightarrow{\alpha} G'$; and,

2. $G \xrightarrow{\alpha} G'$ if and only if $\mathrm{unf}(G) \xrightarrow{\alpha} G'$. ⌟

Proof. Apply or disapply rule [G-Rec] accordingly. □

We note that this transition relation ($\xrightarrow{\alpha}$) can be interpreted as a partial function, as is done in [GJP+19]. Fix a transition label α and a global type G, there exists at most one G' with $G \xrightarrow{\alpha} G'$.

Remark 2.17 (Parallel Global Types and Causality Analysis). In some MPST works [HYC08; BHT+10], the global type syntax includes an additional form: G_1, G_2 for parallel composition[2]. Usually, the participants in G_1 and G_2 are required to be distinct, and there is no corresponding parallel construct in local types (since a participant may only participate in one global type in the parallel composition).

In those works, a global type G undergoes a *linearity* analysis [HYC08, Def. 3.5] to check causality of communication. A causality analysis [HYC08, Fig. 5] is performed on a global type to ensure the absence of weak sequencing.

Such analysis is dispensed with in later works [DY13], by the introduction of reduction-under-prefix rules such as rule [G-Cnt]. In many recent works, the parallel composition constructs are usually omitted in global types for simplicity (without much loss of expressivity), and we follow the same approach in this thesis. ⌟

Remark 2.18 (Infinite States of Global Type Transitions). Let $\mathcal{R}(G)$ be the set of reachable global types (i.e. states) from G: $\mathcal{R}(G) = \{G' \mid G \to^* G'\}$. We note that the reachable set may be infinite due to recursive types.

We use a recursive form of the example we previously used to explain weak sequencing: let $G = \mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1 . \mathbf{C} \to \mathbf{D} : \ell_2 . \mathbf{t}$, we can reduce the type G by either interactions between $\mathbf{A}$ and $\mathbf{B}$, or interactions between $\mathbf{C}$ and $\mathbf{D}$. A possible sequence of reductions, consisting only of

[2]Some works [DY11; DY13] uses $G_1 | G_2$ in notation.

interactions between **A** and **B**, is shown as follows:

$$\begin{array}{ll}
 & \mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{A}\to\mathbf{B}:\ell_1} & \mathbf{C} \to \mathbf{D} : \ell_2.\mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{A}\to\mathbf{B}:\ell_1} & \mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{C} \to \mathbf{D} : \ell_2.\mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{A}\to\mathbf{B}:\ell_1} & \mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{C} \to \mathbf{D} : \ell_2.\mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{A}\to\mathbf{B}:\ell_1} & \ldots
\end{array}$$

Alternatively, another possible sequence of reductions, consisting only of interactions between **C** and **D**, is shown as follows:

$$\begin{array}{ll}
 & \mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{C}\to\mathbf{D}:\ell_2} & \mathbf{A} \to \mathbf{B} : \ell_1.\mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{C}\to\mathbf{D}:\ell_2} & \mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{A} \to \mathbf{B} : \ell_1.\mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{C}\to\mathbf{D}:\ell_2} & \mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{A} \to \mathbf{B} : \ell_1.\mu \mathbf{t}.\mathbf{A} \to \mathbf{B} : \ell_1.\mathbf{C} \to \mathbf{D} : \ell_2.\mathbf{t} \\
\xrightarrow{\mathbf{C}\to\mathbf{D}:\ell_2} & \ldots
\end{array}$$

It follows that the set $\mathcal{R}(G)$ can be infinite. ⌟

Remark 2.19 (Inductive Transition Rules). [3] Consider the following type G:

$$G = \mu \mathbf{t}.\mathbf{A} \to \mathbf{B} \left\{ \begin{array}{l} \mathsf{More}.\mathbf{t} \\ \mathsf{Done}.\mathbf{C} \to \mathbf{D} : \mathsf{Blah}.\mathsf{end} \end{array} \right\}.$$

The global type G describes a protocol between **A** and **B** where a (possibly empty) series of More messages are communicated following a Done message; and, *independently*, between **C** and **D** over an exchange of message Blah. Note that this global type G is not projectable on **C** (or **D**): merging fails between **t** and $\mathbf{D}\oplus\mathsf{Blah}.\mathsf{end}$.

Due to weak sequencing, the communication of the message Blah may occur in the absence of any other messages, following the spirit of rule [G-Cnt]. However, such an application cannot be realised under an inductive setup, since the derivation would be circular.

We set $\alpha = \mathbf{C} \to \mathbf{D} : \mathsf{Blah}$. To find the global type under reduction, we first observe that G is a recursive type and we apply rule [G-Rec] to unfold it once:

$$G[\mu \mathbf{t}.G/\mathbf{t}] = \mathbf{A} \to \mathbf{B} \left\{ \begin{array}{l} \mathsf{More}.\mu \mathbf{t}.\mathbf{A} \to \mathbf{B} \left\{ \begin{array}{l} \mathsf{More}.\mathbf{t} \\ \mathsf{Done}.\mathbf{C} \to \mathbf{D} : \mathsf{Blah}.\mathsf{end} \end{array} \right\} \\ \mathsf{Done}.\mathbf{C} \to \mathbf{D} : \mathsf{Blah}.\mathsf{end} \end{array} \right\}.$$

We note that the action does not match the prefix, therefore we need to engage rule [G-Cnt], where we have two subgoals, one for each branch. In the Done branch, the reduction is a

[3]With thanks to Keigo Imai for discussions regarding this remark.

straightforward application of rule [G-Pfx]. However, in the More branch, we have encountered the same goal as what we started with: the reduction of G over action α.

As a consequence, the reduction $G \xrightarrow{\mathsf{C} \to \mathsf{D} : \mathsf{Blah}} \mu\mathbf{t}.\mathsf{A} \to \mathsf{B} \begin{Bmatrix} \mathsf{More}.\mathbf{t} \\ \mathsf{Done}.\mathsf{end} \end{Bmatrix}$ is not derivable under an inductive interpretation of the transition rules.

One may be tempted to interpret the reduction rules coinductively, so that the aforementioned reduction can be derived. However, interpreting the rules using coinduction directly may include unintended transitions. For example, we can derive

$$\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t} \xrightarrow{\mathsf{C} \to \mathsf{D} : \mathsf{Blah}} \mathsf{A} \to \mathsf{B} : \mathsf{More}.\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t},$$

which would be absurd, because roles C and D do not participate in the global type. The coinductive derivation is given as follows:

$$\dfrac{\{\mathsf{A}, \mathsf{B}\} \cap \{\mathsf{C}, \mathsf{D}\} = \emptyset \quad \dfrac{\cdots}{\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t} \xrightarrow{\mathsf{C} \to \mathsf{D} : \mathsf{Blah}} \mathsf{A} \to \mathsf{B} : \mathsf{More}.\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t}}}{\dfrac{\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t} \xrightarrow{\mathsf{C} \to \mathsf{D} : \mathsf{Blah}} \mathsf{A} \to \mathsf{B} : \mathsf{More}.\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t}}{\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t} \xrightarrow{\mathsf{C} \to \mathsf{D} : \mathsf{Blah}} \mathsf{A} \to \mathsf{B} : \mathsf{More}.\mu\mathbf{t}.\mathsf{A} \to \mathsf{B} : \mathsf{More}.\mathbf{t}}\,[\text{G-Rec}]}\,[\text{G-Cnt}]$$

The optimal solution may be achieved with a mixed interpretation, e.g. an inference system using coaxioms by Dagnino [Dag19]: the rule [G-Pfx] must appear in the coinductive derivation at least once in order to remove vacuous transitions. ⌟

We have given the transitions of global types in this subsection. An interesting property to note is that a global type G that is not terminated ($G \neq \mathsf{end}$) is always able to reduce by some action (e.g. the first prefix after successive unfolding). This property is sometimes known as *progress*, and it will be useful later when we demonstrate the connection between semantics of global and local types.

2.4.2 Local Type Semantics

We give transitions of individual local types in Def. 2.20. Given the nature of communicating systems, giving semantics of a single agent in the system is not sufficient. We then give transitions of (closed) collections of local types (known as a *configuration*) in Def. 2.23.

Definition 2.20 (Local Type Transition)**.** The transition relation of local types ($\xrightarrow{\alpha}$), from the view point of a participant $\mathbf{p}$, is inductively defined as follows:

$$\frac{j \in I}{\mathbf{q}\oplus\{\ell_i(\mathsf{S}_i).L_i\}_{i\in I} \xrightarrow{\mathbf{p}\to\mathbf{q}:\ell_j(\mathsf{S}_j)} L_j}\text{[L-Send]} \qquad \frac{j \in I}{\mathbf{q}\&\{\ell_i(\mathsf{S}_i).L_i\}_{i\in I} \xrightarrow{\mathbf{q}\to\mathbf{p}:\ell_j(\mathsf{S}_j)} L_j}\text{[L-Recv]}$$

$$\frac{L[\mu\mathbf{t}.L/\mathbf{t}] \xrightarrow{\alpha} L'}{\mu\mathbf{t}.L \xrightarrow{\alpha} L'}\text{[L-Rec]}$$

⌟

Compared to the transition relation of global types, the local type transitions are more straightforward. The sequencing of actions is *strong* in local types, as there are no rules providing for reductions in continuations (cf. rule [G-Cnt]).

Remark 2.21 (Transition Actions in Local Types). We have used the same action labels in global and local types for simplicity. For a more precise modelling, the action labels can be split into sending and receiving, e.g. $\underline{\mathbf{p}} \to \mathbf{q} : \ell_j(\mathsf{S}_j)$ (where the underlined participant is sending) and $\mathbf{p} \to \underline{\mathbf{q}} : \ell_j(\mathsf{S}_j)$ (where the underlined participant is receiving). ⌟

Proposition 2.22 (Reductions under Unfolding)**.** *Unfolding a local type L (one time or successively) does not change its reductions:*

1. $L \xrightarrow{\alpha} L'$ *if and only if* $\mathrm{unf}^1(L) \xrightarrow{\alpha} L'$*; and,*
2. $L \xrightarrow{\alpha} L'$ *if and only if* $\mathrm{unf}(L) \xrightarrow{\alpha} L'$. ⌟

Proof. Apply or disapply rule [L-Rec] accordingly. □

Local types are collected together in a *configuration*, and we give definition of configurations and their transitions.

Definition 2.23 (Configuration)**.** A configuration $\mathcal{S} = \{L_\mathbf{r} \mid \mathbf{r} \in \mathbb{P}\}$ is a collection of local types, indexable via participants from a fixed set $\mathbb{P}$.

We write $\mathcal{S}(\mathbf{r})$ to denote the entry of participant $\mathbf{r}$ in the configuration $\mathcal{S}$.

We write $\mathrm{dom}(\mathcal{S})$ to denote the domain of the configuration $\mathcal{S}$, namely the fixed set $\mathbb{P}$. ⌟

Intuitively, a configuration $\mathcal{S}$ reduces by an action α if the subjects of that action α reduce, and all other participants remain unchanged.

Definition 2.24 (Configuration Transition)**.** The transition relation of configurations ($\xrightarrow{\alpha}$) is based on the transition relation of local types. We define $\mathcal{S} \xrightarrow{\alpha} \mathcal{S}'$, if:

1. $\mathrm{dom}(\mathcal{S}) = \mathrm{dom}(\mathcal{S}')$;
2. $\forall \mathbf{r} \in \mathrm{subj}(\alpha) : \mathcal{S}(\mathbf{r}) \xrightarrow{\alpha} \mathcal{S}'(\mathbf{r})$; and,
3. $\forall \mathbf{r} \in (\mathrm{dom}(\mathcal{S}) \setminus \mathrm{subj}(\alpha)) : \mathcal{S}(\mathbf{r}) = \mathcal{S}'(\mathbf{r})$. ⌟

Remark 2.25 (Reactive Configuration Semantics). Our configuration reductions $\rightarrow$ are *reactive* [vGHH21, Fig. 3]. A sender does not need a τ-transition (i.e. non-observation action) for choosing the message to send before a communication:

$$\frac{|I| > 1 \quad k \in I}{\mathbf{q}\oplus\{\ell_i.L'_i\}_{i\in I} \xrightarrow{\tau} \mathbf{q}\oplus\ell_k.L'_k}\ \text{[L-Send-Choice]}$$

Consequently, the configuration reductions under a non-reactive semantics require the sending side to be a singleton without choices, since any choices must be made prior to the synchronising communication.

Reactive semantics may hide certain undesirable behaviour. We reuse the example from van Glabbeek et al. [vGHH21, Ex. 7]: let $\text{dom}(\mathcal{S}) = \{\mathbf{A}, \mathbf{B}\}$, $\mathcal{S}(\mathbf{A}) = \mathbf{B}\oplus\{\ell_1.\texttt{end};\ \ell_2.\texttt{end}\}$ and $\mathcal{S}(\mathbf{B}) = \mathbf{A}\&\ell_1.\texttt{end}$. Under a non-reactive semantics, the configuration would be *stuck* if $\mathbf{A}$ selects label ℓ_2, i.e. a configuration $\mathcal{S} \xrightarrow{\tau} \mathcal{S}' \not\rightarrow$, where $\text{dom}(\mathcal{S}') = \{\mathbf{A}, \mathbf{B}\}$, $\mathcal{S}'(\mathbf{A}) = \mathbf{B}\oplus\ell_2.\texttt{end}$ and $\mathcal{S}'(\mathbf{B}) = \mathbf{A}\&\ell_1.\texttt{end}$. Under a reactive semantics (as in Def. 2.24), only the reductions available to both of the action subjects are considered, and the configuration $\mathcal{S}$ does not get stuck.

In this work, we focus on configurations associated with global types, and the interplay of projection and subtyping (explained in § 2.4.3) ensures that no such mismatch can occur. ⌟

We have given the transitions of local types and configurations in this subsection. Now that we have given semantics for both global types and local types, we are ready to explain the connection between them in the next subsection.

2.4.3 Relating Global Types and Configuration Semantics

Recall that MPST uses a top-down design methodology: global types govern the communication structure of processes. While we do not explicitly introduce a calculus for processes and a typing system, the local types describe the possible behaviours of a process. In other words, a global type provides an abstract model over a configuration, when each entry in the configuration is related to the *projection* upon that participant. In such cases, we show that there is an *operational correspondence* between the global type and the related configuration. As a consequence of operational correspondence, configurations governed by global types can be implemented independently and executed autonomously, while enjoying properties from their corresponding global types. We explain this relation between semantics of global types and configurations in this subsection.

Subtyping over Local Types and Configurations

Before explaining the relation between configurations and global types, we first introduce a limited[4] form of subtyping over local types. In this presentation, we do not consider any

[4]See [GJP+19] for the precise subtyping relation.

subtyping relations over the payload types (also known as *subsorting*[5]) when defining subtyping over local types.

Definition 2.26 (Subtyping on Local Types)**.** Subtyping ($\leq$) is a relation over closed local types, coinductively defined over the following rules:

$$\dfrac{}{\texttt{end} \leq \texttt{end}}\ \text{[S-End]} \qquad \dfrac{L'[\mu\mathbf{t}.L'/\mathbf{t}] \leq L}{\mu\mathbf{t}.L' \leq L}\ \text{[S-Rec-UnfL]} \qquad \dfrac{L \leq L'[\mu\mathbf{t}.L'/\mathbf{t}]}{L \leq \mu\mathbf{t}.L'}\ \text{[S-Rec-UnfR]}$$

$$\dfrac{\forall i \in I : L'_i \leq L''_i}{\mathbf{p}\oplus\{\ell_i(\mathsf{S}_i).L'_i\}_{i\in I} \leq \mathbf{p}\oplus\{\ell_i(\mathsf{S}_i).L''_i\}_{i\in I}}\ \text{[S-Send]} \qquad \dfrac{\forall i \in I : L'_i \leq L''_i}{\mathbf{p}\&\{\ell_i(\mathsf{S}_i).L'_i\}_{i\in I\cup J} \leq \mathbf{p}\&\{\ell_i(\mathsf{S}_i).L''_i\}_{i\in I}}\ \text{[S-Recv]}$$

⌟

A local type can be understood as a declaration of required communication behaviours for a given process: a receiving type $\mathbf{p}\&\{\ell_i(\mathsf{S}_i).L'_i\}_{i\in I}$ requires a process implementing this type to be able to handle *all* messages labelled ℓ_i from $\mathbf{p}$; a sending type $\mathbf{p}\oplus\{\ell_i(\mathsf{S}_i).L'_i\}_{i\in I}$ requires a process implementing this type to be able to send *one of* messages labelled ℓ_i to $\mathbf{p}$. In this sense, a local type L is a subtype of L', if the subtype L declares no more behaviour than the supertype L'; as a consequence, L poses no more requirements on implementing processes than L'.

Following this reasoning, the substantive rule [S-Recv], allows a receiving type in the supertype position (with index set I) to have fewer branches than one in the subtype position (with index set $I \cup J$).

The subtyping rules presented here are non-standard (cf. [GH05; GJP+19]) in the following way: our rule [S-Send] requires an identical index set I on both sides, instead of allowing fewer branches at the supertype position (as in standard subtyping relations). On this aspect, our subtyping relation follows the subtyping relation defined in Deniélou and Yoshida [DY13, Appendix § A.1].

We note two interesting properties of this subtyping relation. Suppose $L \leq L'$, all possible behaviour of the supertype L' are inhabited by the subtype L. The substantive rule for receiving types [S-Recv] ensures all receiving actions at the supertype position are preserved by the subtype. We note that this is also true for sending actions (rule [S-Send]), by not permitting additional behaviours at the supertype position, where we (following Deniélou and Yoshida [DY13]) depart from the standard subtyping relation. The other property is that the merge for two local types (if it exists), is a subtype of the local types being merged: take an index $i \in I$, we have $\sqcup_{i\in I} L_i \leq L_i$.

Proposition 2.27 (Subtyping is Reflexive)**.** *For any closed local type L, $L \leq L$.* ⌟

Proof. Each of the subtyping rule can be applied in a reflexive manner. Inspect the constructor of the local type L, and apply the rule matching the constructor accordingly. □

Lemma 2.28 (Subtyping Preserves Supertype Behaviour)**.** *If $L_1 \leq L_2$ and $L_2 \xrightarrow{\alpha} L'_2$, then there exists L'_1 such that $L_1 \xrightarrow{\alpha} L'_1$ and $L'_1 \leq L'_2$.* ⌟

[5]Some authors use the word 'sort' for payload types, as opposed to 'types' that denote session types.

Proof. We analyse each case of the subtyping relation $L_1 \leq L_2$ (Def. 2.26).

- Case [S-End]:

  Vacuous. No reduction is available for $\mathtt{end}$.

- Case [S-Send]:

  We have $\mathbf{p}\oplus\{\ell_i(S_i).L'_i\}_{i\in I} \leq \mathbf{p}\oplus\{\ell_i(S_i).L''_i\}_{i\in I}$ with $L'_i \leq L''_i$ for all $i \in I$. The only reduction rule that applies is [L-Send]. Take any $j \in I$, we have $\mathbf{p}\oplus\{\ell_i(S_i).L''_i\}_{i\in I} \xrightarrow{\mathbf{q}\to\mathbf{p}:\ell_j(S_j)} L''_j$.

  Using the same index $j \in I$, we have $\mathbf{p}\oplus\{\ell_i(S_i).L'_i\}_{i\in I} \xrightarrow{\mathbf{q}\to\mathbf{p}:\ell_j(S_j)} L'_j$, and we have $L'_j \leq L''_j$ from premise of subtyping, as required.

- Case [S-Recv]:

  We have $\mathbf{p}\&\{\ell_i(S_i).L'_i\}_{i\in I\cup J} \leq \mathbf{p}\&\{\ell_i(S_i).L''_i\}_{i\in I}$ with $L'_i \leq L''_i$ for all $i \in I$. The only reduction rule that applies is [L-Recv]. Take any $j \in I$, we have $\mathbf{p}\&\{\ell_i(S_i).L''_i\}_{i\in I} \xrightarrow{\mathbf{p}\to\mathbf{q}:\ell_j(S_j)} L''_j$.

  Using the same index $j \in I \cup J$, we have $\mathbf{p}\&\{\ell_i(S_i).L'_i\}_{i\in I\cup J} \xrightarrow{\mathbf{p}\to\mathbf{q}:\ell_j(S_j)} L'_j$, and we have $L'_j \leq L''_j$ from premise of subtyping, as required.

- Case [S-Rec-UnfL]:

  We have $\mu\mathbf{t}.L' \leq L$ with $L'[\mu\mathbf{t}.L'/\mathbf{t}] \leq L$. By Prop. 2.22, unfolding a local type one time does not alter its reduction behaviour, i.e. $L'[\mu\mathbf{t}.L'/\mathbf{t}]$ and L' have identical behaviour. The required result then follows from applying coinductive hypothesis on $L'[\mu\mathbf{t}.L'/\mathbf{t}] \leq L$.

- Case [S-Rec-UnfR]:

  Similar to Case [S-Rec-UnfL]. □

Lemma 2.29 (Merging Respects Subtyping). *If $L = L_1 \sqcup L_2$, then $L \leq L_1$ and $L \leq L_2$.* ⌟

Proof. The case for plain merging is simple: it follows from the reflexivity of subtyping (Prop. 2.27).

The case for full merging is more involved. Let us consider each case of full merging (Def. 2.11):

- When two receiving types are merged, namely $\mathbf{p}\&\{\ell_i(S_i).L'_i\}_{i\in I} \sqcup \mathbf{p}\&\{\ell_j(S'_j).L''_j\}_{j\in J} = \mathbf{p}\&\{\ell_k(S_k).(L'_k \sqcup L''_k)\}_{k\in I\cap J} \cup \{\ell_i(S_i).L'_i\}_{i\in I\setminus J} \cup \left\{\ell_j(S'_j).L''_j\right\}_{j\in J\setminus I}$, where $\forall k \in I \cap J : S_k = S'_k$.

  For simplicity, let us denote $L_l = \mathbf{p}\&\{\ell_i(S_i).L'_i\}_{i\in I}$ and L_{merged} be the result of the merge.

  Applying inductive hypothesis on $L'_k \sqcup L''_k$ (for $k \in I \cap J$), we have $(L'_k \sqcup L''_k) \leq L'_k$.

  To show $L_{merged} \leq L_l$, the rule [S-Recv] needs to be satisfied, where we need to show the following (dividing the index set I into two disjoint sets $I \setminus J$ and $I \cap J$): (1) $\forall i \in I \setminus J : L'_i \leq L'_i$ and $S_i = S_i$, which follows from reflexivity of subtyping; and (2) $\forall k \in I \cap J : (L'_k \sqcup L''_k) \leq L'_k$ and $S_k = S_k$, which follows from inductive hypothesis and side condition of merging.

- Other cases follow from applying inductive hypothesis and reflexivity of subtyping. □

We extend the subtyping relation over configurations via point-wise applications.

Definition 2.30 (Subtyping on Configurations)**.** Subtyping over configurations $\mathcal{S} \leq \mathcal{S}'$ holds, if: (1) $\mathrm{dom}(\mathcal{S}) = \mathrm{dom}(\mathcal{S}')$; and, (2) $\forall \mathbf{p} \in \mathrm{dom}(\mathcal{S}) : \mathcal{S}(\mathbf{p}) \leq \mathcal{S}'(\mathbf{p})$. ⌟

We note that two configurations related by subtyping have *identical* reductions, due to asymmetric nature of subtyping rules for sending and receiving types.

Lemma 2.31 (Configuration Subtyping Preserves Behaviour)**.** *Suppose* $\mathcal{S}_1 \leq \mathcal{S}_2$.

1. *If* $\mathcal{S}_2 \xrightarrow{\alpha} \mathcal{S}_2'$*, then there exists* $\mathcal{S}_1'$ *such that* $\mathcal{S}_1 \xrightarrow{\alpha} \mathcal{S}_1'$ *and* $\mathcal{S}_1' \leq \mathcal{S}_2'$*;*
2. *If* $\mathcal{S}_1 \xrightarrow{\alpha} \mathcal{S}_1'$*, then there exists* $\mathcal{S}_2'$ *such that* $\mathcal{S}_2 \xrightarrow{\alpha} \mathcal{S}_2'$ *and* $\mathcal{S}_1' \leq \mathcal{S}_2'$*.* ⌟

Proof. Item 1 follows directly from Lem. 2.28.

The case for Item 2 is more interesting: a reduction with action α requires both of the subjects to reduce with that action, namely the sender and the receiver. We first show the existence of a reduction of $\mathcal{S}_2$ with action α, the other requirement can be proved easily.

Suppose $\alpha = \mathbf{p} \to \mathbf{q} : \ell(\mathsf{S})$, and further suppose $\mathcal{S}_1 \xrightarrow{\alpha} \mathcal{S}_1'$ and $\mathcal{S}_2 \not\xrightarrow{\alpha}$. Since $\mathcal{S}_1 \leq \mathcal{S}_2$, we have $\mathcal{S}_1(\mathbf{p}) \leq \mathcal{S}_2(\mathbf{p})$. Since $\mathbf{p}$ is the sending rule, the substantive subtyping rule that applies to the local type of $\mathbf{p}$ must be [S-Send], which does not add or remove any behaviour. Contradiction. □

Relating Configurations and Global Types

We can now define a relation, which we call *association*, between a configuration $\mathcal{S}$ and a global type G.

Definition 2.32 (Association)**.** A configuration $\mathcal{S}$ is *associated to* a global type G, written $\mathcal{S} \leq [\![G]\!]$, if:

1. $\mathrm{pts}(G) \subseteq \mathrm{dom}(\mathcal{S})$; and,
2. $\forall \mathbf{p} \in \mathrm{dom}(\mathcal{S}) : \mathcal{S}(\mathbf{p}) \leq G \upharpoonright \mathbf{p}$. ⌟

We note that a configuration $\mathcal{S}$ can contain more participants than those of the global type G (Item 1). This is to allow *inactive* participants (where $\mathbf{p} \notin \mathrm{pts}(G)$ and $\mathcal{S}(\mathbf{p}) \leq \mathtt{end}$) to be present in the configuration.

Finally, the *operational correspondence* theorem says associated global types and configurations have the same reductions.

Theorem 2.33 (Operational Correspondence of Associated Global Types and Configurations)**.** *Suppose* $\mathcal{S} \leq [\![G]\!]$.

1. *If* $G \xrightarrow{\alpha} G'$*, then there exists* $\mathcal{S}'$ *such that* $\mathcal{S} \xrightarrow{\alpha} \mathcal{S}'$ *and* $\mathcal{S}' \leq [\![G']\!]$*;*

2. *If* $\mathcal{S} \xrightarrow{\alpha} \mathcal{S}'$, *then there exists* G' *such that* $G \xrightarrow{\alpha} G'$ *and* $\mathcal{S}' \leq [\![G']\!]$. ⌟

The theorem ensures that associated global types and configurations remain associated after making reductions. There are two main consequences. (1) Using the top-down methodology, we can use a global type to specify the communication structure of all participants, and obtain a configuration via projection. The projected configuration is associated with the global type initially, and will remain so along its possible reductions. (2) A global type is always able to reduce, unless it is terminated end. Applying the theorem, any configuration associated with a non-terminated global type is also always able to reduce — this is a property that is sometimes known as *deadlock freedom* [Kob06; DP22a] or *progress* [DdY07; HYC08; Pie02, § 8.3].

2.5 Summary and Concluding Remarks

In this chapter, we present one specific version of the multiparty session type theory, upon which we develop the rest of the thesis. In our presentation, we focus on the semantics of global and local types, and show their correspondence under projection and subtyping.

We decide not to include specifics of a session π-calculus, or a type system. Readers can follow Scalas and Yoshida [SY19] for a variant of session π-calculus with delegation (i.e. channel passing) and multiple sessions; or Ghilezan et al. [GJP+19] and Yoshida and Gheri [YG20] for a simpler variant that describes only a single session and without delegation.

In this thesis, we would not indulge into details with regards to delegation (channel passing) or session interleavings (i.e. a multi-session calculus). This is done based on anticipation on practical aspects: we consider the simple single-session model of multiparty session types, without channel passing to be more practically applicable, and thus we focus on the simple model and leave any extensions as potential future work.

3 A Theory of Refined Multiparty Session Types

In this chapter, we introduce a theory of *Refined Multiparty Session Types (RMPST)*. We first motivate this development in § 3.1. Then, we give the syntax of types in § 3.2, extending original multiparty session types (MPST) with *refinement types*. We give details of the refinement type system that we use to type expressions in RMPST in § 3.3.

We follow the standard MPST top-down design methodology. *Global types* describe communication structures of many *participants* (also known as *roles*). *Local types*, describing communication structures of a single participant, can be obtained via *projection* (explained in § 3.4). These local session types can then be used to implement each participant, which we will introduce as the main topic of the next chapter (§ 4).

We give semantics of global types and local types in § 3.5, and show the relation of semantics with respect to projection in § 3.6. In addition, we show how the refined global and local types can be erased into basic global and local types in § 3.7. As a consequence, we can compose all endpoint processes implementing local types projected from a given global type, to implement that global type correctly.

3.1 Motivation

From § 2, we know that multiparty session types are able to provide safety and liveness guarantees for a communication system. In particular, on the safety side, a configuration associated with a global type is free from communication mismatches of any payload type or label. This property is especially useful for practical programming purposes: implementations of communication channels in low-level programming usually use raw bytes, and thus are untyped. Without the safety guarantees, untyped data may be misinterpreted and cause errors, e.g. an `int`eger may be interpreted as a `string`, and a lack of the termination character (`\0`) may subsequently cause a buffer overflow.

Building upon the existing MPST theory, we would like to extend the safety guarantees on the value level. We do not only wish to say that the payload value types are correct, but also talk about the properties about the payload values. The ability to describe value-level properties is beneficial, since protocols often involve constraints (e.g. an `int`eger must be non-negative), currently unsupported by the existing MPST theory.

Our proposal, *refined* multiparty session types (RMPST), is to integrate *refinement types* [FP91; RKJ08] in the existing MPST theory, replacing a simple value-level type system with a more powerful refinement type system. By doing so, we extend MPST with the ability to describe value-level constraints using refinement types. Following the development of this chapter, we show the guarantees from MPST are also extended to RMPST.

3.2 Syntax of Refined Global and Local Types

We define the syntax of refined multiparty session types (RMPST) in Def. 3.1.

Definition 3.1 (Syntax of Refined Multiparty Session Types)**.** The syntax of base types $\mathtt{S}$, refinement types T, expressions E, global types G, and local types L are given as follows:

$\mathtt{S}$	$::=$	$\mathtt{int} \mid \mathtt{bool} \mid \ldots$	Base Types
T	$::=$	$x : \mathtt{S}\{E\}$	Refinement Types
E	$::=$	$x \mid \underline{c} \mid op_1\ E \mid E\ op_2\ E'$	Expressions
G	$::=$		Global Types
	$\mid$	$\mathbf{p} \to \mathbf{q}\{\ell_{\mathsf{i}}(x_i : T_i).G'_i\}_{i \in I}$	Message
	$\mid$	$\mu\mathbf{t}\ (x^{\mathbb{P}} := E : T).G'$	Recursion
	$\mid$	$\mathbf{t}\ \langle x := E \rangle \quad \mid \quad \mathtt{end}$	Type Var., End
L	$::=$		Local (Pre-)Types
	$\mid$	$\mathbf{p}\&\{\ell_{\mathsf{i}}(x_i : T_i).L'_i\}_{i \in I}$	Receiving
	$\mid$	$\mathbf{p}\oplus\{\ell_{\mathsf{i}}(x_i : T_i).L'_i\}_{i \in I}$	Sending
	$\mid$	$\sum\{\ell_{\mathsf{i}}(x_i : T_i).L'_i\}_{i \in I}$	Silent Prefix
	$\mid$	$\mu\mathbf{t}\ (x^{\omega} := E : T).L'$	Recursion (Unrestricted)
	$\mid$	$\mu\mathbf{t}\ (x^{0} : T).L'$	Recursion (Irrelevant)
	$\mid$	$\mathbf{t}\ \langle x := E \rangle \quad \mid \quad \mathbf{t} \quad \mid \quad \mathtt{end}$	Type Var., End

We use different colours for different syntactical categories to help disambiguation, but the syntax can be understood without colours. We use brown for global types, dark blue for local types, blue for expressions, dark green for base and refinement types, indigo with sans serif fonts for labels, and **Teal** with bold, sans serif fonts for participants (roles).

Participants are taken from a fixed set $\mathcal{R}$. We use lower case letters $\mathbf{p}, \mathbf{q}, \mathbf{r}, \ldots$ as meta-variables (it is possible for two meta-variables to refer to the same concrete role); and upper case letters $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$ for concrete (distinct) roles taken from the set of defined roles. ⌟

Value Types and Expressions

We use $\mathtt{S}$ for base types of values, ranging over integers, booleans, etc. We choose not to fix a set of base types, as they can be extended in the future when necessary, provided that values of the base types must be able to be transmitted. (For instance, a function is typically not considered transmittable.)

Expressions consist of variables x, constants $\underline{c}$ (e.g. integer literals $\underline{n}$, boolean literals $\mathtt{true}$ and $\mathtt{false}$), and unary (op_1) and binary (op_2) operations. The choice of operators are also not fixed, and open to extension when necessary, provided that they can be encoded a logical theory[1]. We assume common arithmetic and logical operators are in scope, and use them in examples.

A *refinement type* is of the form $(x : \mathtt{S}\{E\})$: the base type $\mathtt{S}$ is *refined* by a boolean expression, acting as a predicate on the members of the base type. A value x of the type has base type $\mathtt{S}$, and is refined by a boolean expression E. The boolean expression E acts as a predicate on the members x (possibly involving the variable x). For example, we can express natural numbers as $(x : \mathtt{int}\{x \geq 0\})$. We give more details on refinement type and the type system in § 3.3.

We use $\mathrm{fv}(\cdot)$ to denote the free variables in refinement types, expressions, etc. We consider variable x be bound in the refinement expression E, i.e. $\mathrm{fv}(x : \mathtt{S}\{E\}) = \mathrm{fv}(E) \setminus \{x\}$.

Where there is no ambiguity, we use the base type $\mathtt{S}$ directly as an abbreviation of a refinement type $(x : \mathtt{S}\{\mathtt{true}\})$, where x is a fresh variable, and $\mathtt{true}$ acts as a predicate that accepts all values of base type $\mathtt{S}$.

Global Types

Global types (or *protocols*) range over $G, G', G_i, \ldots$ Global types give an overview of the overall communication structure. We extend the standard global types (Def. 2.1) with refinement types and variable bindings in message prefixes. Extensions to the syntax are shaded in the following explanations.

$\mathbf{p} \to \mathbf{q}\left\{\ell_i(\boxed{x_i} : T_i).G_i\right\}_{i \in I}$ is a message from $\mathbf{p}$ to $\mathbf{q}$ (where $\mathbf{p} \neq \mathbf{q}$), which branches into one or more continuations with label ℓ_i, carrying a payload variable x_i with type T_i. We omit the curly braces when there is only one branch, like $\mathbf{p} \to \mathbf{q} : \ell(x : T).G'$. The payload variable x_i is bound in the continuation global type G_i, for all $i \in I$. We sometimes omit the variable if it is not used in the continuations or its refinement type. The free variables are defined as:

$$\mathrm{fv}(\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I}) = \bigcup_{i \in I} \mathrm{fv}(T_i) \cup \bigcup_{i \in I} (\mathrm{fv}(G_i) \setminus \{x_i\})$$

We require that the index set I is finite and not empty, and all labels ℓ_i are distinct. To avoid duplication, we write $\ell(x : \mathtt{S}\{E\})$ instead of $\ell(x : (x : \mathtt{S}\{E\}))$, where the first x denotes a variable in the message and binds any later continuations, and the second x represents member values in the refinement type and binds the refined expression E.

We extend the construct of recursive protocols to include a variable carrying a value in the inner protocol. In this way, we enhance the expressivity of the global types by allowing a recursion variable to be maintained across iterations of global protocols. A recursive global type $\mu \mathbf{t}\ \boxed{(x^{\mathbb{P}} := E : T)}\ .G'$ specifies a variable x (located in an non-empty set of participants $\mathbb{P}$) carrying type T in the recursive type, initialised with expression E. Participants in the set $\mathbb{P}$ will

[1]In practice, Satisfiability Modulo Theories (SMT) solvers are usually used to implement refinement type systems.

carry the variable x, and the variable x can be updated when unfolding the recursion (to be introduced later).

Without loss of generality, we rule out degenerate forms of recursion:

(1) the inner type G' must *not* also be a recursive type: otherwise, the inner type can be merged with the outer type;

(2) the type variable $\mathbf{t}$ must appear in the inner type G': otherwise, the recursion can be removed without effect.

The type variable $\mathbf{t}\left\langle \boxed{x := E} \right\rangle$ is annotated with an assignment of expression E to variable x. The assignment updates the variable x in the current recursive protocol to expression E. The free variables in recursive types and type variables are defined as:

$$\begin{aligned} \text{fv}\big(\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G'\big) &= \text{fv}(T) \cup \text{fv}(E) \cup (\text{fv}(G') \setminus \{x\}) \\ \text{fv}(\mathbf{t}\,\langle x := E \rangle) &= \text{fv}(E) \end{aligned}$$

We require that recursive types are contractive [Pie02, §21], so that recursive protocols have at least a message prefix, and protocols such as $\mu\mathbf{t}\,(x^{\{\mathsf{A}\}} := E_1 : T).\mathbf{t}\,\langle x := E_2 \rangle$ are not allowed. We also require recursive types to be closed with respect to type variables, e.g. protocols such as $\mathbf{t}\,\langle x := E \rangle$ alone are not allowed.

We write $G[\mu\mathbf{t}\,(x^{\mathbb{P}} : T).G/\mathbf{t}]$[2] to substitute all occurrences of type variables with expressions $\mathbf{t}\,\langle x := E \rangle$ into $\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G$ (note that E is preserved). We assume bound variables inside the global type G (the inner type) are refreshed during the substitution, so that there are different from existing variables.

We write $\mathbf{r} \in G$ to say a role $\mathbf{r}$ is a participating role in the global type G, i.e. $\mathbf{r}$ occurs in a communication prefix $\mathbf{r} \to \mathbf{p}\,\{\cdots\}$ or $\mathbf{p} \to \mathbf{r}\,\{\cdots\}$.

Notation 3.2 (No Variable Shadowing). For convenience, we assume that no variable in a given global type G and a local type L shadows another variable in the same type. ⌟

Example 3.3 (Substitution of Type Variables with Expressions)**.** Let

$$G_{rec} = \mu\mathbf{t}\,(x^{\{\mathsf{A},\mathsf{B}\}} := 0 : \mathtt{int}).G_{inner} \quad\text{and}\quad G_{inner} = \mathbf{A} \to \mathbf{B} : \mathsf{Msg}(y : \mathtt{int}).\mathbf{t}\,\langle x := x + 1 \rangle$$

When we unfold the recursion, we perform the substitution:

$$G_{inner}[\mu\mathbf{t}\,(x^{\{\mathsf{A},\mathsf{B}\}} : \mathtt{int}).G_{inner}/\mathbf{t}] = \begin{array}{l} \mathbf{A} \to \mathbf{B} : \mathsf{Msg}(y : \mathtt{int}). \\ \mu\mathbf{t}\,(x^{\{\mathsf{A},\mathsf{B}\}} := \underline{\underline{x}} + 1 : \mathtt{int}).\mathbf{A} \to \mathbf{B} : \mathsf{Msg}(y' : \mathtt{int}).\mathbf{t}\,\langle x := x + 1 \rangle. \end{array}$$

Note that the variable y inside recursion is refreshed to become y', and the type after substitution is *not* closed (the doubly underlined x is a free variable), and we explain the motivation of this behaviour when we introduce semantics. ⌟

[2]The initial expression E of the replacement recursive global type $\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G$ is omitted in the substitution notation. This is because the substitution operation would *disregard* any initial expression E in the replacement recursive type, and use only the expression provided in the type variable (i.e. the update expression E' in $\mathbf{t}\,\langle x := E' \rangle$) instead.

Example 3.4 (Global Types)**.** We give the following examples of global types.

1. $G_1 = \mathbf{A} \to \mathbf{B} : \mathsf{Fst}(x : \mathtt{int}).\mathbf{B} \to \mathbf{C} : \mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathbf{C} \to \mathbf{D} : \mathsf{Trd}(z : \mathtt{int}\{x = z\}).\mathtt{end}$.

   G_1 describes a protocol where **A** sends an `int` x to **B**, and **B** relays an `int` y that is equal to x to **C**, similarly for **C** to **D**. Note that we are allowed to write $x = z$ in the refinement of z, despite that x is not known to **C** (because x is transmitted between **A** and **B**).

2. $$G_2 = \mathbf{A} \to \mathbf{B} : \mathsf{Number}(x : \mathtt{int}).\mathbf{B} \to \mathbf{C} \begin{Bmatrix} \mathsf{Positive}(\mathtt{unit}\{x > 0\}).\mathtt{end} \\ \mathsf{Zero}(\mathtt{unit}\{x = 0\}).\mathtt{end} \\ \mathsf{Negative}(\mathtt{unit}\{x < 0\}).\mathtt{end} \end{Bmatrix}$$

   G_2 describes a protocol where **A** sends an `int` x to **B**, and **B** tells **C** whether the `int` x is Positive, Zero, or Negative. We omit the payload variables for the last message here, since it is not used later in the continuation.

3. $$\begin{aligned} G_3 = \mu\mathbf{t}\,(&try := 0 : \mathtt{int}\{try \geq 0 \land try \leq 3\}). \\ &\mathbf{A} \to \mathbf{B} : \mathsf{Password}(pwd : \mathtt{string}). \\ &\mathbf{B} \to \mathbf{A} \begin{Bmatrix} \mathsf{Correct}(\mathtt{unit}).\mathtt{end} \\ \mathsf{Retry}(\mathtt{unit}\{try < 3\}).\mathbf{t}\,\langle try := try + 1\rangle \\ \mathsf{Denied}(\mathtt{unit}\{try = 3\}).\mathtt{end} \end{Bmatrix} \end{aligned}$$

   G_3 describes a protocol where **A** authenticates with **B** with maximum 3 tries. ⌟

Local (Pre-)Types

Local types range over $L, L', L_i, \ldots$ Local types give a view of the communication structure of an endpoint. We distinguish *types* and *pre*-types as follows: a local *pre*-type is generated from the given syntax, whereas a local type is a local pre-type obtained from projecting a global type. We are usually interested in studying the properties of local types, and most of the theorems are concerned with local types (i.e. obtained via projection).

Taking the view point of the role **q**, the local type $\mathbf{p}\oplus\{\ell_i(x_i : T_i).L_i\}_{i \in I}$ describes that the role **q** sends a message to the partner role **p** ($\mathbf{p} \neq \mathbf{q}$, similar thereafter) with label ℓ_i (where i is selected from an index set I), carrying payload variable x_i with type T_i, and continues with L_i. It is also said that the role **q** takes an *internal choice*.

Dually, the local type $\mathbf{p}\&\{\ell_i(x_i : T_i).L_i\}_{i \in I}$ describes that the role **q** receives a message from the partner role **p**. In this case, it is also said that the role **q** offers an *external choice*. We omit curly braces when there is only a single branch (as is done for global messages).

We add a new syntax construct of $\sum\{\ell_i(x_i : T_i).L_i\}_{i \in I}$ for *silent local types*. In the singleton case of $|I| = 1$, we omit the summation sign and curly braces: $\ell(x : T).L$. We use this new construct of silent prefix to represent 'latent knowledge' obtained from the global protocol, but not in the form of a message. Silent local types are useful to model variables obtained under irrelevant quantification [Pfe01; AS12]. These variables can be used in the construction of a type, but cannot be used in that of an expression, as we explain later in § 3.3. We also show an example of a silent local type later in Ex. 3.20, after we show how silent local types are used in *projection*, the process of obtaining local types from a global type.

For recursive types, we use two variants of each constructs (recursion and type variables), for unrestricted x^{ω} and irrelevant x^{0} recursion variables. The unrestricted forms of constructs ($\mu\mathbf{t}\,(x^{\omega} := E : T).L'$ and $\mathbf{t}\,\langle x := E\rangle$) are analogous to their global type counterparts, which are used when the recursion variable is located at the current role (cf. the set $\mathbb{P}$ in the global type form). The irrelevant forms of constructs ($\mu\mathbf{t}\,(x^{0} : T).L'$ and $\mathbf{t}$) do not contain any expression E, compared to the unrestricted form, since no such expression is needed when the variable is not located at the current role. We define type variable substitutions in an analogous way to global types (for the unrestricted form).

Remark 3.5 (Single Payload Variable). For simplicity and convenience, wherever a payload variable occurs, the variable and its type are shown in a singleton form. We assume that multiple payload variables can be encoded in a singular form using a (dependent) pair, and may use them in examples.

In addition, we may also mix variables with different multiplicities in a local recursive type, e.g. $\mu\mathbf{t}\,(x^{\omega} := E : T, y^{0} : T').\cdots$, where the corresponding type variable only needs to supply expressions for variables declared under the restricted form ω. ⌟

In this section, we have given the syntax of various constructs in our refined MPST theory. Before going into details about the communication aspects, we will continue with a brief explanation of refinement types in the next section.

3.3 Typing Expressions with Refinement Types

We use E, E', E_i to range over expressions. We build upon a refinement type system for typing expressions, in the style of Rondon et al. [RKJ08]. We extend the typing contexts to include extra annotations for encoding *knowledge* of each variable: i.e. whether a role knows the value of the variable (currently or in the future).

Typing Contexts

We define two kinds of typing contexts, for use in handling global types and local types respectively. For convenience, we assume that variable names in a single typing context are mutually distinct.

$$\Gamma ::= \varnothing \mid \Gamma, x^{\mathbb{P}} : T \mid \Gamma, \widehat{x}^{\mathbb{P}} : T \qquad \Sigma ::= \varnothing \mid \Sigma, x^{\theta} : T \qquad \theta ::= 0 \mid \hat{0} \mid \omega$$

We annotate global and local typing context entries differently. For global contexts Γ, variables carry the annotation of a set of roles $\mathbb{P} \subseteq \mathcal{R}$, to denote the set of roles that have the knowledge of the value of the variables. In addition, we record two variants of variables in global context Γ, the unrestricted variant x (for current knowledge) and hatted variant $\widehat{x}$ (for prospective knowledge), and we assume that at most one variant is present in a global context for the same variable. For convenience, we may write $\widetilde{x}$ to denote either x or $\widehat{x}$. We explain the difference marked by the two variants later in § 3.5 when we introduce semantics.

For local contexts Σ, variables carry an annotation of their multiplicity θ. A variable with multiplicity 0 is an *irrelevantly quantified* variable (irrelevant variable for short), which must not appear in the expression when typing (sometimes denoted as $x \div T$ in the literature [Pfe01; AS12]). An irrelevant variable cannot appear in a value-level expression, and may only appear in a type-level expression (i.e. in the predicate of refinement types). A variable with multiplicity ω is a variable without restriction. We sometimes omit the multiplicity ω, and consider it as the default multiplicity. A variable with multiplicity $\hat{0}$ is *currently* an irrelevantly quantified variable, but may become an unrestricted variable in the future through reductions. This concept is similar to the hatted variant in global contexts, and denotes a form of prospective knowledge. We write $\tilde{0}$ to range over multiplicities 0 or $\hat{0}$.

Notation 3.6 (Membership Notations for Typing Contexts). We write $\widetilde{x}^{\mathbb{P}} : T \in \Gamma$ when there exist Γ_1, Γ_2 such that $\Gamma = \Gamma_1 , \widetilde{x}^{\mathbb{P}} : T , \Gamma_2$. We write $x \in \Gamma$ when there exists $\mathbb{P}, T$ such that $\widetilde{x}^{\mathbb{P}} : T \in \Gamma$; and $x \notin \Gamma$ for its negation.

Analogously, we write $x^{\theta} : T \in \Gamma$ when there exist Σ_1, Σ_2 such that $\Sigma = \Sigma_1 , x^{\theta} : T , \Sigma_2$. We write $x \in \Sigma$ when there exists θ, T such that $x^{\theta} : T \in \Sigma$, and $x \notin \Sigma$ for its negation. ⌟

We define the expansion of typing contexts ($\Gamma + x^{\mathbb{P}} : T; \Sigma + x^{\theta} : T$) in Def. 3.7.

Definition 3.7 (Typing Context Expansion)**.** The *expansion* of a global typing context Γ with an entry $\widetilde{x}^{\mathbb{P}} : T$, denoted $\Gamma + \widetilde{x}^{\mathbb{P}} : T$, is defined as follows:

$$\Gamma + x^{\mathbb{P}} : T = \begin{cases} \Gamma , x^{\mathbb{P}} : T & \text{if } x \notin \Gamma \\ \Gamma_1 , x^{\mathbb{P}} : T , \Gamma_2 & \text{if } \Gamma = \Gamma_1 , \widetilde{x}^{\mathbb{P}} : T , \Gamma_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Gamma + \widehat{x}^{\mathbb{P}} : T = \begin{cases} \Gamma , \widehat{x}^{\mathbb{P}} : T & \text{if } x \notin \Gamma \\ \Gamma_1 , \widehat{x}^{\mathbb{P}} : T , \Gamma_2 & \text{if } \Gamma = \Gamma_1 , \widehat{x}^{\mathbb{P}} : T , \Gamma_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The *expansion* of a local typing context Σ with an entry $x^{\theta} : T$, denoted $\Sigma + x^{\theta} : T$, is defined as follows:

$$\Sigma + x^{\omega} : T = \begin{cases} \Sigma , x^{\omega} : T & \text{if } x \notin \Sigma \\ \Sigma_1 , x^{\omega} : T , \Sigma_2 & \text{if } \Sigma = \Sigma_1 , x^{\omega} : T , \Sigma_2 \text{ or } \Sigma = \Sigma_1 , x^{\hat{0}} : T , \Sigma_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Sigma + x^{\tilde{0}} : T = \begin{cases} \Sigma\,, x^{\tilde{0}} : T & \text{if } x \notin \Sigma \\ \Sigma_1\,, x^{\tilde{0}} : T\,, \Sigma_2 & \text{if } \Sigma = \Sigma_1\,, x^{\tilde{0}} : T\,, \Sigma_2 \text{ (the multiplicity must be the same)} \\ \text{undefined} & \text{otherwise} \end{cases}$$

⌟

Different from appending an new entry to a typing context, *expansion* ensures that the resulting typing context does not only contain the new entry, but also includes no less information in the typing context, in a way that is consistent according to annotations or multiplicities. This definition will play an important role when we introduce semantics (§ 3.5). We explain the intricate details as follows.

If the new entry to be added is not present in the original typing context, the new entry is simply appended at the end. Otherwise, if an entry with identical annotation or multiplicity is present, then no change takes place, since the desired new information is already present. We now turn to the interesting cases, where prospective knowledge becomes promoted. If we expand a global context Γ with an entry with *current* knowledge $x^{\mathbb{P}} : T$, and an entry with *prospective* knowledge $\widehat{x}^{\mathbb{P}} : T$ is present: if the two entries share the same type T and role set $\mathbb{P}$, then the resulting context would contain a *current* knowledge entry, i.e. without hat (all other entries unchanged). This is to be contrasted with the converse case being *undefined*, as converting current knowledge into prospective knowledge would result in a loss of information, current knowledge cannot become prospective knowledge.

Following a similar reasoning, a prospective irrelevant entry (i.e. with hat) $x^{\hat{0}}$ in a local context Σ can be updated into x^{ω} when expanded, but not vice versa. An irrelevant entry (i.e. without hat) x^{0}, however, cannot be updated into x^{ω} during expansion. This distinction reflects that the multiplicity $\hat{0}$ denotes prospective knowledge, variables with that multiplicity may be promoted into an unrestricted multiplicity ω in the future.

Proposition 3.8 (Entry Present After Expansion)**.** *If* $\Gamma' = \Gamma + \widetilde{x}^{\mathbb{P}} : T$ *is defined, then* $\widetilde{x}^{\mathbb{P}} : T \in \Gamma'$*; if* $\Sigma' = \Sigma + x^{\theta} : T$ *is defined, then* $x^{\theta} : T \in \Sigma'$*.* ⌟

Proof. Follows directly from the definition of expansion (Def. 3.7). □

Well-formedness

We need to define well-formedness judgements on refinement types (under a typing context) for two reasons. The first is to ensure that variables occurring in a type are bound by the typing context, so that an open type is not well-formed. The second is to ensure that the *refinement* expression attached to the base type is a well-typed boolean expression, thus ruling out ill-formed types such as $x : \texttt{int}\{42\}$.

Before giving the well-formedness judgements, we first give some auxiliary definitions.

Definition 3.9 (Typing Context Promotion)**.** We define Γ^+ (resp. Σ^+) to be the *promotion* of a global (resp. local) typing context Γ (resp. Σ).

$$\begin{array}{rcl} (\varnothing)^+ & = & \varnothing \\ (\Gamma\,, \tilde{x}^{\mathbb{P}} : T)^+ & = & \Gamma^+\,, x^{\mathbb{P}} : T \end{array} \qquad \begin{array}{rcl} (\varnothing)^+ & = & \varnothing \\ (\Sigma\,, x^{\theta} : T)^+ & = & \Sigma^+\,, x^{\omega} : T \end{array} \quad \lrcorner$$

We define typing context *promotion* in Def. 3.9. This concept is used in well-formed judgement, for removing all restrictions on annotations or multiplicities for a given typing context.

We observe that each entry is promoted to the 'least restricted' form. The promoted empty context remains empty: $\varnothing^+ = \varnothing$. For global typing contexts, all entries with hats have the hat annotation removed, promoting all current or prospective knowledge to current: $\tilde{x}^{\mathbb{P}} : T$ is promoted to $x^{\mathbb{P}} : T$. For local typing contexts, all multiplicities (including irrelevant multiplicity 0) are promoted to become unrestricted: $x^{\theta} : T$ is promoted to $x^{\omega} : T$.

Lemma 3.10 (Promotion of Global Context Expansions)**.** $(\Gamma + \tilde{x}^{\mathbb{P}} : T)^+ = \Gamma^+ + x^{\mathbb{P}} : T$. ⌟

Proof. By case analysis of typing context expansions (Def. 3.7).

Suppose $x \notin \Gamma$: we expand the left hand side to $(\Gamma + \tilde{x}^{\mathbb{P}} : T)^+ = (\Gamma\,, \tilde{x}^{\mathbb{P}} : T)^+ = \Gamma^+\,, x^{\mathbb{P}} : T$, and the right hand side to $\Gamma^+ + x^{\mathbb{P}} : T = \Gamma^+\,, x^{\mathbb{P}} : T$.

Suppose $\tilde{x}^{\mathbb{P}} : T \in \Gamma$: we have that $\Gamma = \Gamma_1\,, \tilde{x}^{\mathbb{P}} : T\,, \Gamma_2$, and thus the promoted context is $\Gamma^+ = {\Gamma_1}^+\,, x^{\mathbb{P}} : T\,, {\Gamma_2}^+$. We proceed with case analyses over restrictions in the typing context entry and the expanding entry:

- Suppose $\Gamma = \Gamma_1\,, x^{\mathbb{P}} : T\,, \Gamma_2$. There is one valid case for expansion, i.e. LHS $= \Gamma + x^{\mathbb{P}} : T$.

  Before applying promotion on LHS, we have $\Gamma + x^{\mathbb{P}} : T = \Gamma$, and thus LHS $= \Gamma^+$. On RHS, we have $\Gamma^+ + x^{\mathbb{P}} : T = \Gamma^+$.

- Suppose $\Gamma = \Gamma_1\,, \widehat{x}^{\mathbb{P}} : T\,, \Gamma_2$. We first consider the case when the expanding entry has the same multiplicity, i.e. LHS $= \Gamma + \widehat{x}^{\mathbb{P}} : T = \Gamma$.

  Before applying promotion on LHS, we have again $\Gamma + \widehat{x}^{\mathbb{P}} : T = \Gamma$, and thus LHS $= \Gamma^+$. On RHS, we have $\Gamma^+ + x^{\mathbb{P}} : T = \Gamma^+$.

- Suppose $\Gamma = \Gamma_1\,, \widehat{x}^{\mathbb{P}} : T\,, \Gamma_2$. We then consider the case when the expanding entry has a different multiplicity, i.e. LHS $= \Gamma + x^{\mathbb{P}} : T = \Gamma$.

  Before applying promotion on LHS, we have $\Gamma + x^{\mathbb{P}} : T = \Gamma_1\,, x^{\mathbb{P}} : T\,, \Gamma_2$, and therefore LHS $= (\Gamma + x^{\mathbb{P}} : T)^+ = {\Gamma_1}^+\,, x^{\mathbb{P}} : T\,, {\Gamma_2}^+ = \Gamma^+$. On RHS, we have $\Gamma^+ + x^{\mathbb{P}} : T = \Gamma^+$. □

The well-formedness judgement of a refinement type is of form $\Sigma \vdash T$ ty, stating the refinement type T is well-formed under the local typing context Σ. We show the well-formedness rule [WF-Rty] below (there is only a single rule):

$$\frac{\Sigma^+\,, x : \mathtt{S} \vdash_{\text{base}} E : \mathtt{bool}}{\Sigma \vdash x : \mathtt{S}\{E\}\ \text{ty}}\ \text{[WF-Rty]}$$

$$\frac{\mathsf{Typeof}(\underline{c}) = \mathtt{S}}{\Sigma \vdash \underline{c} : (v{:}\mathtt{S}\{v = \underline{c}\})}\ [\text{TE-Const}] \qquad \frac{\mathsf{Typeof}(op_1) = \mathtt{S}_1 \to \mathtt{S}_2 \quad \Sigma \vdash E_1 : \mathtt{S}_1}{\Sigma \vdash op_1\ E_1 : (v{:}\mathtt{S}_2\{v = op_1\ E_1\})}\ [\text{TE-Unop}]$$

$$\frac{\mathsf{Typeof}(op_2) = \mathtt{S}_1 \to \mathtt{S}_2 \to \mathtt{S}_3 \quad \Sigma \vdash E_1 : \mathtt{S}_1 \quad \Sigma \vdash E_2 : \mathtt{S}_2}{\Sigma \vdash E_1\ op_2\ E_2 : (v{:}\mathtt{S}_3\{v = E_1\ op_2\ E_2\})}\ [\text{TE-Binop}]$$

$$\frac{}{\Sigma_1\,, x^{\omega} : (v{:}\mathtt{S}\{E\})\,, \Sigma_2 \vdash x : (v{:}\mathtt{S}\{E \wedge x = v\})}\ [\text{TE-Var}]$$

$$\frac{\Sigma \vdash E : (v{:}\mathtt{S}\{E_1\}) \quad \mathsf{Valid}(\llbracket\Sigma\rrbracket \wedge \llbracket E_1\rrbracket \implies \llbracket E_2\rrbracket)}{\Sigma \vdash E : (v{:}\mathtt{S}\{E_2\})}\ [\text{TE-Sub}]$$

Figure 3.1: Typing Rules for Expressions in a Local Typing Context

For a refinement type $x{:}\mathtt{S}\{E\}$ to be well-formed, the expression E must, under a *base* type system *without* refinements (using only base types), have a `bool`ean type under the *promoted* context Σ^+, extended with variable x (representing the members of the type) with a base type $\mathtt{S}$. The typing context Σ^+ promotes the irrelevant quantified variables $x^{\tilde{0}}$ into unrestricted variables x^{ω}, so that they can be used in the expression E inside the refinement type.

The well-formedness of a local typing context is defined inductively, requiring all refinement types in the context to be well-formed. We omit the judgements for brevity.

Typing Expressions

We assign refinement type to expressions under local typing contexts, forming judgements of form $\Sigma \vdash E : T$, and show key typing rules in Fig. 3.1. In particular, we modify the typing rules in a standard refinement type system [RKJ08; VSJ+14; VTC+17], to give considerations to the multiplicities attached to variables [Pfe01].

Rule [TE-Const] assigns constant values a refinement type that only contains that constant value, i.e. a singleton type affirming that the constant is of a type that has its own value. The predicate $\mathsf{Typeof}(\cdot)$ provides the base type of the constants, e.g. $\mathsf{Typeof}(\underline{42}) = \mathtt{int}$. As a result, the integer literal $\underline{42}$ can be assigned a refinement type $v{:}\mathtt{int}\{v = \underline{42}\}$.

Rules [TE-Unop] and [TE-Binop] gives typing derivations for unary and binary operators, with a corresponding refinement type that encodes the operation at type level. Again, the predicate $\mathsf{Typeof}(\cdot)$ provides the base function types of the operators. For example, the following (specialised) rule gives typing derivations for the plus operator.

$$\frac{\Sigma \vdash E_1 : \mathtt{int} \quad \Sigma \vdash E_2 : \mathtt{int}}{\Sigma \vdash E_1 + E_2 : (v{:}\mathtt{int}\{v = E_1 + E_2\})}\ [\text{TE-Plus}]$$

We draw attention to the handling of variables (rule [TE-Var]). An irrelevant variable in the typing context cannot appear in an expression [Pfe01, Fig. 1], i.e. there is *no* derivation for $\Sigma_1 , x^{\tilde{0}} : T , \Sigma_2 \vdash x : T$. These variables can only be used in an refinement type (rule [WF-Rty]).

One of the key features of refinement type systems is the use of a semantic subtyping relation [BGH+12], which we describe in rule [TE-Sub]. The semantic subtyping relation is determined by checking the validity of logical formulas that encode the refinement conditions: the predicate $\mathsf{Valid}(\cdot)$ queries a solver whether a formula is valid[3]. We write $[\![E]\!]$ to denote the encoding of expression E into the underlying logic[4] as a formula. For encoding typing contexts $[\![\Sigma]\!]$, we inductively encode each typing context entry $x^{\theta} : (v : \mathtt{S}\{E\})$ by encoding the term $E[x/v]$, i.e. substituting the variable v that represents all values in the refinement type with the variable name x in the typing context entry.

We say a global type G (resp. a local type L) is closed under a global typing context Γ (resp. a local typing context Σ), if all free variables in the type are in the domain of the typing context.

Lemma 3.11 (Weakening (New Entry))**.** *If* $\Sigma \vdash E : T$ *and* $\Sigma \vdash T'$ ty*, then* $\Sigma , x^{\theta} : T' \vdash E : T$. ⌟

Proof. By induction on the typing rules of expressions (Fig. 3.1). □

Lemma 3.12 (Weakening (Multiplicities))**.** *If* $\Sigma_1 , x^{\tilde{0}} : T , \Sigma_2 \vdash E : T'$*, then the multiplicity of* x *in the local typing context can be promoted, giving* $\Sigma_1 , x^{\omega} : T , \Sigma_2 \vdash E : T'$. ⌟

Proof. By induction on typing rules of expressions (Fig. 3.1). The notable case is [TE-Var]: the proposition holds vacuously, because there is no derivation as $\Sigma_1 , x^{\tilde{0}} : T , \Sigma_2 \vdash x : T$. □

Remark 3.13 (Empty Type). A refinement type may be *empty* (*uninhabited*), without any expression being able to be assigned that type.

We can construct such a type under the empty context $\varnothing$ as $(x : \mathtt{S}\{\mathtt{false}\})$ with any base types $\mathtt{S}$, since no value can satisfy the boolean expression of $\mathtt{false}$. Another example is a refinement type for an integer that is both negative and positive $(x : \mathtt{int}\{x > 0 \land x < 0\})$, where the refinement contains a contradiction. Similarly, under the typing context $x^{\omega} : \mathtt{int}\{x > 0\}$, the refinement type $(y : \mathtt{int}\{y < 0 \land y > x\})$ is empty. In these cases, the typing context with the specified type becomes inconsistent, i.e. the encoding into the underlying logic entails falsity.

Moreover, an empty type can also occur *without* inconsistency. For instance, in a typing context of $x^{0} : \mathtt{int}$, the type $(y : \mathtt{int}\{y > x\})$ is empty — it is not possible to produce such a value without referring to the variable x (cf. [TE-Var])[5]. ⌟

[3]Usually a Satisfiability Modulo Theories (SMT) solver is used, and the actual query to solver is a satisfiability query. A validity query is converted to a satisfiability query via applying negation.

[4]We do not impose any restriction on the specific logic to use. However, the choice of the logic may affect decidability and the availability of operators. For example, non-linear arithmetic operators, such as multiplication, may cause undecidability.

[5]We consider `int` to be the set of integers in the mathematical sense, and thus there is no maximum integer. Arguably, if `int` is the set of machine integers (e.g. 32-bit signed integers), then the type $(y : \mathtt{int}\{y \geq x\})$ could be inhabited by the maximum integer, without the need to refer to the variable x.

Remark 3.14 (Empty Local Type). A local type L can be empty (uninhabited) because one of the value types in the protocol is an empty type (cf. Rem. 3.13).

For example, the local type $\mathbf{A}\oplus\mathsf{Impossible}(x : \mathtt{int}\{x > 0 \land x < 0\}).\mathtt{end}$ cannot be implemented, since such an x cannot be provided due to inconsistency.

Without inconsistency, the local type $\mathsf{Pos}(x : \mathtt{int}\{x > 0\}).\mathbf{A}\oplus\mathsf{Impossible}(y : \mathtt{int}\{y > x\}).\mathtt{end}$ cannot be implemented. When constructing a message $\mathsf{Impossible}$ to send, it is forbidden to use the variable x. Therefore, an implementation would need to account for an arbitrary value of x. There is no integer greater than all arbitrary values of x. ⌟

Remark 3.15 (Implementable Local Types). Consider the following local type:

$$L = \mathbf{B}\&\mathsf{Num}(x : \mathtt{int}).\mathbf{B}\oplus \begin{Bmatrix} \mathsf{Pos}(\mathtt{unit}\{x > 0\}).\mathtt{end} \\ \mathsf{Neg}(\mathtt{unit}\{x < 0\}).\mathtt{end} \end{Bmatrix}.$$

When the variable x has the value 0, neither of the choices Pos or Neg could be selected, as the refinements are not satisfied. In this case, the local type L cannot be implemented, as the sending type may not be implemented in a *total* way. ⌟

In this section, we have introduced the value-level aspects of refined theory. We proceed with the communciation aspects, first by introducing projection in the next section.

3.4 Projection of Typing Contexts and Types

In the methodology of multiparty session types, developers specify a global type, and obtain local types for the participants via *projection*. In the original theory (§ 2.3), projection is a *partial* function that takes a global type G and a participant $\mathbf{p}$, and returns a local type L. The resulting local type L describes the local communication behaviour for participant $\mathbf{p}$ in the global scenario. Such workflow has the advantage that each endpoint can obtain a local type *separately*, and thus implementations of participants can benefit from this modularity.

Projection is defined as a *partial* function, since only *well-formed* global types can be projected to all participants. In this case, the partiality of projection arises from a local type compatibility relation ($\bowtie$), to be explained later in Def. 3.18, requiring a non-choice participant not to be 'confused' by the progression of the global type. In original MPST theory, a partial *merge operator* (Def. 2.11) is used to handle such situation. We motivate our deviations later in detail.

In RMPST, projection needs to be extended to include global typing contexts: a global type may need an accompanying typing context to track information of variables (along side knowledge of each participant). We first define the projection of global typing contexts (Fig. 3.2), and then define the projection of global types under a global typing context (Fig. 3.3). We use expression typing judgements in the definition of projection, to type-check expressions (e.g. at recursion definitions and type variables) against their prescribed types.

$$\frac{}{\varnothing \upharpoonright \mathbf{p} = \varnothing}\ \text{[P-Empty]} \qquad \frac{\mathbf{p} \in \mathbb{P} \qquad \Gamma \upharpoonright \mathbf{p} = \Sigma}{(\Gamma\,, x^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma\,, x^{\omega} : T}\ \text{[P-Var-}\omega\text{]} \qquad \frac{\mathbf{p} \in \mathbb{P} \qquad \Gamma \upharpoonright \mathbf{p} = \Sigma}{(\Gamma\,, \widehat{x}^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma\,, x^{\hat{0}} : T}\ \text{[P-Var-}\hat{0}\text{]}$$

$$\frac{\mathbf{p} \notin \mathbb{P} \qquad \Gamma \upharpoonright \mathbf{p} = \Sigma}{(\Gamma\,, \widetilde{x}^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma\,, x^{0} : T}\ \text{[P-Var-0]}$$

Figure 3.2: Projection Rules for Global Contexts $\boxed{\Gamma \upharpoonright \mathbf{p} = \Sigma}$

Projection of Global Typing Contexts

We define the judgement $\Gamma \upharpoonright \mathbf{p} = \Sigma$ for the projection of a global typing context Γ to a participant $\mathbf{p}$. The rules are given in Fig. 3.2. In the global context Γ, a variable x is annotated with a set of participants $\mathbb{P} \subseteq \mathcal{R}$ who know the value. We first consider whether the participant $\mathbf{p}$ should know the variable, by validating $\mathbf{p} \in \mathbb{P}$. If the projected participant does not know the variable, i.e. $\mathbf{p} \notin \mathbb{P}$, then rule [P-Var-0] gives an entry in local typing context with multiplicity 0.

Otherwise, if the projected participant $\mathbf{p}$ is in the set $\mathbb{P}$, then we need to examine whether the entry in the global typing context has current or perspective knowledge (i.e. whether the entry has a hat). Rule [P-Var-ω] is applied to obtain an unrestricted variable x^{ω} for an entry with current knowledge (without hat); rule [P-Var-$\hat{0}$] is applied to obtain a perspective irrelevant variable $x^{\hat{0}}$ for an entry with perspective knowledge (with hat).

Proposition 3.16 (Projection of Context Entries). *Suppose* $\Gamma \upharpoonright \mathbf{r} = \Sigma$.

1. *If* $x^{\mathbb{P}} : T \in \Gamma$ *and* $\mathbf{r} \in \mathbb{P}$, *then* $x^{\omega} : T \in \Sigma$;
2. *If* $\widehat{x}^{\mathbb{P}} : T \in \Gamma$ *and* $\mathbf{r} \in \mathbb{P}$, *then* $x^{\hat{0}} : T \in \Sigma$;
3. *If* $\widetilde{x}^{\mathbb{P}} : T \in \Gamma$ *and* $\mathbf{r} \notin \mathbb{P}$, *then* $x^{0} : T \in \Sigma$;
4. *If* $x \notin \Gamma$, *then* $x \notin \Sigma$. ⌟

Proof. Follows directly from the definition of global typing context projection (Fig. 3.2). □

Lemma 3.17 (Projection of Expansions). *Suppose* $\Gamma \upharpoonright \mathbf{p} = \Sigma$. *The projection of global typing context* Γ *expanded with* $x^{\mathbb{P}} : T$ *or* $\widehat{x}^{\mathbb{P}} : T$ *onto* $\mathbf{p}$ *satisfies that:*

1. $(\Gamma + x^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \begin{cases} \Sigma + x^{\omega} : T & \text{if } \mathbf{p} \in \mathbb{P} \\ \Sigma + x^{0} : T & \text{if } \mathbf{p} \notin \mathbb{P} \end{cases}$

2. $(\Gamma + \widehat{x}^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \begin{cases} \Sigma + x^{\hat{0}} : T & \text{if } \mathbf{p} \in \mathbb{P} \\ \Sigma + x^{0} : T & \text{if } \mathbf{p} \notin \mathbb{P} \end{cases}$ ⌟

Proof. By case analysis of typing context expansion (Def. 3.7).

1. - Case $\Gamma + x^{\mathbb{P}} : T = \Gamma , x^{\mathbb{P}} : T$, when $x \notin \Gamma$.

     If $\mathbf{p} \in \mathbb{P}$ (resp. $\mathbf{p} \notin \mathbb{P}$), we have the projection $(\Gamma + x^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma , x^{\omega} : T$ by rule [P-Var-ω] (resp. $(\Gamma + x^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma , x^{0} : T$ by rule [P-Var-0]). Since $x \notin \Gamma$ and $\Gamma \upharpoonright \mathbf{p} = \Sigma$, we can apply Prop. 3.16 (Item 4) to know $x \notin \Sigma$. Then, we can apply expansion to get $\Sigma + x^{\omega} : T = \Sigma , x^{\omega} : T$ (resp. $\Sigma + x^{0} : T = \Sigma , x^{0} : T$), as required.
   - Case $\Gamma + x^{\mathbb{P}} : T = \Gamma_1 , x^{\mathbb{P}} : T , \Gamma_2$ when $\Gamma = \Gamma_1 , \widetilde{x}^{\mathbb{P}} : T , \Gamma_2$.

     If $\mathbf{p} \in \mathbb{P}$, we have $(\Gamma + x^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma_1 , x^{\omega} : T , \Sigma_2$ by rule [P-Var-ω], where $\Gamma_1 \upharpoonright \mathbf{p} = \Sigma_1$ and $\Gamma_2 \upharpoonright \mathbf{p} = \Sigma_2$. We consider two sub-cases depending on the annotation of x in Γ.
     - If $x^{\mathbb{P}} : T \in \Gamma$, since $\Gamma \upharpoonright \mathbf{p} = \Sigma$, we can apply Prop. 3.16 (Item 1) to know $x^{\omega} : T \in \Sigma$. Since variables names are unique in a typing context, we have $\Sigma = \Sigma_1 , x^{\omega} : T , \Sigma_2$. Then, we can apply expansion to get $\Sigma + x^{\omega} : T = \Sigma_1 , x^{\omega} : T , \Sigma_2$, as required.
     - Otherwise, $\widehat{x}^{\mathbb{P}} : T \in \Gamma$, since $\Gamma \upharpoonright \mathbf{p} = \Sigma$, we can apply Prop. 3.16 (Item 2) to know $x^{\hat{0}} : T \in \Sigma$. Since variables names are unique in a typing context, we have $\Sigma = \Sigma_1 , x^{\hat{0}} : T , \Sigma_2$. Then, we can apply expansion to get $\Sigma + x^{\omega} : T = \Sigma_1 , x^{\omega} : T , \Sigma_2$, as required.

     If $\mathbf{p} \notin \mathbb{P}$, we have $(\Gamma + x^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma_1 , x^{0} : T , \Sigma_2$ by rule [P-Var-0], where $\Gamma_1 \upharpoonright \mathbf{p} = \Sigma_1$ and $\Gamma_2 \upharpoonright \mathbf{p} = \Sigma_2$. We have $\widetilde{x}^{\mathbb{P}} : T \in \Gamma$ and $\Gamma \upharpoonright \mathbf{p} = \Sigma$, so we can apply Prop. 3.16 (Item 3) to know $x^{0} : T \in \Sigma$. Since variables names are unique in a typing context, we have $\Sigma = \Sigma_1 , x^{0} : T , \Sigma_2$. Then, we can apply expansion to get $\Sigma + x^{0} : T = \Sigma_1 , x^{0} : T , \Sigma_2$, as required.

2. - Case $\Gamma + \widehat{x}^{\mathbb{P}} : T = \Gamma , \widehat{x}^{\mathbb{P}} : T$, when $x \notin \Gamma$.

     If $\mathbf{p} \in \mathbb{P}$ (resp. $\mathbf{p} \notin \mathbb{P}$), we have the projection $(\Gamma + \widehat{x}^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma , x^{\hat{0}} : T$ by rule [P-Var-$\hat{0}$] (resp. $(\Gamma + \widehat{x}^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma , x^{0} : T$ by rule [P-Var-0]). Since $x \notin \Gamma$ and $\Gamma \upharpoonright \mathbf{p} = \Sigma$, we can apply Prop. 3.16 (Item 4) to know $x \notin \Sigma$. Then, we can apply expansion to get $\Sigma + x^{\hat{0}} : T = \Sigma , x^{\hat{0}} : T$ (resp. $\Sigma + x^{0} : T = \Sigma , x^{0} : T$), as required.
   - Case $\Gamma + \widehat{x}^{\mathbb{P}} : T = \Gamma_1 , \widehat{x}^{\mathbb{P}} : T , \Gamma_2$ when $\Gamma = \Gamma_1 , \widehat{x}^{\mathbb{P}} : T , \Gamma_2$.

     If $\mathbf{p} \in \mathbb{P}$, we have $(\Gamma + \widehat{x}^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma_1 , x^{\hat{0}} : T , \Sigma_2$ by rule [P-Var-$\hat{0}$], where $\Gamma_1 \upharpoonright \mathbf{p} = \Sigma_1$ and $\Gamma_2 \upharpoonright \mathbf{p} = \Sigma_2$. We have $\widehat{x}^{\mathbb{P}} : T \in \Gamma$, and $\Gamma \upharpoonright \mathbf{p} = \Sigma$, so we can apply can apply Prop. 3.16 (Item 2) to know $x^{\hat{0}} : T \in \Sigma$. Since variables names are unique in a typing context, we have $\Sigma = \Sigma_1 , x^{\hat{0}} : T , \Sigma_2$. Then, we can apply expansion to get $\Sigma + x^{\hat{0}} : T = \Sigma_1 , x^{\hat{0}} : T , \Sigma_2$, as required.

     If $\mathbf{p} \notin \mathbb{P}$, we have $(\Gamma + \widehat{x}^{\mathbb{P}} : T) \upharpoonright \mathbf{p} = \Sigma_1 , x^{0} : T , \Sigma_2$ by rule [P-Var-0], where $\Gamma_1 \upharpoonright \mathbf{p} = \Sigma_1$ and $\Gamma_2 \upharpoonright \mathbf{p} = \Sigma_2$. We have $\widehat{x}^{\mathbb{P}} : T \in \Gamma$ and $\Gamma \upharpoonright \mathbf{p} = \Sigma$, so we can apply Prop. 3.16 (Item 3) to know $x^{0} : T \in \Sigma$. Since variables names are unique in a typing context, we have $\Sigma = \Sigma_1 , x^{0} : T , \Sigma_2$. Then, we can apply expansion to get $\Sigma + x^{0} : T = \Sigma_1 , x^{0} : T , \Sigma_2$, as required. □

Projection of Global Types

Before we give the rules for projection, we need another auxiliary definition. We define (global) type variable contexts as $\Phi ::= \varnothing \mid \Phi , \mathbf{t} : \langle x^{\mathbb{P}}, T \rangle$. We use type variable contexts to track the recursion variable and its type at each type variable declaration.

$$(\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i\in I}) \upharpoonright_\Phi \mathbf{r} = \begin{cases} \mathbf{q}\oplus\{\ell_i(x_i : T_i).L_i\}_{i\in I} & \text{if } \mathbf{r} = \mathbf{p} \\ \mathbf{p}\&\{\ell_i(x_i : T_i).L_i\}_{i\in I} & \text{if } \mathbf{r} = \mathbf{q} \\ \sum\{\ell_i(x_i : T_i).L_i\}_{i\in I} & \text{if } \mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\}, \forall i \in I : \mathbf{r} \in G_i \text{ and } \bowtie_{i\in I} L_i \\ \mathtt{end} & \text{if } \mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\} \text{ and } \forall i \in I : \mathbf{r} \notin G_i \\ \text{undefined} & \text{if } \mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\} \text{ and (not } \bowtie_{i\in I} L_i \text{ or} \\ & \exists i, j \in I : \mathbf{r} \in G_i \text{ and } \mathbf{r} \notin G_j) \end{cases}$$
$$\text{where } L_i = G_i \upharpoonright_\Phi \mathbf{r}$$

$$(\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G) \upharpoonright_\Phi \mathbf{r} = \begin{cases} \mu\mathbf{t}\,(x^{\omega} := E : T).L & \text{if } \mathbf{r} \in \mathbb{P} \text{ and } \mathbf{r} \in G \\ \mu\mathbf{t}\,(x^{0} : T).L & \text{if } \mathbf{r} \notin \mathbb{P} \text{ and } \mathbf{r} \in G \\ \mathtt{end} & \text{if } L = \mu\widetilde{\mathbf{t}'(\cdots)}.\mathbf{t} \text{ or } L = \mu\widetilde{\mathbf{t}'(\cdots)}.\mathbf{t}\,\langle x := E' \rangle \end{cases}$$
$$\text{where } L = G \upharpoonright_{\Phi, \mathbf{t} : \langle x^{\mathbb{P}}, T \rangle} \mathbf{r}$$

$$\mathbf{t}\,\langle x := E \rangle \upharpoonright_\Phi \mathbf{r} = \begin{cases} \mathbf{t}\,\langle x := E \rangle & \text{if } \Phi(\mathbf{t}) = \langle x^{\mathbb{P}}, T \rangle \text{ and } \mathbf{r} \in \mathbb{P} \\ \mathbf{t} & \text{if } \Phi(\mathbf{t}) = \langle x^{\mathbb{P}}, T \rangle \text{ and } \mathbf{r} \notin \mathbb{P} \end{cases}$$

$$\mathtt{end} \upharpoonright_\Phi \mathbf{r} = \mathtt{end}$$

Figure 3.3: Projection Rules for Global Types $\boxed{G \upharpoonright_\Phi \mathbf{p} = L}$

We write $G \upharpoonright_{\Phi} \mathbf{r} = L$ for projecting a global type G onto a role $\mathbf{r}$, under the type variable context Φ, resulting in a local type L (when projection is defined). Projection rules are shown in Fig. 3.3. We now explain the projection rules in the following text.

If the prefix of G is a message from role $\mathbf{p}$ to role $\mathbf{q}$, the projection onto role $\mathbf{p}$ (resp. $\mathbf{q}$) results in a local type with a send (resp. receive) prefix, similar to the original theory. For other participating roles $\mathbf{r}$ (also known as a *third* role), the projection results in a local type with a *silent label*, with a prefix $\ell_i(x_i : T_i)$. This follows the concept of a coordinated distributed system, where all the participants follow a global protocol, and base assumptions of their local actions on actions of participants roles not involving them. Noticeably, the projection defined in the original MPST theory does not contain information for role $\mathbf{r}$ about a message between $\mathbf{p}$ and $\mathbf{q}$. We depart from the original theory, and use the silent prefix to retain such information, especially the refinement type T of the payload.

When projecting upon a third role, we use a compatibility relation $\bowtie$ defined over local types, to characterise local types that can be safely combined without 'causing confusion'. We give the definition of the compatibility relation in Def. 3.18.

Definition 3.18 (Local Type Compatibility)**.** *Compatibility* ($\bowtie$) is a relation over local types L, such that:

$$\frac{\forall i \in I : \forall j \in J : \ell_i \neq \ell_j}{\mathbf{p}\&\{\ell_i(x_i : T_i).L_i\}_{i \in I} \bowtie \mathbf{p}\&\{\ell_j(x_j : T_j).L_j\}_{j \in J}} \qquad \frac{\forall i \in I : L_i \bowtie L}{\sum\{\ell_i(x_i : T_i).L_i\}_{i \in I} \bowtie L}$$

$$\frac{\forall j \in J : L \bowtie L_j}{L \bowtie \sum\{\ell_j(x_j : T_j).L_j\}_{j \in J}}$$

We write $\bowtie_{i \in I} L_i$, if for all $i, j \in I, i \neq j$ implies $L_i \bowtie L_j$ holds, i.e. each distinct pairs of local types in the set are compatible. ⌟

Used as a side condition for projection of a global type onto a third role, the compatibility relation ensures the *uniqueness* of choice. (Note that $\bowtie_{i \in i} L_i$ for a singleton set holds vacuously.) This will be explained in more detail when we introduce local type semantics in § 3.5. In essence, the compatibility relation disregards silent prefixes in a local type, and requires two receiving prefixes to have disjoint labels. If two receiving prefixes have non-disjoint labels, we would not be able to uniquely determine which local type is engaged. We do not consider sending prefixes to be compatible at all. Allowing two sending prefixes with identical labels would also make it impossible to ascertain which local type is engaged; and sending prefixes with non-identical labels are non-mergeable in the basic MPST theory (cf. Def. 2.11).

Example 3.19 (Compatible and Incompatible Local Types)**.** We show some examples of compatible and incompatible local types:

1. Let $L_1 = \mathbf{A}\&\ell_1(\mathtt{int}).\mathtt{end}$ and $L_2 = \mathbf{A}\&\ell_2(\mathtt{int}).\mathtt{end}$.

We have $L_1 \bowtie L_2$, since the two local types are both receiving from $\mathbf{A}$, and have distinct labels.

Furthermore, let $L'_1 = \ell_3(v_1 : \mathtt{int}).L_1$ and $L'_2 = \ell_4(v_2 : \mathtt{int}).L_2$.

We have $L'_1 \bowtie L'_2$, since the compatibility relation ignores silent prefixes, and thus $L_1 \bowtie L_2$.

2. Let $L_3 = \mathbf{A}\oplus\ell_1(\mathtt{int}).\mathtt{end}$ and $L_4 = \mathbf{A}\oplus\ell_2(\mathtt{int}).\mathtt{end}$.

   We have $L_3 \not\bowtie L_4$, since the compatibility relation is not defined over any send prefixes.

3. Let $L_5 = \mathbf{A}\& \begin{Bmatrix} \ell_1(\mathtt{int}).\mathtt{end} \\ \ell_2(\mathtt{int}).\mathtt{end} \end{Bmatrix}$ and $L_6 = \mathbf{A}\& \begin{Bmatrix} \ell_3(\mathtt{int}).\mathtt{end} \\ \ell_2(\mathtt{int}).\mathtt{end} \end{Bmatrix}$.

   We have $L_5 \not\bowtie L_6$, since both receive types have an overlapping label: ℓ_2[6]. ⌟

Recall that we dinstinguish local types from local *pre*-types, by designating that local types are those obtained via projection. This distinction is not in use for the original theory (§ 2), but introduced in our extended theory, since there are local pre-types that are syntactically correct, but not obtainable from projection of any global type. An example of a local pre-type, but not a local type, can arise from not satisfying the compatibility relation: $\sum \begin{Bmatrix} \ell_1(\mathtt{int}).\mathbf{A}\oplus\ell_1(\mathtt{int}).\mathtt{end} \\ \ell_2(\mathtt{int}).\mathbf{A}\oplus\ell_2(\mathtt{int}).\mathtt{end} \end{Bmatrix}$.

This compatibility relation is reminiscent of the mergeability relation [YDB+10] (mentioned in § 2.3), and we indeed draw inspiration from the early works. Our compatibility relation may appear as restrictive in the sense that we require compatible local types to have completely *disjoint* receive actions. However, it is important to note that we use this relation as a side condition in projection, and the projection onto a third role is put together by a choice of silent prefixes, and one may draw some relevance to the choice operator. In contrast, the original theory uses a merge operator, and uses no silent prefixes — the choice is embedded and hoisted in the merged type.

We now return to the remaining cases of projection. If the type under projection G is a recursive type $\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$, the projection preserves the recursion construct if the projected role is in the inner protocol (i.e. $\mathbf{r} \in G'$). Moreover, depending on whether the projected role $\mathbf{p}$ has knowledge of the recursion variable x (by checking whether $\mathbf{p} \in \mathbb{P}$), the resulting local type would either take the unrestricted form $\mu\mathbf{t}\,(x^{\omega} := E : T).L$, or the irrelevant form $\mu\mathbf{t}\,(x^{0} : T).L$. For roles that do not participate in the inner global type, or the cases where the projection would not result in a contractive local type (cf. Def. 2.11, Page 16), projection results in $\mathtt{end}$. (Tilde denotes a possibly empty sequence of recursion declarations.)

If the type under projection G is a type variable $\mathbf{t}\,\langle x := E\rangle$, projection also preserves the type variable construct: we look up the type variable context Φ, and produce a construct consistent with the recursion declaration, namely $\mathbf{t}\,\langle x := E\rangle$ or $\mathbf{t}$, by checking whether the projected role has knowledge of the recursion variable. Note that the use of a type variable context ensures that global types with unbound type variables are not projectable.

[6]In the original MPST theory, the (unrefined) local types are *mergeable*, despite the overlapping ℓ_2 label. We motivate the need of disjointness when we introduce semantics.

Finally, the projection of a terminated global type $\mathtt{end}$ is a terminated local type $\mathtt{end}$.

For convenience, we write $\langle \Gamma; G\rangle \upharpoonright \mathbf{r} = \langle \Sigma; L\rangle$ for the projection of a global context and type (closed with regards to type variables) $\langle \Gamma; G\rangle$, i.e. $\Gamma \upharpoonright \mathbf{r} = \Sigma$ and $G \upharpoonright_{\emptyset} \mathbf{r} = L$.

Example 3.20 (Projection of Global Types of Ex. 3.4 (1))**.** We draw attention to the projection of G_1 to $\mathbf{C}$, under the empty context $\emptyset$.

$$\langle \emptyset; G_1\rangle \upharpoonright \mathbf{C} = \langle \emptyset; \mathsf{Fst}(x : \mathtt{int}).\mathbf{B}\&\mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathbf{D}\oplus\mathsf{Trd}(z : \mathtt{int}\{x = z\}).\mathtt{end}\rangle\,.$$

We note that the local type for $\mathbf{C}$ has a silent prefix $\mathsf{Fst}(x : \mathtt{int})$. The silent prefix binds the variable x in the continuation, and adds its type to the 'local knowledge' of the role $\mathbf{C}$, yet the actual value of x is unknown (explained in detail later). ⌟

Lemma 3.21 (Inversion of Projection)**.** *Let $G \upharpoonright_{\Phi} \mathbf{p} = L$. If L is of form:*

1. $\mathbf{q}\oplus\{\ell_i(x_i : T_i).L'_i\}_{i\in I}$, *then G is of form* $\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G'_i\}_{i\in I}$, *and* $G'_i \upharpoonright_{\Phi} \mathbf{p} = L'_i$;
2. $\mathbf{q}\&\{\ell_i(x_i : T_i).L'_i\}_{i\in I}$, *then G is of form* $\mathbf{q} \to \mathbf{p}\{\ell_i(x_i : T_i).G'_i\}_{i\in I}$, *and* $G'_i \upharpoonright_{\Phi} \mathbf{p} = L'_i$;
3. $\sum\{\ell_i(x_i : T_i).L'_i\}_{i\in I}$, *then G is of form* $\mathbf{s} \to \mathbf{t}\{\ell_i(x_i : T_i).G'_i\}_{i\in I}$, $\mathbf{p} \notin \{\mathbf{s}, \mathbf{t}\}$, *and* $G'_i \upharpoonright_{\Phi} \mathbf{p} = L'_i$;
4. $\mu\mathbf{t}\,(x^{\omega} := E : T).L'$, *then G is of form* $\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$, $\mathbf{p} \in \mathbb{P}$, *and* $G' \upharpoonright_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{p} = L'$;
5. $\mu\mathbf{t}\,(x^{0} : T).L'$, *then G is of form* $\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$, $\mathbf{p} \notin \mathbb{P}$, *and* $G' \upharpoonright_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{p} = L'$. ⌟

Proof. Follows directly from the definition of global type projection. (Fig. 3.3). □

Lemma 3.22 (Projection Commutes with Unfolding Recursion)**.** *If* $(\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G) \upharpoonright_{\Phi} \mathbf{r} = \mu\mathbf{t}\,(x^{\omega} := E : T).L$ *or* $(\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G) \upharpoonright_{\Phi} \mathbf{r} = \mu\mathbf{t}\,(x^{0} : T).L$, *then* $(G[\mu\mathbf{t}\,(x : T).G/\mathbf{t}]) \upharpoonright_{\Phi} \mathbf{r} = L[\mu\mathbf{t}\,(x : T).L/\mathbf{t}]$. ⌟

Proof. By induction on the definition of global type projection (Fig. 3.3). □

Well-formedness of Global Types

We now define the well-formedness of a global type G in Def. 3.23. In a nutshell, a well-formed global type G must be *projectable* onto all participants $\mathrm{pts}(G)$ in the global type, as required in the original MPST theory (cf. § 2.3). Moreover, we use well-formedness judgements of form $\Phi; \Gamma \vdash G$ gty to say a global type G is syntactically well-formed under the type variable context Φ and the global typing context Γ. The syntactic well-formedness judgements require payload refinement types inside the given global type to be well-formed, and expressions for recursion variables to be well-typed.

$$\frac{\forall i \in I: \quad \forall \mathbf{r} \in (\bigcup_{i \in I} \mathrm{pts}(G_i) \cup \{\mathbf{p}, \mathbf{q}\}) : (\Gamma \upharpoonright \mathbf{r}) \vdash T_i \text{ ty} \qquad \Phi; \Gamma + {x_i}^{\{\mathbf{p},\mathbf{q}\}} : T_i \vdash G_i \text{ gty}}{\Phi; \Gamma \vdash (\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I}) \text{ gty}} \text{[G-WF-Comm]}$$

$$\frac{\emptyset \neq \mathbb{P} \subseteq \mathrm{pts}(G) \qquad \forall \mathbf{p} \in \mathbb{P} : \begin{Bmatrix} (\Gamma \upharpoonright \mathbf{p}) \vdash T \text{ ty} \\ (\Gamma \upharpoonright \mathbf{p}) \vdash E : T \end{Bmatrix} \qquad \Phi, \mathbf{t} : \langle x^{\mathbb{P}}, T \rangle; \Gamma + x^{\mathbb{P}} : T \vdash G \text{ gty}}{\Phi; \Gamma \vdash (\mu \mathbf{t}\, (x^{\mathbb{P}} := E : T).G) \text{ gty}} \text{[G-WF-Rec]}$$

$$\frac{}{\Phi; \Gamma \vdash \mathtt{end} \text{ gty}} \text{[G-WF-End]} \qquad \frac{\Phi(\mathbf{t}) = \langle x^{\mathbb{P}}, T \rangle \qquad \forall \mathbf{p} \in \mathbb{P} : (\Gamma \upharpoonright \mathbf{p}) \vdash E : T}{\Phi; \Gamma \vdash \mathbf{t}\, \langle x := E \rangle \text{ gty}} \text{[G-WF-Var]}$$

Figure 3.4: Syntactic Well-formedness Rules for Global Types $\boxed{\Phi; \Gamma \vdash G \text{ gty}}$

Definition 3.23 (Well-formed Global Types)**.** A global type G is well-formed under a typing context Γ, i.e. $\langle \Gamma; G \rangle$ is well-formed, if:

1. well-formedness judgement holds under the empty type variable context and the promoted global typing context: $\emptyset; \Gamma^+ \vdash G$ gty (given in Fig. 3.4); and,
2. for all roles in the global type $\mathbf{r} \in G$, the projection $\langle \Gamma; G \rangle \upharpoonright \mathbf{r}$ is defined.

We say a global type G is well-formed, if $\langle \emptyset; G \rangle$ is well-formed. ⌟

The syntactic well-formedness rules are defined inductively. Rule [G-WF-End] states all terminated global types end are syntactically well-formed. Rule [G-WF-Comm] requires each branch $i \in I$ in a communication $\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I}$ have a well-formed refinement type T_i under the projected local typing context, as well as a well-formed continuation G_i with the payload variable x_i added to the typing context.

Rule [G-WF-Rec] requires a recursive global type $\mu \mathbf{t}\, (x^{\mathbb{P}} := E : T).G$ to have a non-empty set $\mathbb{P}$, while being a subset of the participants of the inner global type G. We require at least one role (i.e. a non-empty set $\mathbb{P}$) to know the value of the recursion variable x, so that the expression E is type-checked against the prescribed type T at least once. This requirement ensures that we cannot inject an empty type T with an empty participant set $\mathbb{P}$. In addition, for each role inside the set $\mathbb{P}$, the projected local typing context must be able to validate the well-formedness of refinement type T, and assign that refinement type T to the initialisation expression E. By doing so, we ensure that the initialisation expression E can be used safely by a projected local type, in particular, the expression E does not involve any variable without current knowledge. Finally, the inner global type G is also syntactically well-formed with the recursion variable x added to the typing context, and type variable $\mathbf{t}$ added to the type variable context.

Rule [G-WF-Var] imposes a similar check to a type variable $\mathbf{t}\, \langle x := E \rangle$. The recursion variable entry is retrieved from the type variable context Φ, and all roles in the set $\mathbb{P}$ must have local typing contexts that can assign the refinement type T to the update expression E.

We show some examples of well-formed and ill-formed (i.e. not well-formed) global types.

Example 3.24 (Well-formed and Ill-formed Global Types)**.**

1. Global types in Ex. 3.4 are all well-formed.

2. Let $G_4 = \mu \mathbf{t}\,(x^{\{\mathsf{A},\mathsf{B}\}} := 0 : \mathtt{int}).\mathsf{A} \to \mathsf{B} : \mathsf{Msg}(y : \mathtt{int}).\mathbf{t}'\ \langle x := x+1 \rangle$

   G_4 is not well-formed, since the type variable $\mathbf{t}'$ is not bound.

3. Let $G_5 = \mu \mathbf{t}\,(x^{\emptyset} := 0 : \mathtt{int}).\mathbf{t}\ \langle x := x+1 \rangle$

   G_5 is not well-formed, since the type is not contractive. Despite the recursion variable x is updated at each unfold, there is no actionable prefix in the type.

4. Let $G_6 = \mu \mathbf{t}\,(x^{\{\mathsf{A}\}} := 0 : \mathtt{int}\{x > 0\}).\mathsf{A} \to \mathsf{B} : \mathsf{Msg}(y : \mathtt{int}).\mathbf{t}\ \langle x := x+1 \rangle$

   G_6 is not well-formed, since the initial value 0 for x does not carry the type $x : \mathtt{int}\{x > 0\}$.

   Let $G_6' = \mu \mathbf{t}\,(x^{\{\mathsf{C}\}} := 0 : \mathtt{int}\{x \geq 0\}).\mathsf{A} \to \mathsf{B} : \mathsf{Msg}(y : \mathtt{int}).\mathbf{t}\ \langle x := x+1 \rangle$

   G_6' is not well-formed, since the role C does not participate in the recursion.

   Let $G_6'' = \mu \mathbf{t}\,(x^{\emptyset} := 0 : \mathtt{int}\{x \geq 0\}).\mathsf{A} \to \mathsf{B} : \mathsf{Msg}(y : \mathtt{int}).\mathbf{t}\ \langle x := x+1 \rangle$

   G_6'' is not well-formed, since the recursion variable x is not located in any role.

5. Let $G_7 = \mu \mathbf{t}\,(x^{\{\mathsf{C}\}} := 0 : \mathtt{int}\{x \geq 0\}).\mathsf{A} \to \mathsf{B} : \mathsf{Msg}(y : \mathtt{int}).\mathsf{B} \to \mathsf{C} : \mathsf{Msg}(z : \mathtt{int}).\mathbf{t}\ \langle x := x+1 \rangle$

   G_7 is not well-formed, since the role C does not know the value of x, yet C needs to use the expression $x+1$ in the recursion variable update.

6. Let $G_8 = \mathsf{A} \to \mathsf{B} \left\{ \begin{array}{l} \mathsf{Foo}(x : \mathtt{int}).\mathsf{C} \to \mathsf{D} : \mathsf{Baz}(z : \mathtt{int}).\mathtt{end} \\ \mathsf{Bar}(y : \mathtt{int}).\mathsf{C} \to \mathsf{D} : \mathsf{Baz}(z : \mathtt{int}).\mathtt{end} \end{array} \right\}$

   G_8 is not well-formed, since projection onto role C and D are not defined (sending prefixes are not compatible, see Item 2 of Ex. 3.19). The unrefined form of this global type *is* well-formed under the original MPST theory.

   Let $G_8' = \mathsf{C} \to \mathsf{D} : \mathsf{Baz}(z : \mathtt{int}).\mathsf{A} \to \mathsf{B} \left\{ \begin{array}{l} \mathsf{Foo}(x : \mathtt{int}).\mathtt{end} \\ \mathsf{Bar}(y : \mathtt{int}).\mathtt{end} \end{array} \right\}$

   G_8' is well-formed. This global type is isomorphic to G_8 under the original MPST theory.

7. $G_9 = \mathsf{A} \to \mathsf{B} : \mathsf{Foo}(x : \mathtt{int}).\mathsf{C} \to \mathsf{D} : \mathsf{Baz}(z : \mathtt{int}\{x\}).\mathtt{end}$

   G_9 is not well-formed, since the refinement type of x is not well-formed (refinement expression should be a boolean expression). ⌟

Lemma 3.25 (Weakening of Well-formedness)**.** *If* $\Phi; \Gamma \vdash G$ gty, *then* $\Phi; \Gamma, x^{\mathbb{P}} : T \vdash G$ gty. ⌟

Proof. By induction on the derivation of well-formedness judgement (Fig. 3.4), using Lem. 3.11 where necessary. □

Lemma 3.26 (Recursion Substitution Lemma of Well-formedness)**.** *If* $\Phi, \mathbf{t} : \langle x^{\mathbb{P}}, T\rangle; \Gamma \vdash G$ gty *and* $\Phi; \Gamma \vdash \mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$ gty, *then* $\Phi; \Gamma \vdash G[\mu\mathbf{t}\,(x : T).G'/\mathbf{t}]$ gty. ⌟

Proof. By induction on the derivation of well-formedness judgement $\Phi, \mathbf{t} : \langle x^{\mathbb{P}}, T\rangle; \Gamma \vdash G$ gty (Fig. 3.4). □

In this section, we have given the definitions of projection and well-formedness of global types. We are now ready to give the semantics of refined global types in the next section.

3.5 Labelled Transition System Semantics of Global and Local Types

Similar to the original MPST semantics (§ 2.4), we define a Labelled Transition System (LTS) semantics of refined global and local types. In this section, we first show the semantics of global types, followed by the semantics of local types. The relation between two semantics will follow in the next section (§ 3.6).

Actions (Labels) of the Transition System

We define a synchronous (rendez-vous) semantics for our transition system: we define the label in the LTS as $\alpha ::= \mathbf{p} \to \mathbf{q} : \ell(x : T)$, a message from role $\mathbf{p}$ to $\mathbf{q}$ with label ℓ carrying a value named x with type T. Due to the synchronous nature, we define $\mathrm{subj}(\alpha) = \{\mathbf{p}, \mathbf{q}\}$ to be the subjects of the action α, namely the two roles in the action.

When defining the labels and the semantics, we are more concerned with the refinement types of the payloads — the refinement type T is included in the transition label α. In contrast, we are not interested in the payload values (as long as they type-check against the prescribed type) and leave this matter as an implementation detail (to be explained in § 4).

States of the Transition System

Different from the original LTS semantics [DY13] (Def. 2.15), we include a global typing context Γ in the semantics along with the global type G, to keep track of variables and the knowledge of the variables. Therefore, the transitions will be given in the form of $\langle \Gamma; G\rangle \xrightarrow{\alpha} \langle \Gamma'; G'\rangle$.

3.5.1 Global Types Transitions

We give the transition rules of global types (along with global typing contexts) in Fig. 3.5.

Rule [G-Pfx] allows the reduction of an action in the *prefix* position of a message transmission global type. An action α, matching the definition in set $j \in I$ defined in the prefix, allows the correspondent continuation (with index j) to be selected: the global type after reduction is the matching continuation G_j and the global typing context Γ is expanded to contain the

$$\frac{j \in I}{\left\langle \Gamma; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i\in I} \right\rangle \xrightarrow{\mathbf{p}\to\mathbf{q}:\ell_j(x_j:T_j)} \left\langle \Gamma + x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j; G_j \right\rangle} \text{[G-Pfx]}$$

$$\frac{\{\mathbf{p},\mathbf{q}\} \cap \mathrm{subj}(\alpha) = \varnothing \quad \left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle}{\langle \Gamma; \mathbf{p} \to \mathbf{q} : \ell(x : T).G' \rangle \xrightarrow{\alpha} \langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x : T).G'' \rangle} \text{[G-Cnt]}$$

$$\frac{\left\langle \Gamma + x^{\mathbb{P}} : T; G[\mu\mathbf{t}\,(x : T).G/\mathbf{t}] \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle}{\langle \Gamma; \mu\mathbf{t}\,(x^{\mathbb{P}} \coloneqq E : T).G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle} \text{[G-Rec]}$$

Figure 3.5: Transition Rules for Global Types $\boxed{\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle}$

entry $x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j$ specified in the action α, indicating the sending role $\mathbf{p}$ and receiving role $\mathbf{q}$ have knowledge of the variable after the transmission.

Rule [G-Cnt] allows the reduction of an action α that is *causally independent* of the prefix action in a message transmission global type. The subjects of this action α here must be disjoint from the roles involved in the prefix of the global type. As is the case of the original MPST semantics, if the continuation G' of the global type can make a reduction of that action α to a global type G'', then the same reduction can be made under the prefix.

However, we need to pay extra attention to the typing contexts. When reducing the continuation, we expand the global typing context with the variable of the prefix action, using the hat variant $\widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T$ to indicate persepective knowledge. This expansion ensures that relevant information obtainable from the prefix message is not lost when performing reduction, signifying that the roles would know the value in the future. When the communication of the prefix message occurs, the corresponding entry would be expanded to become current knowledge, as we show in Ex. 3.27. We use the resulting typing context after the reduction of continuation Γ' as the resulting typing context for the entire global type.

A notable change with respect to the original semantics is that this rule applies only when the communication prefix is a singleton (i.e. $|I| = 1$). We will explain its implications in Rem. 3.28.

Rule [G-Rec] allows the reduction of a recursive type by unfolding the type once. Since the global type after substituting the type variable is open (cf. Ex. 3.3), we add the recursion variable entry $x^{\mathbb{P}} : T$ into the global context Γ. Recall that we refresh all the variables inside the recursion for a fresh name (cf. § 3.2, Page 30), this is to ensure that variables across different iterations have distinct names, and no cross-iteration confusion can occur. Suppose the global type in Item 2 of Ex. 3.4 is recursive, and we do keep the same variable name x for the number across iterations. Suppose in one iteration, the number x is positive, the typing context would contain an entry $\_{}_1 : \mathtt{unit}\{x > 0\}$ (where _ is a placeholder variable name for the payload in the message), following a Positive message. Suppose we do not perform the refreshing of variables, and in another iteration, the number x is negative (instead of using a refreshed name x'), adding an entry $\_2 : \mathtt{unit}\{x < 0\}$ following a $\mathsf{Negative}$ message would cause a contradiction in the typing context, due to confusion across different iterations of recursion.

Example 3.27 (Global Type Reductions)**.** We demonstrate two reduction paths for a global type

$$G = \mathbf{A} \to \mathbf{B} : \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathbf{C} \to \mathbf{D} : \mathsf{Hola}(y : \mathtt{int}\{y > x\}).\mathtt{end}.$$

Note that the two messages are not causally related (they have disjoint subjects). We have the following two reduction paths of $\langle \emptyset; G\rangle$ (omitting payload in LTS actions for simplicity):

$$\begin{array}{rl}
 & \langle \emptyset; G\rangle \\
\text{[G-Pfx]} \xrightarrow{\mathbf{A}\to\mathbf{B}:\mathsf{Hello}} & \left\langle x^{\{\mathbf{A},\mathbf{B}\}} : \mathtt{int}\{x < 0\}; \mathbf{C} \to \mathbf{D} : \mathsf{Hola}(y : \mathtt{int}\{y > x\}).\mathtt{end}\right\rangle \\
\text{[G-Pfx]} \xrightarrow{\mathbf{C}\to\mathbf{D}:\mathsf{Hola}} & \left\langle x^{\{\mathbf{A},\mathbf{B}\}} : \mathtt{int}\{x < 0\}, y^{\{\mathbf{C},\mathbf{D}\}} : \mathtt{int}\{y > x\}; \mathtt{end}\right\rangle
\end{array}$$

$$\begin{array}{rl}
 & \langle \emptyset; G\rangle \\
\text{[G-Cnt]} \xrightarrow{\mathbf{C}\to\mathbf{D}:\mathsf{Hola}} & \left\langle \widehat{x}^{\{\mathbf{A},\mathbf{B}\}} : \mathtt{int}\{x < 0\}, y^{\{\mathbf{C},\mathbf{D}\}} : \mathtt{int}\{y > x\}; \mathbf{A} \to \mathbf{B} : \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathtt{end}\right\rangle \\
\text{[G-Pfx]} \xrightarrow{\mathbf{A}\to\mathbf{B}:\mathsf{Hello}} & \left\langle x^{\{\mathbf{A},\mathbf{B}\}} : \mathtt{int}\{x < 0\}, y^{\{\mathbf{C},\mathbf{D}\}} : \mathtt{int}\{y > x\}; \mathtt{end}\right\rangle
\end{array}$$

In the second reduction path, we note that the action $\mathbf{C} \to \mathbf{D} : \mathsf{Hola}$ fires before the prefix action $\mathbf{A} \to \mathbf{B} : \mathsf{Hello}$. To keep the reference to the variable x in the type of y, we rely on the entry of $\widehat{x}$ (under a hat) in the global typing context. Moreover, we observe that the entry is expanded to have the hat removed after the transmission of the Hello message with the variable x. ⌟

Remark 3.28 (Alternative for Rule [G-Cnt]). For well-formed global types under typing contexts, the following rule (as appeared in [ZFH+20], adapted), seemingly a generalisation of rule [G-Cnt], is equivalent to rule [G-Cnt].

$$\frac{\{\mathbf{p}, \mathbf{q}\} \cap \mathrm{subj}(\alpha) = \emptyset \quad \forall i \in I : \ \left\langle \Gamma + \widehat{x}_i^{\{\mathbf{p},\mathbf{q}\}} : T_i; G'\right\rangle \xrightarrow{\alpha} \langle \Gamma'; G''\rangle}{\left\langle \Gamma; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i'\}_{i\in I}\right\rangle \xrightarrow{\alpha} \left\langle \Gamma'; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i''\}_{i\in I}\right\rangle}\ \text{[G-Cnt-Alt]}$$

Rule [G-Cnt] does not restrict the top-level global type to be a singleton choice, but instead requires all continuations to be reduced to a single global typing context. However, we note that any global type matching the shape of rule [G-Cnt-Alt] is only projectable when $|I| = 1$, since the local type compatibility relation (Def. 3.18) does not have rules for sending prefixes, cf. Item 6 of Ex. 3.24. ⌟

Lemma 3.29 (Deterministic Global Type Reductions)**.** *If* $\langle \Gamma; G\rangle \xrightarrow{\alpha} \langle \Gamma_1; G_1\rangle$ *and* $\langle \Gamma; G\rangle \xrightarrow{\alpha} \langle \Gamma_2; G_2\rangle$*, then* $\langle \Gamma_1; G_1\rangle = \langle \Gamma_2; G_2\rangle$*.* ⌟

Proof. By induction on global type reduction rules (Fig. 3.5). □

Lemma 3.30 (Subjects are Participants). *If* $\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$*, then any subject of the action* $\mathbf{p} \in \mathrm{subj}(\alpha)$ *must be a participant* $\mathbf{p} \in G$ *of the global type* G*.* ⌟

Proof. By induction on global type reduction rules (Fig. 3.5). □

Lemma 3.31 (Global Knowledge). *Suppose* $\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$*.*

1. *If* $\widehat{x}^{\mathbb{P}} : T \in \Gamma$*, then* $x^{\mathbb{P}} : T \in \Gamma'$ *when* $\alpha = \mathbf{p} \to \mathbf{q} : \ell(x : T)$*, otherwise* $\widehat{x}^{\mathbb{P}} : T \in \Gamma'$*;*
2. *If* $x^{\mathbb{P}} : T \in \Gamma$*, then* $x^{\mathbb{P}} : T \in \Gamma'$*;*
3. *Let* $\alpha = \mathbf{p} \to \mathbf{q} : \ell(x : T)$*, then* $x^{\{\mathbf{p}, \mathbf{q}\}} : T \in \Gamma'$*.* ⌟

Proof. By induction on the derivation of global type reductions (Fig. 3.5), noting that $+$ only appends or updates an entry without removing. □

We show in Thm. 3.32 that well-formedness of global types is preserved under reductions.

Theorem 3.32 (Preservation of Well-formedness). *If* G *is a well-formed global type under a global typing context* Γ *, and* $\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$*, then the reductum* $\langle \Gamma'; G' \rangle$ *is also well-formed.* ⌟

Proof. By induction on the derivation of global type reductions $\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$ (Fig. 3.5).

- Case [G-Pfx]: We have

  $$\langle \Gamma; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I} \rangle \xrightarrow{\mathbf{p} \to \mathbf{q} : \ell_j(x_j : T_j)} \left\langle \Gamma + x_j^{\{\mathbf{p}, \mathbf{q}\}} : T_j; G_j \right\rangle$$

  with $j \in I$.

  By inverting [G-WF-Comm], we have $\varnothing; \Gamma^+ + x_j^{\{\mathbf{p}, \mathbf{q}\}} : T_j \vdash G_j$ gty, as required by Item 1.

  For all $\mathbf{r} \in \{\mathbf{p}, \mathbf{q}\}$, or $\mathbf{r} \in G_i$ for some $i \in I$, we know that the continuation L_j is defined, and given by $G' \upharpoonright_{\varnothing} \mathbf{r}$, as required by Item 2.

- Case [G-Cnt]: We have

  $$\langle \Gamma; \mathbf{p} \to \mathbf{q} : \ell(x : T).G' \rangle \xrightarrow{\alpha} \langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x : T).G'' \rangle$$

  with $\{\mathbf{p}, \mathbf{q}\} \cap \mathrm{subj}(\alpha) = \varnothing$ and $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p}, \mathbf{q}\}} : T; G' \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle$.

  By inverting [G-WF-Comm], we have $\varnothing; \Gamma^+ + x^{\{\mathbf{p}, \mathbf{q}\}} : T \vdash G'$ gty.

  To invoke the inductive hypothesis, we need to show that $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p}, \mathbf{q}\}} : T; G' \right\rangle$ is well-formed: Item 1 is satisfied because $(\Gamma + \widehat{x}^{\{\mathbf{p}, \mathbf{q}\}} : T)^+ = \Gamma^+ + x^{\{\mathbf{p}, \mathbf{q}\}} : T$ (Lem. 3.10), and this is obtained by inversion; Item 2 is satisfied by expanding the definition of projection.

  Applying inductive hypothesis, we have that $\varnothing; (\Gamma')^+ \vdash G''$ gty (Item 1) and projectability of G'' (Item 2).

Combined with other premises obtained from inverting the premise, we can apply [G-WF-Comm] as required, hence Item 1 is satisfied.

For $\mathbf{p}$ and $\mathbf{q}$, projectability of $\mathbf{p} \to \mathbf{q} : \ell(x : T).G''$ follows from the projectability of G''. For $\mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\}$, the result follows from a similar reasoning, exploiting the fact that $\bowtie$ holds trivially with a singleton continuation. Hence Item 2 is satisfied.

- Case [G-Rec]: We have

$$\left\langle \Gamma; \mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$$

  with $\langle \Gamma + x^{\mathbb{P}} : T; G[\mu\mathbf{t}\,(x : T).G/\mathbf{t}] \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$.

  Item 1: By inverting [G-WF-Rec], we have $\mathbf{t} : \langle x^{\mathbb{P}}, T \rangle; (\Gamma^{+} + x^{\mathbb{P}} : T) \vdash G$ gty

  Notice that we have ruled out degenerate forms of recursion (see Page 30), the form of the inner type G must be a communication: Item (1) rules out immediate nested recursion, Item (2) rules out $\mathtt{end}$ or $\mathbf{t}'\,\langle ... \rangle$ since the type variable $\mathbf{t}$ is not used; and contractiveness requirement rules out $\mathbf{t}\,\langle ... \rangle$. By Lem. 3.26, we have $\emptyset; \Gamma^{+} + x^{\mathbb{P}} : T \vdash G[\mu\mathbf{t}\,(x : T).G/\mathbf{t}]$ gty.

  Item 2 follows from Lem. 3.22. □

Moreover, we show in Thm. 3.33 that non-terminated well-formed global types are always reducible.

Theorem 3.33 (Global Reduction Progress). *If G is a well-formed global type under a global typing context Γ, and G is not terminated $G \neq \mathtt{end}$, then there exists $\langle \Gamma'; G' \rangle$ and α such that $\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$.* ⌟

Proof. We consider the form of the global type G.

If G is a communication, we can always reduce via [G-Pfx]. We note that the resulting typing context $\Gamma + x_j^{\{\mathbf{p}, \mathbf{q}\}} : T_j$ must be defined, which we rely on the promoted typing context being defined by inversion of well-formedness rule [G-WF-Comm].

If G is a recursion, then its unfolding must be well-formed (Lem. 3.26), and it must be a communication (so that it can reduce via [G-Pfx] in the premise), and we can apply [G-Rec]. We note that the resulting typing context $\Gamma + x^{\mathbb{P}} : T$ must be defined, which we rely on the promoted typing context being defined by inversion of well-formedness rule [G-WF-Rec].

G cannot be a type variable on its own, since well-formedness required the type to be closed with regards to type variables. □

3.5.2 Local Type Transitions

After defining semantics of global types, we now define the LTS semantics of local types. Similar to what we do in global type semantics, we include local typing contexts Σ in the state of the transition system. Therefore, the judgements of local LTS reductions have form $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$. The transitions are shown in Fig. 3.6.

$$\frac{}{\langle \Sigma; \mu\mathbf{t}\,(x^{\omega} := E : T).L \rangle \rightsquigarrow \langle \Sigma + x^{\omega} : T; L[\mu\mathbf{t}\,(x : T).L/\mathbf{t}] \rangle}\;[\text{L-Pre-Rec-}\omega]$$

$$\frac{}{\langle \Sigma; \mu\mathbf{t}\,(x^{0} : T).L \rangle \rightsquigarrow \langle \Sigma + x^{0} : T; L[\mu\mathbf{t}\,(x : T).L/\mathbf{t}] \rangle}\;[\text{L-Pre-Rec-0}]$$

$$\frac{j \in I}{\langle \Sigma; \sum \{\ell_i(x_i : T_i).L_i\}_{i \in I} \rangle \rightsquigarrow \langle \Sigma + x_j^{0} : T_j; L_j \rangle}\;[\text{L-Pre-Sil}]$$

$$\frac{j \in I}{\langle \Sigma; \mathbf{q} \oplus \{\ell_i(x_i : T_i).L_i\}_{i \in I} \rangle \xrightarrow{\mathbf{p} \to \mathbf{q} : \ell_j(x_j : T_j)} \langle \Sigma + x_j^{\omega} : T_j; L_j \rangle}\;[\text{L-Send}]$$

$$\frac{j \in I}{\langle \Sigma; \mathbf{q} \& \{\ell_i(x_i : T_i).L_i\}_{i \in I} \rangle \xrightarrow{\mathbf{q} \to \mathbf{p} : \ell_j(x_j : T_j)} \langle \Sigma + x_j^{\omega} : T_j; L_j \rangle}\;[\text{L-Recv}]$$

$$\frac{\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma''; L'' \rangle \qquad \langle \Sigma''; L'' \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle}{\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle}\;[\text{L-Pre}]$$

Figure 3.6: Transition Rules for Local Types $\boxed{\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle}$ from the Perspective of Role $\mathbf{p}$ and Silent Transition Rules for Local Types $\boxed{\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma'; L' \rangle}$

Rules [L-Send] and [L-Recv] are the standard for local type transitions. Take the perspective of a participant **p**, a sending (resp. receiving) local type towards (resp. from) a peer **q** is reduced by an action of $\mathbf{p} \to \mathbf{q} : \ell_j(x_j : T_j)$ (resp. $\mathbf{q} \to \mathbf{p} : \ell_j(x_j : T_j)$), where the label ℓ_j, variable x_j, refinement type T_j are chosen among the available choices ($j \in I$). After reduction, the local typing context Σ is expanded with an entry with unrestricted multiplicity $x_j^\omega : T_j$.

The unusual aspect of the transitions is that we also use transitions of form $\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma'; L' \rangle$, which we call *silent* transitions. This form of transitions looks similar to τ-transitions in process algebra, and indeed both are meant to model non-observable actions. The difference is that, in our semantics, the silent actions are not taken *spontaneously*, but *reactively* before an observable action in rule [L-Pre]. This semantics can also be understood as a variation of *angelic non-determinism*[7] [BW81], where we prefer a reduction path that leads to a desired outcome.

We write $\rightsquigarrow^*$ for the reflexive and transitive closure of the silent reduction relation $\rightsquigarrow$. We explain the rules for silent actions as follows.

Rules [L-Pre-Rec-ω] and [L-Pre-Rec-0] are responsible for unfolding recursive local types with both multiplicities. When doing so, the local typing context is expanded with the recursion variable with the appropriate multiplicity. These two rules, combined with [L-Pre], are similar to the unfolding rule of recursive global types [G-Rec], but presented here in separate. In the syntax of RMPST, we are no longer able identify recursive types and their unfoldings, since the recursion constructs may contain different expressions at declarations and uses. Nonetheless, the unfolding of a recursive type does not produce any observable communication action, and this transition can take place spontaneously.

Rule [L-Pre-Sil] allows a choice in a silent prefix to be consumed, by expanding the local typing context with the variable in the prefix in the irrelevant form $x_j^0 : T_j$ (for $j \in I$). This rule allows participants to obtain the 'latent' knowledge from the non-participating interactions from the given global type. Different from rules [L-Pre-Rec-ω] and [L-Pre-Rec-0], the choice in a silent prefix is *not* a spontaneous action, and only taken *as necessary* before an observable action. Allowing a spontaneous action in silent prefixes may lead to stuck local types, when the wrong choice is made. This is also what motivates the local type compatibility relation (Def. 3.18), where we require each choice in a silent prefix to have distinct (receiving) actions to avoid wrong choices.

We note some interesting properties of silent transitions. Suppose $\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma'; L' \rangle$, all reducible actions of $\langle \Sigma'; L' \rangle$ are also reducible actions of $\langle \Sigma; L \rangle$, because of rule [L-Pre]. Therefore, silent reductions can lead to some form of *simulation* [Mil71]. On the other hand, not all reducible actions of $\langle \Sigma; L \rangle$ are reducible actions of $\langle \Sigma'; L' \rangle$, because of rule [L-Pre-Sil]. Once one of the branches is committed when applying [L-Pre-Sil], reductions arising from all other branches cease to be available.

[7]With thanks to the examiners for pointing out this connection.

Example 3.34 (Local Type Reductions)**.** We project the global type defined in Ex. 3.27 onto the roles A and C and show their corresponding reductions. Recall that

$$G = \mathsf{A} \to \mathsf{B} : \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathsf{C} \to \mathsf{D} : \mathsf{Hola}(y : \mathtt{int}\{y > x\}).\mathtt{end},$$

and there are two reductions paths for the global type G. We first project the global type G to role A:

$$\langle \emptyset; G \rangle \restriction \mathsf{A} = \langle \emptyset; \mathsf{B}{\oplus}\mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathtt{end} \rangle .$$

We can use rule [L-Send] to reduce this local type (omitting payload in LTS actions for simplicity):

$$\langle \emptyset; \mathsf{B}{\oplus}\mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathtt{end} \rangle \xrightarrow{\mathsf{A}\to\mathsf{B}:\mathsf{Hello}} \langle x^{\omega} : \mathtt{int}\{x < 0\}; \mathtt{end} \rangle .$$

We then project the global type G to C:

$$\langle \emptyset; G \rangle \restriction \mathsf{C} = \langle \emptyset; \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathsf{D}{\oplus}\mathsf{Hola}(y : \mathtt{int}\{y > x\}).\mathtt{end} \rangle .$$

We can use rule [L-Pre] (with rules [L-Pre-Sil] and [L-Send]) to reduce this local type (we explicitly write the $\rightsquigarrow$ transition here for extra clarity):

$$\begin{array}{rl} & \langle \emptyset; \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathsf{D}{\oplus}\mathsf{Hola}(y : \mathtt{int}\{y > x\}).\mathtt{end} \rangle \\ \rightsquigarrow & \langle x^{0} : \mathtt{int}\{x < 0\}; \mathsf{D}{\oplus}\mathsf{Hola}(y : \mathtt{int}\{y > x\}).\mathtt{end} \rangle \\ \xrightarrow{\mathsf{C}\to\mathsf{D}:\mathsf{Hola}} & \langle x^{0} : \mathtt{int}\{x < 0\}, y^{0} : \mathtt{int}\{y > x\}; \mathtt{end} \rangle \end{array}$$ ⌟

Remark 3.35 (Reductions for Empty Local Types)**.** Our semantics does not exclude empty local types (Rem. 3.14). We consider emptiness of local types and reductions of local types as two orthogonal issues; as a consequence, empty local types may still reduce, instead of getting stuck whenever a value cannot provided for a given payload type.

This does not invalidate the safety properties of endpoints, since no endpoints can be implemented for an empty local type. ⌟

Lemma 3.36 (Local Knowledge)**.** *Suppose* $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$ *or* $\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma'; L' \rangle$.

1. *If* $x^{0} : T \in \Sigma$, *then* $x^{0} : T \in \Sigma'$;

2. *If* $x^{\hat{0}} : T \in \Sigma$, *then* $x^{\omega} : T \in \Sigma'$ *when* $\alpha = \mathsf{p} \to \mathsf{q} : \ell(x : T)$, *otherwise* $x^{\hat{0}} : T \in \Sigma'$;

3. *If* $x^{\omega} : T \in \Sigma$, *then* $x^{\omega} : T \in \Sigma'$.

Moreover, suppose $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$, *and* $\alpha = \mathsf{p} \to \mathsf{q} : \ell(x : T)$, *then* $x^{\omega} : T \in \Sigma'$. ⌟

Proof. By induction on the derivation of local type reductions (Fig. 3.6), noting that $+$ only appends or updates an entry without removing. □

Lemma 3.37 (Compatible Local Types Reduce Disjointly)**.** *If* $L_1 \bowtie L_2$, *and* $\langle \Sigma_1; L_1 \rangle \xrightarrow{\alpha} \langle \Sigma_1'; L_2 \rangle$, *then* $\langle \Sigma_2; L_2 \rangle \not\xrightarrow{\alpha}$. ⌟

Proof. By induction on local type compatibility relation $\bowtie$ (Def. 3.18).

- Base Case: Let $L_1 = \mathbf{p}\&\{\ell_i(x_i : T_i).L_i\}_{i \in I}$, and $L_2 = \mathbf{p}\&\{\ell'_j(x'_j : T'_j).L'_j\}_{j \in J}$, where $\forall i \in I : \forall j \in J : \ell_i \neq \ell'_j$. Since the labels ℓ_i and ℓ'_j are disjoint, the reducible actions for $\langle \Sigma_1; L_1 \rangle$ are $\alpha_i = \mathbf{q} \to \mathbf{p} : \ell_i(x_i : T_i)$, and none of α_i is a reducible action for $\langle \Sigma_2; L_2 \rangle$.

- Inductive Cases: We prove one of the symmetric cases here. Let $L_1 = \sum\{\ell_i(x_i : T_i).L'_i\}_{i \in I}$, where $\forall i \in I : L'_i \bowtie L_2$. By inductive hypothesis, for any $i \in I$, we know that L'_i has disjoint actions from L_2. Since L_1 can only reduce via rule [L-Pre], the reducible actions of L_1 is the union of reducible actions of L'_i, and the required result follows. □

Lemma 3.38 (Deterministic Local Type Reductions)**.** *If* $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma_1; L_1 \rangle$ *and* $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma_2; L_2 \rangle$, *then* $\langle \Sigma_1; L_1 \rangle = \langle \Sigma_2; L_2 \rangle$. ⌟

Proof. By induction on local type reduction rules (Fig. 3.6).

Rules [L-Send] and [L-Recv] are deterministic, since the action α dictates the reduction outcome. The interesting case is [L-Pre], where we need to examine each silent transitions: rules [L-Pre-Rec-ω] and [L-Pre-Rec-0] are deterministic, but rule [L-Pre-Sil] is only deterministic when the index set is a singleton (i.e. $|I| = 1$). In the case where there are multiple options for the silent transitions in rule [L-Pre-Sil], we exploit the property that they reduce disjointly (Lem. 3.37). □

Lemma 3.39. $\langle \Sigma + x^{\omega} : T; L \rangle$ *and* $\left\langle \Sigma + x^{\hat{0}} : T; L \right\rangle$ *have the same reductions.* ⌟

Proof. First, we observe that possible reducible actions are determined only by the local type L, which is identical here for the two local types under typing contexts.

Second, whether an action can be taken further depends on whether the typing context expansion is defined (Def. 3.7). In the reduction rules (Fig. 3.6), the two forms of expansions used are of multiplicity 0 (in rules [L-Pre-Sil] and [L-Pre-Rec-0]) or ω (in rules [L-Send], [L-Recv], and [L-Pre-Rec-ω]). The possible difference here is when the entry x of differing multiplicity $\hat{0}$ or ω is being expanded: it is not possible to expand with multiplicity 0, and expanding with multiplicity ω would reach the same result of multiplicity ω. □

In this section, we have given the semantics of refined global and local types using labelled transition systems (LTS). We show that well-formedness of global types are preserved under reductions, and well-formed global types enjoy progress properties. We continue next by showing the relation of global and local type semantics.

3.6 Relating Semantics of Global and Local Types

For the top-down design methodology to work, we must show the relation between reductions of global types and collections of projected local types (known as *configurations*). The relation entails that a given global type matches the behaviour of local types projected from that global type, and this connection is preserved under all possible reductions.

As explained previously in § 2.4.3, the main idea of this correspondence is that: if a global type G reduces with an action α to G', a configuration $\mathcal{S}$ associated to G must also be able to reduce with the same action α to $\mathcal{S}'$, and the reductum configuration $\mathcal{S}'$ and the reductum global type G' remain associated; and vice versa for reductions of the configuration $\mathcal{S}$.

We show that global type reductions are matched by configuration reductions, and leave the other direction as a conjecture. Since our refinement type extensions do not change the communication structure, but only refines the payload types, we can erase the refinements to obtain global and local types using only basic types, and rely on the original MPST for these properties, which we explain in § 3.7.

We extend the Labelled Transition System (LTS) semantics to a collection of local types. We first define *configurations* in Def. 3.40, and then reductions of configurations in Def. 3.41.

Definition 3.40 (Configuration)**.** A configuration $\mathcal{S} = \{\langle \Sigma_{\mathbf{r}}; L_{\mathbf{r}} \rangle\}_{\mathbf{r} \in \mathbb{P}}$ is a collection of local types under typing contexts, indexable via participants.

We write $\mathcal{S}(\mathbf{r})$ to denote the entry of role $\mathbf{r}$ in the configuration $\mathcal{S}$. We write $\mathrm{dom}(\mathcal{S})$ to denote the domain of the configuration $\mathcal{S}$, namely a fixed set of roles $\mathbb{P}$. ⌟

Definition 3.41 (Configuration Reductions)**.** The concrete configuration reduction relation $\mathcal{S} \xrightarrow{\alpha} \mathcal{S}'$, where $\alpha = \mathbf{p} \to \mathbf{q} : \ell(x : T)$, holds if

1. $\mathrm{dom}(\mathcal{S}) = \mathrm{dom}(\mathcal{S}')$,
2. $\mathcal{S}(\mathbf{p}) \xrightarrow{\alpha} \mathcal{S}'(\mathbf{p})$ and $\mathcal{S}(\mathbf{q}) \xrightarrow{\alpha} \mathcal{S}'(\mathbf{q})$, and
3. $\mathcal{S}(\mathbf{r}) = \mathcal{S}'(\mathbf{r})$ for all $\mathbf{r} \in (\mathrm{dom}(\mathcal{S}) \setminus \{\mathbf{p}, \mathbf{q}\})$. ⌟

We define the semantics in a synchronous fashion, i.e. sending and receiving actions are not separated. The configuration $\mathcal{S}$ reduces with an action $\alpha = \mathbf{p} \to \mathbf{q} : \ell(x : T)$, if the entries for roles $\mathbf{p}$ and $\mathbf{q}$ both reduce with that action α, and all other entries must remain unchanged.

Another key component is the association between global types and configurations. Before we give the formal definitions, we use an example to demonstrate potential challenges for defining the association.

Example 3.42. We extend the global type in Ex. 3.27 by adding more messages, and let

$$G = \begin{array}{l} \mathbf{A} \to \mathbf{B} : \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathbf{C} \to \mathbf{D} : \mathsf{Hola}(y : \mathtt{int}\{y > x\}). \\ \mathbf{A} \to \mathbf{B} : \mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}).\mathbf{C} \to \mathbf{D} : \mathsf{Adios}(y' : \mathtt{int}\{y' < x'\}).\mathtt{end}. \end{array}$$

We can see that there are two disjoint sequences of reductions of the global type G that can be interleaved freely: one involving roles $\mathbf{A}$ and $\mathbf{B}$ conversing in English, and another one involving roles $\mathbf{C}$ and $\mathbf{D}$ conversing in Spanish.

Suppose the Spanish speakers make the first move, and let

$$G' = \begin{array}{l} \mathbf{A} \to \mathbf{B} : \mathsf{Hello}(x : \mathtt{int}\{x < 0\}) \\ \mathbf{A} \to \mathbf{B} : \mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}).\mathbf{C} \to \mathbf{D} : \mathsf{Adios}(y' : \mathtt{int}\{y' < x'\}).\mathtt{end}. \end{array}$$

We have the global type reduction (using rule [G-Cnt]):

$$\langle \varnothing; G \rangle \xrightarrow{\mathbf{C}\to\mathbf{D}:\mathsf{Hola}} \left\langle \widehat{x}^{\{\mathbf{A},\,\mathbf{B}\}} : \mathtt{int}\{x < 0\}, y^{\{\mathbf{C},\,\mathbf{D}\}} : \mathtt{int}\{y > x\}; G' \right\rangle.$$

The challenge here is to relate the projections of G and G' onto the English speakers:

$$\langle \varnothing; G \rangle \upharpoonright \mathbf{A} = \left\langle \varnothing; \begin{array}{l} \mathbf{B}\oplus\mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathsf{Hola}(y : \mathtt{int}\{y > x\}). \\ \mathbf{B}\oplus\mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}).\mathtt{end} \end{array} \right\rangle$$

$$\left\langle \begin{array}{l} \widehat{x}^{\{\mathbf{A},\,\mathbf{B}\}} : \mathtt{int}\{x < 0\}, \\ y^{\{\mathbf{C},\,\mathbf{D}\}} : \mathtt{int}\{y > x\} \end{array}; G' \right\rangle \upharpoonright \mathbf{A} = \left\langle \begin{array}{l} x^{\hat{0}} : \mathtt{int}\{x < 0\}, \\ y^{0} : \mathtt{int}\{y > x\} \end{array}; \begin{array}{l} \mathbf{B}\oplus\mathsf{Hello}(x : \mathtt{int}\{x < 0\}). \\ \mathbf{B}\oplus\mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}).\mathtt{end} \end{array} \right\rangle$$

Since the message Hola is consumed by the global type reduction, the appropriate silent prefix in the projection disappears after the reduction. While we see an example in Ex. 3.34 where a silent prefix is removed in a silent transition $\rightsquigarrow$, no silent transition is available here.

A slightly different situation arises when we relate the projections of G and G' onto the Spanish speakers:

$$\langle \varnothing; G \rangle \upharpoonright \mathbf{C} = \left\langle \varnothing; \begin{array}{l} \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathbf{D}\oplus\mathsf{Hola}(y : \mathtt{int}\{y > x\}). \\ \mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}).\mathbf{D}\oplus\mathsf{Adios}(y' : \mathtt{int}\{y' > x'\}). \end{array} \right\rangle$$

$$\left\langle \begin{array}{l} \widehat{x}^{\{\mathbf{A},\,\mathbf{B}\}} : \mathtt{int}\{x < 0\}, \\ y^{\{\mathbf{C},\,\mathbf{D}\}} : \mathtt{int}\{y > x\} \end{array}; G' \right\rangle \upharpoonright \mathbf{C} = \left\langle \begin{array}{l} x^{0} : \mathtt{int}\{x < 0\}, \\ y^{\omega} : \mathtt{int}\{y > x\} \end{array}; \begin{array}{l} \mathsf{Hello}(x : \mathtt{int}\{x < 0\}). \\ \mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}). \\ \mathbf{D}\oplus\mathsf{Adios}(y' : \mathtt{int}\{y' < x'\}).\mathtt{end} \end{array} \right\rangle$$

When we reduce the local type of **C** with the Hola message, we obtain:

$$\left\langle \varnothing; \begin{array}{l} \mathsf{Hello}(x : \mathtt{int}\{x < 0\}).\mathbf{D}\oplus\mathsf{Hola}(y : \mathtt{int}\{y > x\}). \\ \mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}).\mathbf{D}\oplus\mathsf{Adios}(y' : \mathtt{int}\{y' < x'\}). \end{array} \right\rangle$$
$$\xrightarrow{\mathbf{C}\to\mathbf{D}:\mathsf{Hola}} \left\langle \begin{array}{l} x^{0} : \mathtt{int}\{x < 0\}, \\ y^{\omega} : \mathtt{int}\{y > x\} \end{array}; \mathsf{Bye}(x' : \mathtt{int}\{x' > 0\}).\mathbf{D}\oplus\mathsf{Adios}(y' : \mathtt{int}\{y' < x'\}). \right\rangle$$

This time, we need to apply a silent transition from the projection of G' to equate the local type after reduction, since the silent prefix Hola is consumed during the local type reduction. ⌟

Recall that in the original MPST theory (§ 2.4.3), the association involves projection and subtyping. In the context of our refined MPST theory, we have not developed a good definition of subtyping relation; in particular, a subtyping relation that goes beyond the natural extension of the subtyping relation over non-refined local types, and takes into consideration the silent prefixes. We leave this as a possible direction of future work. Instead of subtyping, we use a *converging* relation for local types under typing contexts.

The motivation for this relation is demonstrated in Ex. 3.42, where we need to relate the two local types under typing contexts obtained via projection onto role **A**. These situations arise when the global type reduction uses rule [G-Cnt], which has the characteristic that the prefix action of the global type (and also of the local type by projection) is a singleton. We can see a common sequence of *unique* reductions for both local types under typing contexts, with the help of a silent transition, for the one local type under a typing context to converge to the other, as we show in Ex. 3.43. As previously stated, the silent transitions can be extended to some form of simulation[8]. The *converging* relation can be considered as an extended form of silent transitions, where we accept common unique reductions before silent transitions.

Example 3.43 (Path to Convergence)**.** We show a path to convergence for the two local types under typing contexts for role **A** in Ex. 3.42. For the first one (projection from $\langle \emptyset; G\rangle$):

$$\begin{array}{rl} & \left\langle \emptyset; \begin{array}{l} \mathbf{B}\oplus\mathsf{Hello}(x:\mathtt{int}\{x<0\}).\mathsf{Hola}(y:\mathtt{int}\{y>x\}). \\ \mathbf{B}\oplus\mathsf{Bye}(x':\mathtt{int}\{x'>0\}).\mathtt{end} \end{array} \right\rangle \\ \xrightarrow{\mathbf{A}\to\mathbf{B}:\mathsf{Hello}} & \left\langle x^{\omega}:\mathtt{int}\{x<0\}; \begin{array}{l} \mathsf{Hola}(y:\mathtt{int}\{y>x\}). \\ \mathbf{B}\oplus\mathsf{Bye}(x':\mathtt{int}\{x'>0\}).\mathtt{end} \end{array} \right\rangle \\ \rightsquigarrow & \left\langle \begin{array}{l} x^{\omega}:\mathtt{int}\{x<0\}, \\ y^{0}:\mathtt{int}\{y>x\} \end{array} ; \mathbf{B}\oplus\mathsf{Bye}(x':\mathtt{int}\{x'>0\}).\mathtt{end} \right\rangle \end{array}$$

For the second one (projection from $\langle \emptyset; G\rangle$ after reduction of $\mathbf{C} \to \mathbf{D} : \mathsf{Hola}$):

$$\begin{array}{rl} & \left\langle \begin{array}{l} x^{\hat{0}}:\mathtt{int}\{x<0\}, \\ y^{0}:\mathtt{int}\{y>x\} \end{array} ; \begin{array}{l} \mathbf{B}\oplus\mathsf{Hello}(x:\mathtt{int}\{x<0\}). \\ \mathbf{B}\oplus\mathsf{Bye}(x':\mathtt{int}\{x'>0\}).\mathtt{end} \end{array} \right\rangle \\ \xrightarrow{\mathbf{A}\to\mathbf{B}:\mathsf{Hello}} & \left\langle \begin{array}{l} x^{\omega}:\mathtt{int}\{x<0\}, \\ y^{0}:\mathtt{int}\{y>x\} \end{array} ; \mathbf{B}\oplus\mathsf{Bye}(x':\mathtt{int}\{x'>0\}).\mathtt{end} \right\rangle \end{array}$$

We observe that the two local types under typing contexts converge after a common reduction sequence (which may possibly include silent transitions in the end). ⌟

To formalise this idea of converging local types under typing contexts, we first need some auxiliary definitions. We extend the reduction relations from a single action to a (possibly-empty) sequence of actions in Def. 3.44, known as *traces*. In addition, we distinguish *unique* reductions in Def. 3.45, where there is only a single action to reduce.

Definition 3.44 (Trace)**.** A *trace*, ranged over by σ, is a sequence of actions $\sigma = \alpha_1 \cdot \alpha_2 \cdots \alpha_n$, for some natural number $n \in \mathbb{N}$. We write ε for an empty trace (i.e. $n = 0$).

We extend the reductions naturally to traces: $\langle \Sigma_0; L_0\rangle \xrightarrow{\sigma} \langle \Sigma_n; L_n\rangle$ for a trace $\sigma = \alpha_1 \cdot \alpha_2 \cdots \alpha_n$, if for all $i \in \{1..n\} : \langle \Sigma_{i-1}; L_{i-1}\rangle \xrightarrow{\alpha_i} \langle \Sigma_i; L_i\rangle$. (In particular, $\langle \Sigma_0; L_0\rangle \xrightarrow{\varepsilon} \langle \Sigma_0; L_0\rangle$.)

The subjects of a trace $\sigma = \alpha_1 \cdot \alpha_2 \cdots \alpha_n$ are the union of subjects of actions in the trace: $\mathrm{subj}(\sigma) = \bigcup_{i\in\{1..n\}} \mathrm{subj}(\alpha_i)$.

Global type reduction via a trace σ is defined analogously $\langle \Gamma_0; G_0\rangle \xrightarrow{\sigma} \langle \Gamma_n; G_n\rangle$, using $\langle \Gamma_i; G_i\rangle$ in place of $\langle \Sigma_i; L_i\rangle$ respectively. ⌟

[8]The subtyping relation in the original MPST (Def. 2.26) is a simulation.

Definition 3.45 (Unique Reduction). If $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$, and the action α is unique for the local type under a typing context $\langle \Sigma; L \rangle$, i.e. for any action α', we have $\langle \Sigma; L \rangle \xrightarrow{\alpha'}$ implies $\alpha = \alpha'$, then we write $\langle \Sigma; L \rangle \xrightarrow{!\alpha} \langle \Sigma'; L' \rangle$ to denote this *unique reduction* of $\langle \Sigma; L \rangle$.

We extend the unique reductions naturally to traces: $\langle \Sigma_0; L_0 \rangle \xrightarrow{!\sigma} \langle \Sigma_n; L_n \rangle$ for a trace $\sigma = \alpha_1 \alpha_2 \cdots \alpha_n$, if for all $i \in \{1..n\} : \langle \Sigma_{i-1}; L_{i-1} \rangle \xrightarrow{!\alpha_i} \langle \Sigma_i; L_i \rangle$. (In particular, $\langle \Sigma_0; L_0 \rangle \xrightarrow{!\varepsilon} \langle \Sigma_0; L_0 \rangle$.)

Analogously, we define *unique silent reduction* $\overset{!}{\rightsquigarrow}$. If for all $\langle \Sigma'_1; L'_1 \rangle$ and $\langle \Sigma'_2; L'_2 \rangle$ with $\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma'_1; L'_1 \rangle$ and $\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma'_2; L'_2 \rangle$, we have $\langle \Sigma'_1; L'_1 \rangle = \langle \Sigma'_2; L'_2 \rangle$, then the silent transition is unique, written $\langle \Sigma; L \rangle \overset{!}{\rightsquigarrow} \langle \Sigma'_1; L'_1 \rangle$. We write $\overset{!}{\rightsquigarrow}^*$ for the reflexive and transitive closures of $\overset{!}{\rightsquigarrow}$. ⌟

It is important to note that (concrete) local type reductions are deterministic (Lem. 3.38). Therefore, when a local type under a typing context reduces uniquely, the reduction would be totally deterministic. Similarly, when a local type under a typing context has unique silent transitions, the silent reduction would be deterministic.

Lemma 3.46. *If* $\langle \Sigma; L \rangle \overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$ *and* $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma''; L'' \rangle$ *, then* $\langle \Sigma'; L' \rangle \xrightarrow{\alpha} \langle \Sigma''; L'' \rangle$. ⌟

Proof. We analyse the rule used for $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma''; L'' \rangle$ (Fig. 3.6).

If rules [L-Send] or [L-Recv] are used, then there is no silent transition available, thus $\langle \Sigma; L \rangle = \langle \Sigma'; L' \rangle$, and the required result follows.

If rule [L-Pre] is used, then the silent transition is unique, and we can apply inductive hypothesis on the premise. □

We introduce the converging relation in Def. 3.47.

Definition 3.47 (Converging Local Types Under Typing Contexts). We define a *converging* relation $\lesssim_\sigma$, parameterised by a trace σ, over local types under typing contexts.

The converging relation $\langle \Sigma_1; L_1 \rangle \lesssim_\sigma \langle \Sigma_2; L_2 \rangle$ holds, if there exist local types under typing contexts $\langle \Sigma'; L' \rangle$ such that $\langle \Sigma_1; L_1 \rangle \xrightarrow{!\sigma}\rightsquigarrow^* \langle \Sigma'; L' \rangle$, and $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma}\overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$.

We write $\langle \Sigma_1; L_1 \rangle \lesssim \langle \Sigma_2; L_2 \rangle$, if there exists a trace σ such that $\langle \Sigma_1; L_1 \rangle \lesssim_\sigma \langle \Sigma_2; L_2 \rangle$ holds. ⌟

For two local types under typing contexts to be converging, we allow each local type under its typing context to follow a common trace of unique reductions, then the local type under its typing context on the left hand side is permitted to make any silent transitions, and the one on the right hand side is permitted to make any *unique* silent transitions, for both to converge on the same local type under the same typing context. We can see the two local types under typing contexts in Ex. 3.43 satisfy this converging relation.

Lemma 3.48 (Converging Relation is a Simulation). *Given* $\langle \Sigma_1; L_1 \rangle \lesssim \langle \Sigma_2; L_2 \rangle$*. If* $\langle \Sigma_2; L_2 \rangle \xrightarrow{\alpha} \langle \Sigma'_2; L'_2 \rangle$*, then there exists* $\langle \Sigma'_1; L'_1 \rangle$ *such that* $\langle \Sigma_1; L_1 \rangle \xrightarrow{\alpha} \langle \Sigma'_1; L'_1 \rangle$*, and* $\langle \Sigma'_1; L'_1 \rangle \lesssim \langle \Sigma'_2; L'_2 \rangle$. ⌟

Proof. Suppose $\langle \Sigma_1; L_1 \rangle \lesssim_\sigma \langle \Sigma_2; L_2 \rangle$. We consider different cases of possible trace σ.

- Case $\sigma = \varepsilon$: in this case, we have $\langle \Sigma_1; L_1 \rangle \rightsquigarrow^* \langle \Sigma'; L' \rangle$, and $\langle \Sigma_2; L_2 \rangle \overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$. Any reduction of $\langle \Sigma_2; L_2 \rangle$ must be a reduction of $\langle \Sigma'; L' \rangle$, and thus can be matched by $\langle \Sigma_1; L_1 \rangle$ via rule [L-Pre], and the result will be identical. It is easy to see $\lesssim$ is reflexive, by taking an empty trace ε, followed by no silent transitions.

- Case $\sigma = \alpha \cdot \sigma'$: by definition of unique reduction (Def. 3.45), the action α must appear at the beginning of the trace σ. The required result then follows from taking the unique reduction of $\langle \Sigma_1; L_1 \rangle$ (since it is unique), and converging relation still holds by taking the remaining trace of common unique reductions σ'. □

Lemma 3.49 (Converging Relation is Transitive)**.** *If* $\langle \Sigma_1; L_1 \rangle \lesssim \langle \Sigma_2; L_2 \rangle$ *and* $\langle \Sigma_2; L_2 \rangle \lesssim \langle \Sigma_3; L_3 \rangle$, *then* $\langle \Sigma_1; L_1 \rangle \lesssim \langle \Sigma_3; L_3 \rangle$. ⌟

Proof. Suppose $\langle \Sigma_1; L_1 \rangle \lesssim_\sigma \langle \Sigma_2; L_2 \rangle$ and $\langle \Sigma_2; L_2 \rangle \lesssim_{\sigma'} \langle \Sigma_3; L_3 \rangle$. Furthermore, by expanding the definition, suppose $\langle \Sigma_1; L_1 \rangle \xrightarrow{!\sigma}\rightsquigarrow^* \langle \Sigma'; L' \rangle$, $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma}\overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$ from the first relation, and $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma'}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$, $\langle \Sigma_3; L_3 \rangle \xrightarrow{!\sigma'}\overset{!}{\rightsquigarrow}^* \langle \Sigma''; L'' \rangle$ from the second relation. We show $\langle \Sigma_1; L_1 \rangle \xrightarrow{!\sigma'}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$ for the transitive relation to hold.

From $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma}\overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$ and $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma'}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$, we notice that the traces σ and σ' are both unique reductions arising from $\langle \Sigma_2; L_2 \rangle$. Without loss of generality, we assume σ is a prefix of σ', and let $\sigma' = \sigma \cdot \sigma''$.

We have $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma}\overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$ and $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma \cdot \sigma''}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$. We can decompose both of the reduction sequences into $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma} \langle \Sigma'''; L''' \rangle \overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$, and $\langle \Sigma_2; L_2 \rangle \xrightarrow{!\sigma} \langle \Sigma'''; L''' \rangle \xrightarrow{!\sigma''}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$, since local type reductions are deterministic (Lem. 3.38).

To complete the proof, we need to find a way to compose the reduction $\langle \Sigma_1; L_1 \rangle \xrightarrow{!\sigma}\rightsquigarrow^* \langle \Sigma'; L' \rangle$ with the reduction $\langle \Sigma'''; L''' \rangle \xrightarrow{!\sigma''}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$.

If $\sigma'' = \varepsilon$, then we have $\langle \Sigma'''; L''' \rangle \rightsquigarrow^* \langle \Sigma''; L'' \rangle$. Note that previously, we have from the decomposition that $\langle \Sigma'''; L''' \rangle \overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$. It must be the case that $\langle \Sigma_1; L_1 \rangle \xrightarrow{!\sigma}\overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle \rightsquigarrow^* \langle \Sigma''; L'' \rangle$, which satisfies the requirement since $\overset{!}{\rightsquigarrow}^*$ is a subset of $\rightsquigarrow^*$.

Otherwise, if $\sigma'' = \alpha \cdot \sigma'''$. We decompose $\langle \Sigma'''; L''' \rangle \xrightarrow{!\alpha \cdot \sigma''}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$ into $\langle \Sigma'''; L''' \rangle \xrightarrow{!\alpha} \langle \Sigma'''_\alpha; L'''_\alpha \rangle \xrightarrow{!\sigma'''}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$. Since we have $\langle \Sigma'''; L''' \rangle \overset{!}{\rightsquigarrow}^* \langle \Sigma'; L' \rangle$, and we can apply Lem. 3.46 to obtain $\langle \Sigma'; L' \rangle \xrightarrow{!\alpha} \langle \Sigma'''_\alpha; L'''_\alpha \rangle$. We conclude with $\langle \Sigma_1; L_1 \rangle \xrightarrow{!\sigma}\rightsquigarrow^* \langle \Sigma'; L' \rangle \xrightarrow{!\alpha} \langle \Sigma'''_\alpha; L'''_\alpha \rangle \xrightarrow{!\sigma'''}\rightsquigarrow^* \langle \Sigma''; L'' \rangle$. □

We continue to define the *association* of global types and configurations in Def. 3.50, using the converging relation that we previously defined.

Definition 3.50 (Association of Global Types and Configurations)**.** Let $\langle \Gamma; G \rangle$ be a global type under a typing context. A configuration $\mathcal{S}$ is *associated* to $\langle \Gamma; G \rangle$, if:

1. all participating roles in G are present in the configuration $\mathcal{S}$: $\mathrm{pts}(G) \subseteq \mathrm{dom}(\mathcal{S})$;

2. (a) for any participating role $\mathbf{r} \in G$: the entry must converge with its projection from the global type $\mathcal{S}(\mathbf{r}) \lesssim \langle \Gamma; G \rangle \upharpoonright \mathbf{r}$; and

(b) for any non-participating role $\mathbf{r} \in (\operatorname{dom}(\mathcal{S}) \setminus \operatorname{pts}(G))$, the local type in the entry must be $\mathtt{end}$: $\mathcal{S}(\mathbf{r}) = \langle \Sigma; \mathtt{end} \rangle$ for some local typing context Σ.

We write $\mathcal{S} \Leftrightarrow \langle \Gamma; G \rangle$ if the configuration $\mathcal{S}$ is associated to $\langle \Gamma; G \rangle$. ⌟

We show in Thm. 3.51 that a reduction step of a global type is matched by an associated configuration, and the global type and configuration after reduction remain associated.

Theorem 3.51 (Global Type Reduction Matched by Associated Configuration)**.** *Given associated configuration and global types: $\mathcal{S} \Leftrightarrow \langle \Gamma; G \rangle$. If $\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$, then there exists $\mathcal{S}'$ such that $\mathcal{S} \xrightarrow{\alpha} \mathcal{S}'$ and $\mathcal{S}' \Leftrightarrow \langle \Gamma'; G' \rangle$;* ⌟

Proof. By induction on the derivation of global type reductions $\langle \Gamma; G \rangle \xrightarrow{\alpha} \langle \Gamma'; G' \rangle$ (Fig. 3.5).

- Case [G-Pfx]: We have $G = \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I}$, and the reduction step:

$$\langle \Gamma; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I} \rangle \xrightarrow{\alpha = \mathbf{p} \to \mathbf{q} : \ell_j(x_j : T_j)} \left\langle \Gamma + x_j^{\{\mathbf{p}, \mathbf{q}\}} : T_j; G_j \right\rangle$$

  with $j \in I$. We have $\operatorname{subj}(\alpha) = \{\mathbf{p}, \mathbf{q}\}$, and we first consider these entries and update the configuration $\mathcal{S}$.

  For $\mathbf{p}$ and $\mathbf{q}$, the projected local types under typing contexts are:

$$\langle \Gamma; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I} \rangle \upharpoonright \mathbf{p} = \langle \Sigma_{\mathbf{p}}; \mathbf{q} \oplus \{\ell_i(x_i : T_i).L_{\mathbf{p},i}\}_{i \in I} \rangle,$$
$$\langle \Gamma; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I} \rangle \upharpoonright \mathbf{q} = \langle \Sigma_{\mathbf{q}}; \mathbf{p} \& \{\ell_i(x_i : T_i).L_{\mathbf{q},i}\}_{i \in I} \rangle,$$

  where $\Sigma_{\mathbf{p}} = \Gamma \upharpoonright \mathbf{p}$ and $\Sigma_{\mathbf{q}} = \Gamma \upharpoonright \mathbf{q}$; and for all $i \in I$: $G_i \upharpoonright_{\emptyset} \mathbf{p} = L_{\mathbf{p},i}$ and $G_i \upharpoonright_{\emptyset} \mathbf{q} = L_{\mathbf{q},i}$.

  We can apply [L-Send] on $\mathbf{p}$ (resp. [L-Recv] on $\mathbf{q}$):

$$\langle \Sigma_{\mathbf{p}}; \mathbf{q} \oplus \{\ell_i(x_i : T_i).L_{\mathbf{p},i}\}_{i \in I} \rangle \xrightarrow{\mathbf{p} \to \mathbf{q} : \ell_j(x_j : T_j)} \langle \Sigma_{\mathbf{p}} + x_j^{\omega} : T_j; L_{\mathbf{p},j} \rangle,$$
$$\langle \Sigma_{\mathbf{q}}; \mathbf{p} \& \{\ell_i(x_i : T_i).L_{\mathbf{q},i}\}_{i \in I} \rangle \xrightarrow{\mathbf{p} \to \mathbf{q} : \ell_j(x_j : T_j)} \langle \Sigma_{\mathbf{q}} + x_j^{\omega} : T_j; L_{\mathbf{q},j} \rangle.$$

  By Lem. 3.17, we have that $(\Gamma + x_j^{\{\mathbf{p}, \mathbf{q}\}} : T_j) \upharpoonright \mathbf{p} = \Sigma_{\mathbf{p}} + x_j^{\omega} : T_j$ and $(\Gamma + x_j^{\{\mathbf{p}, \mathbf{q}\}} : T_j) \upharpoonright \mathbf{q} = \Sigma_{\mathbf{q}} + x_j^{\omega} : T_j$.

  From $\mathcal{S} \Leftrightarrow \langle \Gamma; G \rangle$, we have $\mathcal{S}(\mathbf{p}) \lesssim \langle \Gamma; G \rangle \upharpoonright \mathbf{p}$ and $\mathcal{S}(\mathbf{q}) \lesssim \langle \Gamma; G \rangle \upharpoonright \mathbf{q}$. We update the configuration entries to be the local types under typing contexts after reduction $\mathcal{S}'(\mathbf{p}) = \langle \Sigma_{\mathbf{p}} + x_j^{\omega} : T_j; L_{\mathbf{p},j} \rangle$ and $\mathcal{S}'(\mathbf{q}) = \langle \Sigma_{\mathbf{p}} + x_j^{\omega} : T_j; L_{\mathbf{q},j} \rangle$.

  We have shown that $\mathcal{S}'(\mathbf{p})$ and $\mathcal{S}'(\mathbf{q})$ are projections of $\left\langle \Gamma + x_j^{\{\mathbf{p}, \mathbf{q}\}} : T_j; G_j \right\rangle$, so $\lesssim$ holds by reflexivity. Furthermore, we have $\mathcal{S}(\mathbf{p}) \xrightarrow{\alpha} \mathcal{S}'(\mathbf{p})$ and $\mathcal{S}(\mathbf{q}) \xrightarrow{\alpha} \mathcal{S}'(\mathbf{q})$ by Lem. 3.48.

  Now we turn to non-active roles $\mathbf{r} \in (\operatorname{dom}(\mathcal{S}) \setminus \operatorname{subj}(\alpha))$. There are two cases to consider, depending on whether $\mathbf{r}$ is participating. The case where $\mathbf{r}$ is non-participating is trivial, since its projection will be $\mathtt{end}$ and there are no possible actions.

We focus on the case where $\mathbf{r}$ is participating. By Def. 3.41, we have $\mathcal{S}'(\mathbf{r}) = \mathcal{S}(\mathbf{r})$. We are then left to show $\mathcal{S}'(\mathbf{r}) \lesssim \left\langle \Gamma + x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j; G_j \right\rangle \upharpoonright \mathbf{r}$, given that $\mathcal{S}(\mathbf{r}) \lesssim \langle \Gamma; G \rangle \upharpoonright \mathbf{r}$.

We use the transitivity of $\lesssim$ (Lem. 3.49), and show $\langle \Gamma; G \rangle \upharpoonright \mathbf{r} \lesssim \left\langle \Gamma + x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j; G_j \right\rangle \upharpoonright \mathbf{r}$.

For $\mathbf{r}$ the projected local type under a typing context is:

$$\langle \Gamma; \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i\}_{i \in I} \rangle \upharpoonright \mathbf{r} = \langle \Sigma_{\mathbf{r}}; \textstyle\sum \{\ell_i(x_i : T_i).L_{\mathbf{r},i}\}_{i \in I} \rangle ,$$

where $\Sigma_{\mathbf{r}} = \Gamma \upharpoonright \mathbf{r}$, and for all $i \in I$: $G_i \upharpoonright_{\emptyset} \mathbf{r} = L_{\mathbf{r},i}$.

We can apply [L-Pre-Sil] on $\mathbf{r}$: $\langle \Sigma_{\mathbf{r}}; \sum \{\ell_i(x_i : T_i).L_{\mathbf{r},i}\}_{i \in I} \rangle \rightsquigarrow \left\langle \Sigma_{\mathbf{r}} + x_j^0 : T_j; L_{\mathbf{r},j} \right\rangle$.

By Lem. 3.17, we have that $(\Gamma + x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j) \upharpoonright \mathbf{r} = \Sigma_{\mathbf{r}} + x_j^0 : T_j$.

- Case [G-Cnt]: We have $G = \mathbf{p} \to \mathbf{q} : \ell(x : T).G'$, and the reduction step:

$$\langle \Gamma; \mathbf{p} \to \mathbf{q} : \ell(x : T).G' \rangle \xrightarrow{\alpha} \langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x : T).G'' \rangle$$

with $\{\mathbf{p},\mathbf{q}\} \cap \mathrm{subj}(\alpha) = \emptyset$ and $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle$.

We first construct a configuration $\mathcal{S}'' \Leftrightarrow \left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle$, in order to invoke the inductive hypothesis on $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle$.

We first consider the prefix roles $\{\mathbf{p},\mathbf{q}\}$, the projected local types under typing contexts are:

$$\langle \Gamma; \mathbf{p} \to \mathbf{q} : \ell(x : T).G' \rangle \upharpoonright \mathbf{p} = \langle \Sigma_{\mathbf{p}}; \mathbf{q} \oplus \ell(x : T).L'_{\mathbf{p}} \rangle ,$$
$$\langle \Gamma; \mathbf{p} \to \mathbf{q} : \ell(x : T).G' \rangle \upharpoonright \mathbf{q} = \langle \Sigma_{\mathbf{q}}; \mathbf{p} \& \ell(x : T).L'_{\mathbf{q}} \rangle ,$$

where $\Sigma_{\mathbf{p}} = \Gamma \upharpoonright \mathbf{p}$ and $\Sigma_{\mathbf{q}} = \Gamma \upharpoonright \mathbf{q}$; and $G' \upharpoonright_{\emptyset} \mathbf{p} = L'_{\mathbf{p}}$ and $G' \upharpoonright_{\emptyset} \mathbf{q} = L'_{\mathbf{q}}$.

From $\mathcal{S} \Leftrightarrow \langle \Gamma; G \rangle$, we have $\mathcal{S}(\mathbf{p}) \lesssim \langle \Gamma; G \rangle \upharpoonright \mathbf{p}$ and $\mathcal{S}(\mathbf{q}) \lesssim \langle \Gamma; G \rangle \upharpoonright \mathbf{q}$.

Since the projected local types for $\mathbf{p}$ and $\mathbf{q}$ both contain a singleton choice, the common reduction sequence of convergence must either be empty, or begin with the action $\alpha' = \mathbf{p} \to \mathbf{q} : \ell(x : T)$. In either case, we have $\mathcal{S}(\mathbf{p}) \rightsquigarrow^* \langle \Gamma; G \rangle \upharpoonright \mathbf{p}$ and $\mathcal{S}(\mathbf{q}) \rightsquigarrow^* \langle \Gamma; G \rangle \upharpoonright \mathbf{q}$.

We set $\mathcal{S}''(\mathbf{p}) = \left\langle \Sigma_{\mathbf{p}} + x^0 : T; L'_{\mathbf{p}} \right\rangle$ and $\mathcal{S}''(\mathbf{q}) = \left\langle \Sigma_{\mathbf{q}} + x^0 : T; L'_{\mathbf{q}} \right\rangle$, and they are projections of $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle$.

For other non-prefix roles $\mathbf{r} \in (\mathrm{dom}(\mathcal{S}) \setminus \{\mathbf{p},\mathbf{q}\})$, we set $\mathcal{S}''(\mathbf{r}) = \mathcal{S}(\mathbf{r})$. We use a similar argument as the case of [G-Pfx]. The projection onto $\mathbf{r}$ is

$$\langle \Gamma; \mathbf{p} \to \mathbf{q} : \ell(x : T).G' \rangle \upharpoonright \mathbf{r} = \langle \Sigma_{\mathbf{r}}; \ell(x : T).L'_{\mathbf{r}} \rangle$$

where $\Sigma_{\mathbf{r}} = \Gamma \upharpoonright \mathbf{r}$, and $G' \upharpoonright_{\emptyset} \mathbf{r} = L'_{\mathbf{r}}$. We can apply [L-Pre-Sil] for

$$\langle \Sigma_{\mathbf{r}}; \ell(x : T).L'_{\mathbf{r}} \rangle \rightsquigarrow \left\langle \Sigma_{\mathbf{r}} + x^0 : T; L'_{\mathbf{r}} \right\rangle ,$$

which corresponds to the projection of $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle$.

Applying the inductive hypothesis, we obtain a configuration $\mathcal{S}'''$ with $\mathcal{S}'' \xrightarrow{\alpha} \mathcal{S}'''$ and $\mathcal{S}''' \Leftrightarrow \langle \Gamma'; G'' \rangle$.

We now construct a configuration $\mathcal{S}'$ with $\mathcal{S} \xrightarrow{\alpha} \mathcal{S}'$, by setting $\mathcal{S}'(\mathbf{r}) = \mathcal{S}'''(\mathbf{r})$ for $\mathbf{r} \in \mathrm{subj}(\alpha)$, since $\mathcal{S}(\mathbf{r}) = \mathcal{S}''(\mathbf{r})$ and $\mathcal{S}''(\mathbf{r}) \xrightarrow{\alpha} \mathcal{S}'''(\mathbf{r})$; and $\mathcal{S}'(\mathbf{r}) = \mathcal{S}(\mathbf{r})$ for $\mathbf{r} \in (\mathrm{dom}(\mathcal{S}) \setminus \mathrm{subj}(\alpha))$.

We are left to show $\mathcal{S}' \Leftrightarrow \langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x:T).G'' \rangle$.

We begin with the prefix roles $\{\mathbf{p},\mathbf{q}\}$, which we show the case for role $\mathbf{p}$, and the case for role $\mathbf{q}$ is similar. We have $\mathcal{S}'''(\mathbf{p}) \lesssim \langle \Gamma'; G'' \rangle \upharpoonright \mathbf{p}$, where $\mathcal{S}''(\mathbf{p}) = \mathcal{S}'''(\mathbf{p}) = \left\langle \Sigma_{\mathbf{p}} + x^{\hat{0}} : T; L'_{\mathbf{p}} \right\rangle$; and we need to show $\mathcal{S}'(\mathbf{p}) \lesssim \langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x:T).G'' \rangle \upharpoonright \mathbf{p}$.

We compute the projection $\langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x:T).G'' \rangle \upharpoonright \mathbf{p} = \langle \Sigma'_{\mathbf{p}}; \mathbf{q} \oplus \ell(x:T).L''_{\mathbf{p}} \rangle$, where $\Sigma'_{\mathbf{p}} = \Gamma' \upharpoonright \mathbf{p}$ and $L''_{\mathbf{p}} = G'' \upharpoonright_{\emptyset} \mathbf{p}$.

Expanding the definition of $\lesssim$ and applying equality on $\mathcal{S}'''(\mathbf{p}) \lesssim \langle \Gamma'; G'' \rangle \upharpoonright \mathbf{p}$, we have $\left\langle \Sigma_{\mathbf{p}} + x^{\hat{0}} : T; L'_{\mathbf{p}} \right\rangle \xrightarrow{!\sigma} \rightsquigarrow^* \langle \Sigma''_{\mathbf{p}}; L'''_{\mathbf{p}} \rangle$ and $\langle \Gamma'; G'' \rangle \upharpoonright \mathbf{p} = \langle \Sigma'_{\mathbf{p}}; L''_{\mathbf{p}} \rangle \xrightarrow{!\sigma} \overset{!}{\rightsquigarrow}{}^* \langle \Sigma''_{\mathbf{p}}; L'''_{\mathbf{p}} \rangle$.

We note that $\langle \Sigma'_{\mathbf{p}}; \mathbf{q} \oplus \ell(x:T).L''_{\mathbf{p}} \rangle$ has a singleton choice, and thus has a unique reduction $\langle \Sigma'_{\mathbf{p}}; \mathbf{q} \oplus \ell(x:T).L''_{\mathbf{p}} \rangle \xrightarrow{!\mathbf{p} \to \mathbf{q} : \ell(x:T)} \langle \Sigma'_{\mathbf{p}} + x^{\omega} : T; L''_{\mathbf{p}} \rangle$; and the projection of $\langle \Gamma; G \rangle$ onto $\mathbf{p}$ have the same unique reduction $\langle \Sigma_{\mathbf{p}}; \mathbf{q} \oplus \ell(x:T).L'_{\mathbf{p}} \rangle \xrightarrow{!\mathbf{p} \to \mathbf{q} : \ell(x:T)} \langle \Sigma_{\mathbf{p}} + x^{\omega} : T; L'_{\mathbf{p}} \rangle$.

As a prerequisite for the upcoming step, we show that $\Sigma'_{\mathbf{p}} + x^{\hat{0}} : T = \Sigma'_{\mathbf{p}}$: the local typing context $\Sigma'_{\mathbf{p}}$ is projected from Γ', which is obtained from $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle$. Since $\mathrm{subj}(\alpha) \cap \{\mathbf{p},\mathbf{q}\} = \emptyset$, we apply Lem. 3.31 to know that $\widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T \in \Gamma'$, and thus $x^{\hat{0}} : T \in \Sigma'_{\mathbf{p}}$ by Prop. 3.16. Therefore, the expansion does not vary the local typing context, and two typing contexts are the same.

Applying Lem. 3.39 and combining existing results, we have $\langle \Gamma; G \rangle \upharpoonright \mathbf{p} \xrightarrow{!\mathbf{p} \to \mathbf{q} : \ell(x:T) \cdot \sigma} \rightsquigarrow^* \langle \Sigma''_{\mathbf{p}}; L'''_{\mathbf{p}} \rangle$ and $\langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x:T).G'' \rangle \upharpoonright \mathbf{p} \xrightarrow{!\mathbf{p} \to \mathbf{q} : \ell(x:T) \cdot \sigma} \overset{!}{\rightsquigarrow}{}^* \langle \Sigma''_{\mathbf{p}}; L'''_{\mathbf{p}} \rangle$, to establish the convergence relation $\langle \Gamma; G \rangle \upharpoonright \mathbf{p} \lesssim \langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x:T).G'' \rangle \upharpoonright \mathbf{p}$. This can be combined with $\mathcal{S}(\mathbf{p}) \lesssim \langle \Gamma; G \rangle \upharpoonright \mathbf{p}$ from the premise via transitivity to obtain the required result.

For non-prefix participating roles, the projection onto $\mathbf{r}$ is

$$\langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x:T).G'' \rangle \upharpoonright \mathbf{r} = \langle \Sigma'_{\mathbf{r}}; \ell(x:T).L''_{\mathbf{r}} \rangle$$

where $\Sigma'_{\mathbf{r}} = \Gamma' \upharpoonright \mathbf{r}$, and $G'' \upharpoonright_{\emptyset} \mathbf{r} = L''_{\mathbf{r}}$. Since the silent prefix is a singleton, we can apply [L-Pre-Sil] for a unique reduction

$$\langle \Sigma'_{\mathbf{r}}; \ell(x:T).L''_{\mathbf{r}} \rangle \overset{!}{\rightsquigarrow} \left\langle \Sigma'_{\mathbf{r}} + x^{0} : T; L''_{\mathbf{r}} \right\rangle.$$

Using a similar reasoning, we show that $\Sigma'_{\mathbf{r}} + x^{\hat{0}} : T = \Sigma'_{\mathbf{r}}$: the local typing context $\Sigma'_{\mathbf{r}}$ is projected from Γ', which is obtained from $\left\langle \Gamma + \widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T; G' \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle$. Since $\mathrm{subj}(\alpha) \cap \{\mathbf{p},\mathbf{q}\} =$

$\emptyset$, we apply Lem. 3.31 to know that $\widehat{x}^{\{\mathbf{p},\mathbf{q}\}} : T \in \Gamma'$, and thus $x^0 : T \in \Sigma'_{\mathbf{r}}$ by Prop. 3.16. Therefore, the expansion does not vary the local typing context, and two local typing contexts are the same.

Using the previous result, we have $\langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x : T).G'' \rangle \upharpoonright \mathbf{r} \overset{!}{\rightsquigarrow} \langle \Gamma'; G'' \rangle \upharpoonright \mathbf{r}$, and we can conclude $\langle \Gamma'; G'' \rangle \upharpoonright \mathbf{r} \lesssim \langle \Gamma'; \mathbf{p} \to \mathbf{q} : \ell(x : T).G'' \rangle \upharpoonright \mathbf{r}$, by taking the empty reduction sequence ε, and allowing the right hand side to take one unique silent transition to reach the left hand side. This can be combined with the result obtained in the inductive hypothesis, $\mathbb{S}'''(\mathbf{r}) \lesssim \langle \Gamma'; G'' \rangle \upharpoonright \mathbf{r}$ via transitivity, and noting that $\mathbb{S}'''(\mathbf{r}) = \mathbb{S}'(\mathbf{r})$ for $\mathbf{r} \in \text{subj}(\alpha)$; and $\mathbb{S}'''(\mathbf{r}) = \mathbb{S}''(\mathbf{r}) = \mathbb{S}(\mathbf{r}) = \mathbb{S}'(\mathbf{r})$ otherwise, to obtain the required result.

- Case [G-Rec]: We have $G = \mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G$, and the reduction step:

$$\left\langle \Gamma; \mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G' \right\rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle$$

with $\langle \Gamma + x^{\mathbb{P}} : T; G'[\mu\mathbf{t}\,(x : T).G'/\mathbf{t}] \rangle \xrightarrow{\alpha} \langle \Gamma'; G'' \rangle$. To show the required result, we need to invoke the inductive hypothesis on $\mathbb{S} \Leftrightarrow \langle \Gamma + x^{\mathbb{P}} : T; G'[\mu\mathbf{t}\,(x : T).G'/\mathbf{t}] \rangle$.

For each participating role $\mathbf{r} \in G'$, the projection of G will be either $\mu\mathbf{t}\,(x^{\omega} := E : T).L'$ or $\mu\mathbf{t}\,(x^0 : T).L'$, where $G' \upharpoonright_{\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{r} = L'$, which means we can apply silent transition rules [L-Pre-Rec-ω] and [L-Pre-Rec-0] respectively. $\square$

We conjecture the other direction to also hold. We first prove some properties for reductions of a single local type, and then demonstrate the challenges.

Recall that previously we mentioned that local types are *strongly* sequenced (§ 2.4.2, Page 21), and global types are weakly sequenced (§ 2.4.1, Page 18). We show, in the following lemma, that a local type reduction can be matched by a unique trace of reductions[9] on the global type, without using weakly-sequenced reductions.

Lemma 3.52. *If* $\langle \Gamma; G \rangle \upharpoonright \mathbf{r} = \langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$, *then there exists* $\langle \Gamma'; G' \rangle$ *and a unique trace* σ *such that* $\langle \Gamma; G \rangle \xrightarrow{\sigma \cdot \alpha} \langle \Gamma'; G' \rangle$ *and* $\langle \Gamma'; G' \rangle \upharpoonright \mathbf{r} = \langle \Sigma'; L' \rangle$. *Moreover, the role current role* $\mathbf{r}$ *does not appear in the trace* σ *preceding the action* α: $\mathbf{r} \notin \text{subj}(\sigma)$, *and the reduction does not involve rule* [G-Cnt]. ⌟

Proof. By induction on the derivation of local type reductions $\langle \Sigma; L \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$ (Fig. 3.6).

- Case [L-Send]: We have the reduction

$$\langle \Sigma; \mathbf{q} \oplus \{\ell_i(x_i : T_i).L_i\}_{i \in I} \rangle \xrightarrow{\mathbf{r} \to \mathbf{q} : \ell_j(x_j : T_j)} \langle \Sigma + x_j^{\omega} : T_j; L_j \rangle$$

with $j \in I$ and $\Gamma \upharpoonright \mathbf{r} = \Sigma$. By inversion of projection (Lem. 3.21), the global type $G = \mathbf{r} \to \mathbf{q}\,\{\ell_i(x_i : T_i).G_i\}$ where $G_i \upharpoonright_{\emptyset} \mathbf{r} = L_i$ for $i \in I$.

[9] Note that a unique trace of reductions is different from a trace of unique reductions. The latter further requires that no branching occurs.

We take $\sigma = \varepsilon$ (thus $\text{subj}(\sigma) = \emptyset$), and the global type can reduce using rule [G-Pfx]:

$$\langle \Gamma; \mathbf{r} \to \mathbf{q} \{\ell_i(x_i : T_i).G_i\} \rangle \xrightarrow{\mathbf{r}\to\mathbf{q}:\ell_j(x_j : T_j)} \left\langle \Gamma + x_j^{\{\mathbf{r},\mathbf{q}\}} : T_j; G_j \right\rangle$$

and $\Gamma + x_j^{\{\mathbf{r},\mathbf{q}\}} : T_j \restriction \mathbf{r} = \Sigma + x_j^{\omega} : T_j$ by Lem. 3.17.

- Case [L-Recv]: This case is similar to [L-Send]. We have the reduction

  $$\langle \Sigma; \mathbf{q}\&\{\ell_i(x_i : T_i).L_i\}_{i\in I} \rangle \xrightarrow{\mathbf{q}\to\mathbf{r}:\ell_j(x_j : T_j)} \langle \Sigma + x_j^{\omega} : T_j; L_j \rangle$$

  with $j \in I$ and $\Gamma \restriction \mathbf{r} = \Sigma$. By inversion of projection (Lem. 3.21), the global type $G = \mathbf{q} \to \mathbf{r} \{\ell_i(x_i : T_i).G_i\}$ where $G_i \restriction_{\emptyset} \mathbf{r} = L_i$ for $i \in I$.

  We take $\sigma = \varepsilon$ (thus $\text{subj}(\sigma) = \emptyset$), and the global type can reduce using rule [G-Pfx]:

  $$\langle \Gamma; \mathbf{q} \to \mathbf{r} \{\ell_i(x_i : T_i).G_i\} \rangle \xrightarrow{\mathbf{q}\to\mathbf{r}:\ell_j(x_j : T_j)} \left\langle \Gamma + x_j^{\{\mathbf{q},\mathbf{r}\}} : T_j; G_j \right\rangle$$

  and $\Gamma + x_j^{\{\mathbf{q},\mathbf{r}\}} : T_j \restriction \mathbf{r} = \Sigma + x_j^{\omega} : T_j$ by Lem. 3.17.

- Case [L-Pre]: We have $\langle \Sigma; L \rangle \rightsquigarrow \langle \Sigma''; L'' \rangle$ and $\langle \Sigma''; L'' \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$. We perform a case analysis on the silent reduction, and invoke the inductive hypothesis on the concrete reduction.

  – Sub-Case [L-Pre-Sil]: We have the silent reduction

  $$\langle \Sigma; \textstyle\sum \{\ell_i(x_i : T_i).L_i\}_{i\in I} \rangle \rightsquigarrow \left\langle \Sigma + x_j^{0} : T_j; L_j \right\rangle$$

  with $j \in I$ and $\Gamma \restriction \mathbf{r} = \Sigma$.

  By inversion of projection (Lem. 3.21), the global type $G = \mathbf{p} \to \mathbf{q} \{\ell_i(x_i : T_i).G_i\}$ where $G_i \restriction_{\emptyset} \mathbf{r} = L_i$ for $i \in I$, and $\mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\}$.

  We take $\sigma = \mathbf{p} \to \mathbf{q} : \ell_j(x_j : T_j)$, and the global type can reduce using rule [G-Pfx]:

  $$\langle \Gamma; \mathbf{p} \to \mathbf{q} \{\ell_i(x_i : T_i).G_i\} \rangle \xrightarrow{\mathbf{p}\to\mathbf{q}:\ell_j(x_j : T_j)} \left\langle \Gamma + x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j; G_j \right\rangle$$

  and $\Gamma + x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j \restriction \mathbf{r} = \Sigma + x_j^{0} : T_j$ by Lem. 3.17. We note that this trace is unique due to the fact projected silent prefixes reduce disjointly (Lem. 3.37).

  Invoking the inductive hypothesis on $\langle \Sigma + x_j^{0} : T_j; L_j \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$ gives $\langle \Gamma'; G' \rangle$ and a trace σ' with $\left\langle \Gamma + x_j^{\{\mathbf{p},\mathbf{q}\}} : T_j; G_j \right\rangle \xrightarrow{\sigma' \cdot \alpha} \langle \Gamma'; G' \rangle$ and $\langle \Gamma'; G' \rangle \restriction \mathbf{r} = \langle \Sigma'; L' \rangle$.

  We can combine the global type reduction to obtain $\langle \Gamma; G \rangle \xrightarrow{\mathbf{p}\to\mathbf{q}:\ell_j(x_j : T_j)\cdot\sigma'\cdot\alpha} \langle \Gamma'; G' \rangle$. Note that $\mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\}$, so the side condition on the trace is preserved.

– Sub-Case [L-Pre-Rec-ω]: We have the silent reduction

$$\langle \Sigma; \mu \mathbf{t}\,(x^{\omega} := E : T).L \rangle \rightsquigarrow \langle \Sigma + x^{\omega} : T; L[\mu \mathbf{t}\,(x : T).L/\mathbf{t}] \rangle$$

By inversion of projection (Lem. 3.21), the global type $G = \mu \mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$, where $G' \restriction_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{p} = L'$ and $\mathbf{p} \in \mathbb{P}$.

We invoke the inductive hypothesis on $\langle \Sigma + x^{\omega} : T; L[\mu \mathbf{t}\,(x : T).L/\mathbf{t}] \rangle \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$, using a global type under a typing context $\langle \Gamma + x^{\mathbb{P}} : T; G[\mu \mathbf{t}\,(x : T).G/\mathbf{t}] \rangle$. This is a consequence of $G[\mu \mathbf{t}\,(x : T).G/\mathbf{t}] \restriction_{\varnothing} \mathbf{p} = L[\mu \mathbf{t}\,(x : T).L/\mathbf{t}]$ (Lem. 3.22), and $\Gamma + x^{\mathbb{P}} : T \restriction \mathbf{p} = \Sigma + x^{\omega} : T$ (Lem. 3.17, $\mathbf{p} \in \mathbb{P}$).

The inductive hypothesis gives $\langle \Gamma'; G' \rangle$ and a trace σ' with $\langle \Gamma + x^{\mathbb{P}} : T; G[\mu \mathbf{t}\,(x : T).G/\mathbf{t}] \rangle \xrightarrow{\sigma' \cdot \alpha} \langle \Gamma'; G' \rangle$ and $\langle \Gamma'; G' \rangle \restriction \mathbf{r} = \langle \Sigma'; L' \rangle$. We can then take the first reduction of the trace $\sigma' \cdot \alpha$ as the premise of rule [G-Rec], and obtain $\langle \Gamma; \mu \mathbf{t}\,(x^{\mathbb{P}} := E : T).G' \rangle \xrightarrow{\sigma' \cdot \alpha} \langle \Gamma'; G' \rangle$.

– Sub-Case [L-Pre-Rec-ω]: This case is similar to [L-Pre-Rec-0]. We have the silent reduction

$$\left\langle \Sigma; \mu \mathbf{t}\,(x^{0} : T).L \right\rangle \rightsquigarrow \left\langle \Sigma + x^{0} : T; L[\mu \mathbf{t}\,(x : T).L/\mathbf{t}] \right\rangle$$

By inversion of projection (Lem. 3.21), the global type $G = \mu \mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$, where $G' \restriction_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{p} = L'$ and $\mathbf{p} \notin \mathbb{P}$.

Then we can follow the same step as the the previous case to obtain the required result. □

Then, we prove an inversion lemma for local type reductions.

Lemma 3.53 (Inversion of Local Type Reductions). *Fix an action* $\alpha = \mathbf{p} \to \mathbf{q} : \ell(x : T)$.

1. *If* $\langle \Gamma; G \rangle \restriction \mathbf{p} \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$*, then one of the following holds:*

   (a) $G = \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i'\}_{i \in I}$*, and there exists* $j \in I$ *with* $\ell_j = \ell$*,* $x_j = x$*, and* $T_j = T$*;*

   (b) $G = \mathbf{s} \to \mathbf{t} : \ell'(x' : T').G'$ *with* $\mathbf{p} \notin \{\mathbf{s}, \mathbf{t}\}$*;*

   (c) $G = \mu \mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$*;*

2. *If* $\langle \Gamma; G \rangle \restriction \mathbf{q} \xrightarrow{\alpha} \langle \Sigma'; L' \rangle$*, then one of the following holds:*

   (a) $G = \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : T_i).G_i'\}_{i \in I}$*, and there exists* $j \in I$ *with* $\ell_j = \ell$*,* $x_j = x$*, and* $T_j = T$*;*

   (b) $G = \mathbf{s} \to \mathbf{t}\left\{\ell_j(x_j : T_j).G_j\right\}_{j \in J}$ *with* $\mathbf{q} \notin \{\mathbf{s}, \mathbf{t}\}$*;*

   (c) $G = \mu \mathbf{t}\,(x^{\mathbb{P}} := E : T).G'$. ⌟

Proof. By inversion of the local type reduction (Fig. 3.6), along with inversion of projection (Lem. 3.21).

1. Case *(a)* corresponds to rule [L-Send], case *(b)* corresponds to rule [L-Pre] with the silent transition rule [L-Pre-Sil], case *(c)* corresponds to rule [L-Pre] with the silent transition rule [L-Pre-Rec-ω] or rule [L-Pre-Rec-0].

   We draw some attention to case *(b)*, in particular, the restriction that the global type must be a singleton branch. This is due to the compatibility relation ($\bowtie$) (Def. 3.18). There is no rule for two sending local types to be compatible, so the only way for a sending local type to reduce in such a situation is when the branch in the global is a singleton (cf. Ex. 3.19, Item 2).

2. Similar to the previous case, except the restriction on case *(b)* no longer applies. □

The challenge with proving that configuration reductions are matched by global type reductions lies in the convergence relation $\lesssim$. In particular, the usual proof technique in MPST relies on configuration entries being a subtype of the projection, whereas in our case, the configuration entries and the projection are related by silent transitions and a convergence sequence.

Moreover, our configuration reduction semantics is defined in a way where non-active entries are unchanged, which leads to a gap between the global type governing the configuration, and the global type that is used to produce the projections, as we demonstrate with the following example Ex. 3.54.

Example 3.54 (Configuration Reductions)**.** We use the global type in Ex. 3.4, Item 1.

$$G = \mathbf{A} \to \mathbf{B} : \mathsf{Fst}(x : \mathtt{int}).\mathbf{B} \to \mathbf{C} : \mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathbf{C} \to \mathbf{D} : \mathsf{Trd}(z : \mathtt{int}\{x = z\}).\mathtt{end}.$$

We construct a configuration using by projecting the global type onto all roles.

$$\mathcal{S} = \left\{ \begin{array}{l} \mathbf{A} : \langle \varnothing; \mathbf{B}{\oplus}\mathsf{Fst}(x : \mathtt{int}).\mathtt{end}\rangle \\ \mathbf{B} : \langle \varnothing; \mathbf{A}\&\mathsf{Fst}(x : \mathtt{int}).\mathbf{C}{\oplus}\mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathtt{end}\rangle \\ \mathbf{C} : \langle \varnothing; \mathsf{Fst}(x : \mathtt{int}).\mathbf{B}\&\mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathbf{D}{\oplus}\mathsf{Trd}(z : \mathtt{int}\{x = z\}).\mathtt{end}\rangle \\ \mathbf{D} : \langle \varnothing; \mathsf{Fst}(x : \mathtt{int}).\mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathbf{C}\&\mathsf{Trd}(z : \mathtt{int}\{x = z\}).\mathtt{end}\rangle \end{array} \right\}$$

The configuration $\mathcal{S}$ can make the following reduction:

$$\begin{array}{l} \mathcal{S} \\ \xrightarrow{\mathbf{A}\to\mathbf{B}:\mathsf{Fst}} \left\{ \begin{array}{l} \mathbf{A} : \langle x^{\omega} : \mathtt{int}; \mathtt{end}\rangle \\ \mathbf{B} : \langle x^{\omega} : \mathtt{int}; \mathbf{C}{\oplus}\mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathtt{end}\rangle \\ \mathbf{C} : \langle \varnothing; \mathsf{Fst}(x : \mathtt{int}).\mathbf{B}\&\mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathbf{D}{\oplus}\mathsf{Trd}(z : \mathtt{int}\{x = z\}).\mathtt{end}\rangle \\ \mathbf{D} : \langle \varnothing; \mathsf{Fst}(x : \mathtt{int}).\mathsf{Snd}(y : \mathtt{int}\{x = y\}).\mathbf{C}\&\mathsf{Trd}(z : \mathtt{int}\{x = z\}).\mathtt{end}\rangle \end{array} \right\} \end{array}$$

We can see that entries of **C** and **D** remain to be their projection of G, despite the progression of the overall global type after the message Fst from **A** to **B**. ⌟

To prove the conjecture, we need a tighter relation between configuration entries and the projection of the governing global type, or between the global type that projects to the configuration and the governing local type.

In the next section, we demonstrate how to erase the refined global and local types to basic global and local types without refinements. While we have not been able to show operational correspondence in both directions here, we can recover a weaker form of the operational correspondence property from the original MPST theory.

3.7 Erasing Refinements

In this section, we show how refined global and local types can be erased into basic (i.e. without refinement types) global and local types. The erasure process converts the syntax of refined MPST (Def. 3.1) to the original MPST (Def. 2.1).

We use $\ulcorner G_{refined} \urcorner = G_{basic}$ and $\ulcorner L_{refined} \urcorner = L_{basic}$ to denote the erasure of refined type $G_{refined}$ (resp. $L_{refined}$) into the basic type G_{basic} (resp. L_{basic}), and give the full definition in Def. 3.55. Note that the refined types inside the erasure operator follow the syntax in Def. 3.1, and the basic types on the right hand side follow the syntax in Def. 2.1.

Definition 3.55 (Erasure of Global and Local Types)**.** We define the *erasure* of a refined global type G, denoted $\ulcorner G \urcorner$, as follows:

$$\begin{array}{rcl}
\ulcorner \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).G_i'\}_{i\in I} \urcorner & = & \mathbf{p} \to \mathbf{q}\{\ell_i(\mathsf{S}_i).\ulcorner G_i' \urcorner\}_{i\in I} \\
\ulcorner \mu \mathbf{t}\,(x^{\mathbb{P}} := E : T).G' \urcorner & = & \mu \mathbf{t}.\ulcorner G' \urcorner \\
\ulcorner \mathbf{t}\,\langle x := E \rangle \urcorner & = & \mathbf{t} \\
\ulcorner \mathtt{end} \urcorner & = & \mathtt{end}
\end{array}$$

We define the *erasure* of a refined local type L, denoted $\ulcorner L \urcorner$, as follows:

$$\begin{array}{rcl}
\ulcorner \mathbf{p}\&\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).L_i'\}_{i\in I} \urcorner & = & \mathbf{p}\&\{\ell_i(\mathsf{S}_i).\ulcorner L_i' \urcorner\}_{i\in I} \\
\ulcorner \mathbf{p}\oplus\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).L_i'\}_{i\in I} \urcorner & = & \mathbf{p}\oplus\{\ell_i(\mathsf{S}_i).\ulcorner L_i' \urcorner\}_{i\in I} \\
\ulcorner \sum\{\ell_i(x_i : T_i).L_i'\}_{i\in I} \urcorner & = & \sqcup_{i\in I} \ulcorner L_i \urcorner \\
\ulcorner \mu \mathbf{t}\,(x^{\omega} := E : T).L' \urcorner & = & \mu \mathbf{t}.\ulcorner L' \urcorner \\
\ulcorner \mu \mathbf{t}\,(x^{0} : T).L' \urcorner & = & \mu \mathbf{t}.\ulcorner L' \urcorner \\
\ulcorner \mathbf{t}\,\langle x := E \rangle \urcorner & = & \mathbf{t} \\
\ulcorner \mathbf{t} \urcorner & = & \mathbf{t} \\
\ulcorner \mathtt{end} \urcorner & = & \mathtt{end}
\end{array}$$

$\sqcup$ is the full merge operator defined in Def. 2.11. ⌟

In addition, we show that erasure commutes with projection in Thm. 3.56.

Theorem 3.56 (Erasure Commutes with Projection). *If a refined global type G projects onto $\mathbf{r}$ as a local type L: $G \upharpoonright_\Phi \mathbf{r} = L$, then the erased global type $\ulcorner G \urcorner$ projects onto $\mathbf{r}$ as the erased local type $\ulcorner L \urcorner$: $\ulcorner G \urcorner \upharpoonright \mathbf{r} = \ulcorner L \urcorner$.* ⌟

Proof. By induction on global type projection $G \upharpoonright_\Phi \mathbf{r} = L$ (Fig. 3.3).

- Case $(\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).G_i'\}_{i \in I}) \upharpoonright_\Phi \mathbf{r}$

  Erasing LHS gives $\ulcorner(\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).G_i'\}_{i\in I})\urcorner = \mathbf{p} \to \mathbf{q}\{\ell_i(x_i : \mathsf{S}_i).\ulcorner G_i' \urcorner\}_{i \in I}$.

  - Sub-Case $\mathbf{r} = \mathbf{p}$: Refined projection gives $\mathbf{q}\oplus\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).L_i\}_{i\in I}$, where $G_i' \upharpoonright_\Phi \mathbf{r} = L_i$.
    Erasing RHS gives $\ulcorner \mathbf{q}\oplus\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).L_i\}_{i\in I}\urcorner = \mathbf{q}\oplus\{\ell_i(\mathsf{S}_i).\ulcorner L_i \urcorner\}_{i\in I}$.
    Basic projection $\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : \mathsf{S}_i).\ulcorner G_i' \urcorner\}_{i\in I} \upharpoonright \mathbf{r} = \mathbf{q}\oplus\{\ell_i(\mathsf{S}_i).\ulcorner L_i \urcorner\}_{i\in I}$ holds by applying inductive hypothesis on $G_i' \upharpoonright_\Phi \mathbf{r} = L_i$.
  - Sub-Case $\mathbf{r} = \mathbf{q}$: similar to the previous case.
  - Sub-Case $\mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\}, \forall i \in I : \mathbf{r} \in G_i'$ and $\bowtie_{i\in I} L_i$:
    Refined projection gives $\sum\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).L_i\}_{i\in I}$, where $G_i' \upharpoonright_\Phi \mathbf{r} = L_i$.
    Erasing RHS gives $\ulcorner \sum\{\ell_i(x_i : \mathsf{S}_i\{E_i\}).L_i\}_{i\in I}\urcorner = \sqcup_{i\in I}\ulcorner L_i \urcorner$.
    Basic projection $\mathbf{p} \to \mathbf{q}\{\ell_i(x_i : \mathsf{S}_i).\ulcorner G_i' \urcorner\}_{i\in I} \upharpoonright \mathbf{r} = \sqcup_{i\in I}\ulcorner L_i \urcorner$ holds by applying inductive hypothesis on $G_i' \upharpoonright_\Phi \mathbf{r} = L_i'$.
  - Sub-Case $\mathbf{r} \notin \{\mathbf{p}, \mathbf{q}\}$ and $\forall i \in I : \mathbf{r} \notin G_i'$:
    Refined projection gives $\mathtt{end}$, which erases to $\mathtt{end}$.
    Basic projection gives $\sqcup_{i\in I} L_i$. Since $\forall i \in I : \mathbf{r} \notin G_i'$, the erased projection gives $\ulcorner G_i' \upharpoonright_\Phi \mathbf{r} \urcorner = \ulcorner \mathtt{end} \urcorner = \mathtt{end}$, and merging $\mathtt{end}$s gives $\mathtt{end}$.

- Case $(\mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G') \upharpoonright_\Phi \mathbf{r}$:

  Erasing LHS gives $\ulcorner \mu\mathbf{t}\,(x^{\mathbb{P}} := E : T).G' \urcorner = \mu\mathbf{t}.\ulcorner G' \urcorner$.

  - Sub-Case $\mathbf{r} \in \mathbb{P}$ and $\mathbf{r} \in G'$:
    Refined projection gives $\mu\mathbf{t}\,(x^{\omega} := E : T).L'$, where $L' = G' \upharpoonright_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{r}$.
    Erasing RHS gives $\ulcorner \mu\mathbf{t}\,(x^{\omega} := E : T).L' \urcorner = \mu\mathbf{t}.\ulcorner L' \urcorner$.
    Applying inductive hypothesis on $G' \upharpoonright_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{r} = L'$, and utilising $\mathbf{r} \in G'$, we know the erased type L' is not of form $\widetilde{\mu\mathbf{t}'}.\mathbf{t}$.
    Therefore, the basic projection gives $\mu\mathbf{t}.\ulcorner G' \urcorner \upharpoonright \mathbf{r} = \mu\mathbf{t}.\ulcorner L' \urcorner$, as required.
  - Sub-Case $\mathbf{r} \in \mathbb{P}$ and $\mathbf{r} \notin G'$:
    Refined projection gives $\mu\mathbf{t}\,(x^{0} : T).L$, where $L' = G' \upharpoonright_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{r}$.
    Erasing RHS gives $\ulcorner \mu\mathbf{t}\,(x^{0} : T).L \urcorner = \mu\mathbf{t}.\ulcorner L' \urcorner$.
    Applying inductive hypothesis on $G' \upharpoonright_{\Phi,\mathbf{t}:\langle x^{\mathbb{P}}, T\rangle} \mathbf{r} = L'$, and utilising $\mathbf{r} \in G'$, we know the erased type L' is not of form $\widetilde{\mu\mathbf{t}'}.\mathbf{t}$.
    Therefore, the basic projection gives $\mu\mathbf{t}.\ulcorner G' \urcorner \upharpoonright \mathbf{r} = \mu\mathbf{t}.\ulcorner L' \urcorner$, as required.

  - Sub-Case $L' = G' \restriction_{\Phi, \mathbf{t}: \langle x^{\mathbb{P}}, T \rangle} \mathbf{r}$, and either $L' = \mu \mathbf{t}'\widetilde{(\cdots)}.\mathbf{t}$ or $L' = \mu \mathbf{t}'\widetilde{(\cdots)}.\mathbf{t}\, \langle x := E' \rangle$

    Refined projection gives `end`, which erases to `end`.

    Applying inductive hypothesis on $G' \restriction_{\Phi, \mathbf{t}: \langle x^{\mathbb{P}}, T \rangle} \mathbf{r} = L'$. For either case of L', the erased type is of form $\ulcorner \mu \mathbf{t}'\widetilde{(\cdots)}.\mathbf{t} \urcorner = \ulcorner \mu \mathbf{t}'\widetilde{(\cdots)}.\mathbf{t}\, \langle x := E' \rangle \urcorner = \mu \widetilde{\mathbf{t}'}.\mathbf{t}$, which corresponds to the case in the basic projection to give `end`.

- Case $\mathbf{t}\, \langle x := E \rangle \restriction_{\Phi} \mathbf{r}$:

  Erasing LHS gives $\ulcorner \mathbf{t}\, \langle x := E \rangle \urcorner = \mathbf{t}$, and basic projection has only one case that $\mathbf{t} \restriction \mathbf{r} = \mathbf{t}$.

  - Sub-Case $\Phi(\mathbf{t}) = \langle x^{\mathbb{P}}, T \rangle$ and $\mathbf{r} \in \mathbb{P}$:

    Refined projection gives $\mathbf{t}\, \langle x := E \rangle$, which erases to $\mathbf{t}$, as required.

  - Sub-Case $\Phi(\mathbf{t}) = \langle x^{\mathbb{P}}, T \rangle$ and $\mathbf{r} \notin \mathbb{P}$:

    Refined projection gives $\mathbf{t}$, which erases to $\mathbf{t}$, as required.

- Case `end` $\restriction_{\Phi} \mathbf{r} =$ `end`:

  Erasing LHS gives $\ulcorner$`end`$\urcorner =$ `end`, and erasing RHS gives $\ulcorner$`end`$\urcorner =$ `end`.

  Basic projection `end` $\restriction \mathbf{r} =$ `end` holds. □

Erasure is a useful tool to retain properties of original MPST without refinement types, e.g. operational correspondence (Thm. 2.33). A global type can be specified and validated using refined MPST, while an implementation may apply erasure and operate at the level of basic MPST. We conclude the technical development of our refined multiparty session type theory here, and discuss related work in the next section.

3.8 Related Work

We first give a summary of related work that applies refinement types for verification and reasoning, and then summarise related session type that makes use of refinement types, dependent types, or other techniques to express extra constraints.

Refinement Types for Verification and Reasoning

Refinement types are initially introduced to allow recursive data structures to be specified in more details using predicates [FP91]. Many subsequent works on this topic [BBF+11; VSJ+14; SK16; VTC+17] utilise Satisfiability Modulo Theories (SMT) solvers, such as Z3 [DB08], to aid the type system deciding a subtyping relation using SMT encodings. A recent tutorial by Jhala and Vazou [JV21] covers development in this topic in great depth.

Refinement types have been applied to a number of different application domains, such as resource usage analysis [HVH19; KWR+20], information security [BFG10; BBF+11; SCF+13;

BCE+15; PSY+20; LKB+21], theorem proving [SHK+16; VTC+17], and verifying machine learning models [KKK+20; GLL+22] and smart contracts [NSC+22; TML+22]. In this chapter, we show how to use refinement types to specify and verify distributed protocols, by combining refinement and multiparty session types in a single framework.

Session Types with Refinement Types or Dependent Types

Bhargavan et al. [BCD+09] use refinement types to implement a limited form of multiparty session types. Session types are encoded in refinement types. The specification language they use, albeit similar to MPST, has limited expressive power. Only patterns of interactions where participants alternate between sending and receiving are permitted. Moreover, they do not study data dependencies in protocols, hence they can neither specify, nor verify constraints on payloads or recursions.

Bocchi et al. [BHT+10] propose a multiparty session π-calculus with logical assertions, where global and local types are extended to include *assertions*. A type system ensures that the assertion at type level must hold. While their work does not use refinement types or dependent types, we include it in the discussion here due to similarity in nature. Both our work and their work enable data constraints to be specified, but we choose a framework that is more type-based (i.e. using refinement types), instead of logic-based (i.e. using assertions). When projecting global types, we use silent prefixes to simplify projection, without the need to introduce existential quantifications, as is necessary in their work. In addition, in their work, a global protocol with assertions must be *well-asserted* [BHT+10, § 3.1]. In particular, the *history sensitivity* requirement states:

> A predicate guaranteed by a participant **p** can only contain those interaction variables that **p** knows.
>
> Bocchi et al. [BHT+10, p. 166]

Our theory lifts this restriction by allowing variables unknown to a sending role to be used in the global or local type, whereas such variables cannot be used in the implementation. For example, Item 1 in Ex. 3.4 fails the well-asserted requirement in [BHT+10]. In the refinement $x = z$ for variable z (for message label Trd), the variable x is not known to **C**, hence the protocol would not be well-asserted. In our setup, such protocol is permitted, the endpoint implementation for **C** can provide the value y received from **B** to satisfy the refinement type.

The subsequent extension by Bocchi et al. [BDY13] incorporates *virtual states* in the global type specifications. Each participant in a session keeps some variables in its virtual state, and may update the values at interactions with other participants. Our theory does not support the virtual states, yet the recursion variables can be considered as a limited form of such. However, their extension [BDY13] retains the limitation of history sensitivity from [BHT+10].

Toninho and Yoshida [TY17] extend MPST with value dependent types. Refinement types and value dependent types are also closely related concepts, both providing extra expressivity over a basic type system. In their work, the dependent type system verifies properties on values using witnesses in the form of proof objects, which need to be constructed manually. In our work, the refinement type system utilises an SMT solver, and thus dispenses with the need to construct proof objects manually. It is also worth noting that their work also retains some form of history sensitivity [BHT+10].

De Muijnck-Hughes et al. [dMBV19] propose an Embedded Domain Specific Language (EDSL) approach of implementing multiparty sessions in IDRIS. They use value dependent types in IDRIS to define combinators, with options to specify data dependencies. As a consequence, their work also requires construction of proof objects. This approach forms an example of a wider, *resource dependent* EDSL framework [dMBV20, § 4.3], where (communication) channels are the resources to be tracked.

Gheri et al. [GLS+22] propose a framework for designing multiparty protocols with assertions using an extended form of *choreography automata*. Choreography automata (initially proposed by Barbanera et al. [BLT20]) are an alternative way to describe multiparty protocols, using finite state machines under well-formedness conditions of *well-branched* and *well-sequenced*. The extended form supports assertions, and thus allows data and control flow constraints to be specified. In addition, they describe a technique to encode global type (with non-directed choices, i.e. $\mathbf{p} \to \mathbf{q}_i\{\cdots\}_{i \in I}$, where $\mathbf{q}_i$ can be different) using choreography automata. To our knowledge, there has been no comparison of expressivity of (asserted) choreography automata and (refined) global types.

Jongmans and van den Bos [JvdB22] focus on functional correctness of choreography programming using Hoare logic style predicate transformers. Choreography programming (initially proposed by Carbone and Montesi [CM13]) incorporates the implementation (i.e. processes, including the *values* of payload) at the global level, whereas global types merely describe the *types* of payload. While our system can mimic choreography programming by specifying a very specific refinement type on the payload, such that the required payload is enforced by the refinement type, their theory contains additional control flow constructs such as `while` loops and `if` conditionals, which we do not support directly.

In the setting of binary session types, Das and Pfenning [DP20] extend session types with arithmetic refinements, with applications to cost analysis for computing upper bounds of work for a given session type [DP22b]. Hinrichsen et al. [HBK19; HBK22] combine binary session types with concurrent separation logic, allowing reasoning about mixed-paradigm concurrent programs. They also plan to extend the framework to MPST. In a similar fashion, Swamy et al. [SRF+20] and Fromherz et al. [FRS+21] provide a framework of concurrent separation logic in $F^\star$, and demonstrate its expressivity by showing how (dependent) binary session types can be represented in the logic and used for verification and reasoning.

3.9 Summary and Concluding Remarks

In this chapter, we present a theory of refined multiparty session types, where we integrate refinement types into multiparty session types, in order to express data constraints in communication protocols. We show that the theory retains the guarantees of communication safety and liveness, and we will show in the next chapter how the theory can be implemented in a programming language.

The semantics we have given for the refined multiparty session types is synchronous, but we believe that the semantics can be extended to support asynchronous communication (with buffered communication, as in Communicating Finite State Machine (CFSM) [BZ83]) with the same operational correspondence result. There are two reasons to support this belief: *(1)* our extension does not add new behaviour of global and local types from the communication aspect, and thus the semantics[10] and the relation we have shown in this section can be adapted by separating send actions and receive actions, in the style of [DY13]; *(2)* the refinement types can be erased into base types, and we can then rely on the results from the original (asynchronous) multiparty session types from [DY13].

[10]For example, in global type reductions, a sending action gives full knowledge to the sending role, whereas the receiving role has perspective knowledge only — which requires changes in global typing contexts.

4 Implementing Refined Protocols in F*

In this chapter, we demonstrate how to practically implement refined protocols, based on the theory we introduced in § 3. For this, we propose a toolchain, named SESSION*, where we generate a callback-styled, refinement-typed Application Programming Interface (API) for implementing communication endpoints in F* [SHK+16].

We first introduce the workflow of our toolchain with an overview in § 4.1, as well as the running example we use throughout this chapter. In § 4.2, we provide a brief introduction and summarise the key features of F*, especially those we utilise for our generated code. Using our running example, we explain the generated APIs in F* in § 4.3, and how to implement endpoints in § 4.4. We outline the function we generate for executing the endpoint in § 4.5. We evaluate the performance of our generated code in § 4.6 with experiments.

The toolchain, SESSION*, was submitted as an accompanying artifact of [ZFH+20], and evaluated to be functional, reusable, and available. The artifact contains the source code of the toolchain, with examples and benchmarks used for the evaluation. The artifact is in the form of a Docker image, and is deposited at `https://zenodo.org/record/3970760`. The source files used for creating the artifact are available at `https://github.com/sessionstar/oopsla20-artifact`.

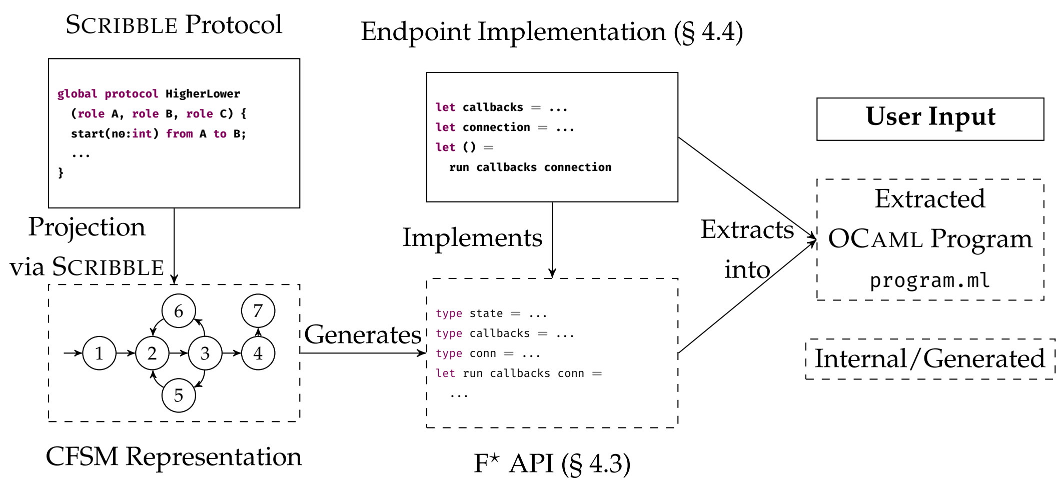
4.1 Toolchain Overview

Before introducing the toolchain, let us have a brief recap of the top-down design methodology of multiparty session types. The global protocol, describing the overall communication among participants, is first designed; and then projected into local types, which are used for implementing the individual endpoints.

The toolchain we propose in this chapter, SESSION*, follows this methodology. We present an overview of our toolchain in Fig. 4.1, where we distinguish user provided input by developers in $\boxed{\text{solid boxes}}$, from generated code or toolchain internals in dashed boxes.

Development begins with specifying a protocol using an *extended* SCRIBBLE protocol description language. SCRIBBLE is closely associated with the MPST theory [Hu17; NY19], and provides a user-friendly syntax for multiparty protocols. We have briefly introduced SCRIBBLE in § 2.2.3. We extend the SCRIBBLE toolchain[1] to support RMPST, allowing refinements to be added via annotations. The extended SCRIBBLE toolchain (as part of SESSION*) validates the well-formedness of the protocol, and produces a representation in the form of a *Communicating Finite State Machine (CFSM)* [BZ83] for a given participant.

[1]The SCRIBBLE part of the toolchain is mainly implemented by Raymond Hu, not the thesis author.



```
global protocol HigherLower
  (role A, role B, role C) {
  start(n0:int) from A to B;
  ...
}
```

```
let callbacks = ...
let connection = ...
let () =
  run callbacks connection
```

```
type state = ...
type callbacks = ...
type conn = ...
let run callbacks conn =
  ...
```

Figure 4.1: Overview of Toolchain

For each participant, we then use a code generator (also as part of SESSION*) to generate F* APIs from the CFSM, utilising a number of advanced type system features available in F* (explained in § 4.2). The generated APIs, described in detail in § 4.3, consist of various type definitions, and an entry point function taking *callbacks* (i.e. how to handle received data and how to provide data for sending) and *connections* (i.e. communication primitives) as arguments.

In our API design, we separate the concern of *communication* and *program logic*. The *callbacks*, corresponding to program logic, do not involve communication primitives — they are invoked to prompt a value to be sent, or to process a received value. Separately, developers provide a *connection*, which takes care of communication, allowing base types to be serialised/deserialised and transmitted to/from other participants. Developers implement the endpoint by providing both callbacks and connections, which will be explained in § 4.4, according to the generated refinement typed APIs.

Our generated entry point provides the gluing code to invoke the correct callbacks and communication primitives according to the local type. Thus, the developer can run the protocol by invoking the generated entry point. The F* source files can be verified using the F* compiler, and extracted to an OCAML program (or other supported targets) for efficient execution.

4.1.1 Running Example: A Number Guessing Game 'HigherLower'

We use a multiparty number guessing game, 'HigherLower', as our running example for demonstrating our toolchain. We describe this game in a SCRIBBLE protocol, a protocol description language that is closely related to multiparty session types (see § 2.2.3). The SCRIBBLE protocol of the game is shown in Fig. 4.2.

We first invite the reader to disregard annotations (strings after the @ sign in blue colour) for the moment being, while we introduce the communication structure. The game involves three

```
 1  global protocol HigherLower(role A, role B, role C) {
 2    // A tells B a secret number ‘n0‘,
 3    // and the number ‘t0‘ of attempts that C has to guess it
 4    start(n0:int) from A to B;   @'0<=n0<100'
 5    limit(t0:int) from A to B;   @'0<t0'
 6    do Aux(A, B, C);             @'B[n0, t0]'  }
 7  aux global protocol Aux(role A, role B, role C) @'B[n:int{0<=n<100}, t:int{0<t}]' {
 8    guess(x:int)             from C to B;      @'0<=x<100'         // Next guess by C
 9    choice at B {   higher() from B to C;      @'n>x && t>1'       // Secret is higher
10                    higher() from B to A;
11                    do Aux(A, B, C);           @'B[n, t-1]'
12             } or {   win()    from B to C;      @'n=x'              // C wins, A loses
13                    lose()   from B to A;
14             } or {   lower()  from B to C;      @'n<x && t>1'       // Secret is lower
15                    lower()  from B to A;
16                    do Aux(A, B, C);           @'B[n, t-1]'
17             } or {   lose()   from B to C;      @'n!=x && t=1'       // A wins, C loses
18                    win()    from B to A;
19  }          }
```

Figure 4.2: A *Refined* SCRIBBLE Global Protocol for a `HigherLower` Game

participants: **A** to setup the game, **B** to conduct the game, and **C** to play the guessing game. The game begins with **A** setting up the game by selecting a number `n0` to be guessed (in a `start` message to **B**, Line 4), and the number of guesses available `t0` (in a `limit` message to **B**, Line 5).

To win the guessing game, the player **C** needs to guess the number supplied by **A** correctly within the specified rounds. To make a guess, the player **C** sends a number to the conductor **B** in a `guess` message (Line 8). The conductor **B** inspects the number and provides a response: `win` (Line 12), in the case that the player **C** has made the correct guess and wins the game; `lose` (Line 17), in the case that the player **C** has used up all the chances and loses. In these two cases, the game concludes. Otherwise, the conductor **B** provides a hint before allowing the player **C** to make another guess: `higher` (Line 9), in case that the secret is higher than the previous guess; `lower` (Line 14), in case that the secret is lower than the previous guess. In these two cases, the player **C** can make another guess and the game continues.

In this snippet, recursive protocols are declared as *auxiliary* protocols using the keyword `aux` (e.g. Line 7), and invoked by the keyword `do` (e.g. Lines 6 and 11); whereas we have used the keywords `rec` and `continue` in § 2.2.3.

Now we draw attention to the annotations that contain the refinements and recursion variables. We want to be able to describe the guessing game in finer details, in particular, by imposing constraints on data values and control flow using refinements. For example, the number to be guessed is refined to be an integer between 0 (inclusive) and 100 (exclusive), as specified by the refinement `0<=n0<100` (Line 8). Additionally, the number of rounds of guesses must be a positive integer, as specified by the refinement `0<t0` (Line 5).

In the auxiliary protocol, we see two recursion variables located at **B**, `n` for the correct number to be guessed, and `t` for the round of guesses. When calling the auxiliary protocol initially, these two variables are set to be the values provided by messages from **A** (Line 6).

The refinements on a message can also impose a restriction at a choice in the protocol: when the conductor **B** responds to a guess by the player **C**. We see that the `higher` message carries a refinement `n>x && t>1` (Line 9), which requires the correct answer `n` to be greater than the guess `x`, and there are more than 1 guess remaining `t>1`. If the refinement cannot be satisfied, then the choice cannot be selected, thus acting as a control-flow constraint on the protocol. In a subsequent recursive call to the auxiliary protocol (Line 11), the number of remaining guesses is decremented to be `t-1`. The rest of the choices, `lower`, `win`, and `lose`, are refined similarly.

In the remainder of this chapter, we will use this running example to illustrate how we generate APIs for each endpoint, and how a develop can use our generated API to implement this protocol. But before that, we first give a brief introduction of our target programming language F⋆ in the next section.

4.2 Targeting the F⋆ Language for Implementation

F⋆ [SHK+16] is a verification-oriented, ML-like functional programming language with a rich set of type system features. We choose F⋆ as our target programming language due to its advanced type system. In this section, we give a brief overview of features of F⋆ that we utilise, and explain the outline of our generated APIs.

In our generated API, we utilise the F⋆ features explained as follows (along with reference to relevant part of the tutorial). For other features, interested readers can follow a more comprehensive (official) tutorial of F⋆ at `https://www.fstar-lang.org/tutorial/`.

Refinement Types A *refinement type*[2] in F⋆ has the form `x:t{e}`, where `t` is a base type (e.g. `int`), `x` is a variable that stands for values of this type, and `e` is a boolean expression[3] for *refinement*, possibly containing `x`. In short, the values of this refinement type are the *subset* of values of `t` such that the (boolean) refinement expression `e` evaluates to `true`. For example, natural numbers can be defined as `x:int{x≥0}`.

In F⋆, type-checking refinement types is done with the assistance of the Z3 SMT solver [DB08]. Refinements are encoded into SMT formulae and the solver decides the satisfiability of SMT formulae during type-checking. This feature enables automation in reasoning and saves the need for manual proofs in many scenarios.

[2] `https://www.fstar-lang.org/tutorial/book/part1/part1_getting_off_the_ground.html#boolean-refinement-types`

[3] The refinement expression `e` must be pure, total computation (to be explained later).

Dependent Functions with Effects A (dependent) function[4] in F* has a type of the form

$$\texttt{(x:t}_1\texttt{)} \rightarrow \texttt{E t}_2,$$

where $\texttt{t}_1$ is the argument type, `E` describes the *effect* of the function, and $\texttt{t}_2$ is the result type, which may also refer to the argument `x`. For example, a function taking an integer and returning an integer greater than the argument can be assigned the type:

$$\texttt{(x:int)} \rightarrow \texttt{Tot y:int\{y>x\}}.$$

The effect `Tot` in the type indicates the function is total[5]. In fact, `Tot` is the default effect, for terminating and side-effect free computations.

On the permissive end is the arbitrary effect `ML` (corresponds to all possible effects in an ML language), which permits state mutation, non-terminating recursion, I/O, exceptions, etc.

Code Extraction Being a verification-oriented programming language, F* provides functionalities to prove properties about programs. Moreover, verified F* code, after successful type-checking, can be *extracted* into a different programming language for execution[6]. In our case, we extract verified F* programs into OCAML for execution. Additionally, F* supports extraction to F$^\#$ and C [PZR+17].

The Ghost Effect and erased Types A type can be marked `erased`[7] in F*, so that values of such types are not available for computation (after extraction), but only for proof purposes (during type-checking).

The type constructor `erased` is accompanied with the ghost effect to mark computationally irrelevant code. The type system prevents the use of erased values in computationally relevant code, so that the values can be indeed safely erased.

In the following snippet, we quickly demonstrate this feature: the effect `GTot` stands for ghost and total, and cannot be mixed with the default pure effect `Tot` (i.e. the function `not_allowed` does not type-check).

```
(* the field ‘x1‘ is a normal integer and ‘x2‘ is an erased integer *)
type t = { x1: int; x2: erased int; }

(* Signature of ‘reveal‘ in standard library *)
val reveal: erased 'a →  GTot 'a

(* Accessing an erased value at value level is not allowed *)
let not_allowed (o: t) = reveal o.x2
(* Accessing at type level is allowed *)
val allowed: (o: t{reveal o.x2 ≥ 0}) → int
```

[4]https://www.fstar-lang.org/tutorial/book/part1/part1_getting_off_the_ground.html#functions
[5]https://www.fstar-lang.org/tutorial/book/part4/part4_computation_types_and_tot.html
[6]https://www.fstar-lang.org/tutorial/book/part1/part1_execution.html
[7]https://www.fstar-lang.org/tutorial/book/part4/part4_ghost.html

In our case, we use the `erased` types to mark variables *unknown* to an endpoint, whose values are not known due to not being a party of the message interaction. For example, in our running example (Fig. 4.2), the player **C** does not know the value of the number to be guessed `n0`, but nonetheless knowns its type from the protocol.

Now that we have introduced the key features of F*, we are ready to give a brief overview of our generated code. We follow the code generation approach using Communicating Finite State Machine (CFSM) [HY16]. To implement an endpoint for a participant, our generated code (as shown in Fig. 4.1, F* API) consists of:

State Types: Allowing developers to access variables known at a given state.

Callbacks: A record type consisting of functions that correspond to CFSMs transitions. These functions implement program logic.

Connections: A record type consisting of functions for sending/receiving base values to/from other participants. These functions handle the communication aspects.

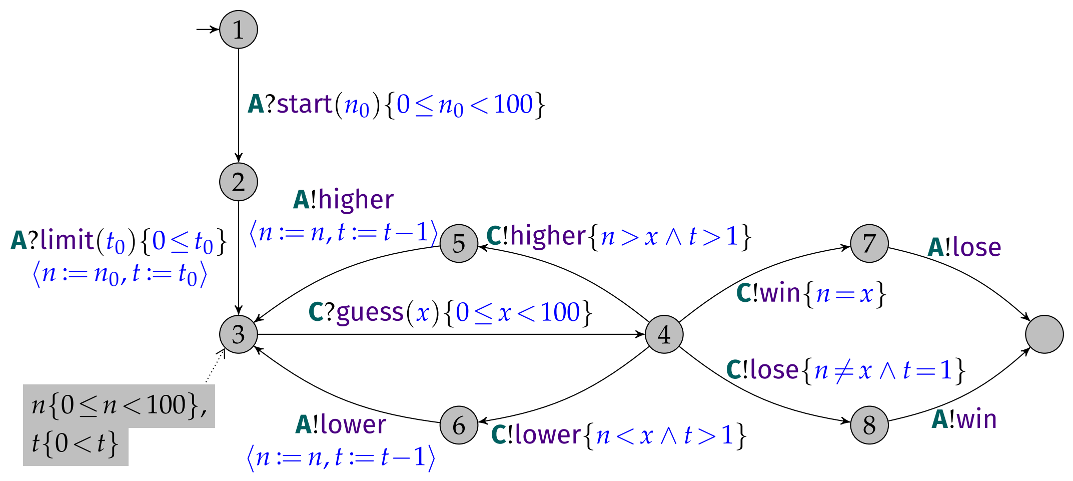
Entry Point: A generated function that takes callbacks and connections as arguments, and executes the endpoint CFSM by invoking provided callback functions and connection functions.

To implement an endpoint, a developer needs to provide implementations of the generated callback and connection types, by providing appropriate functions to handle program logic and communication aspects. The F* compiler checks whether the implemented functions type-check against the prescribed types. Passing the callback and connection record to the generated entry point function, the endpoint is implemented. In the subsequent sections, we explain the design of the API in detail, and how a developer can use these APIs to implement an endpoint.

4.3 Projection and F* Callback API Generation

After the protocol is specified in SCRIBBLE, the next step of the workflow (Fig. 4.1) is to *project* the *refined* global protocol onto each role. Although all endpoints together must comply to global protocol, projection allows each endpoint to be *separately* implemented and verified, a key requirement for practical distributed programming. The formal definition of projection has been given in § 3.4, so we won't repeat the details here.

Here, we use our running example of `HigherLower` (Fig. 4.2), and focus on the conductor role **B**, and the player role **C** to explain how we generate the APIs in F*. We use a Communicating Finite State Machine (CFSM)-based code generation approach, following [HY16]. The connection between local types and CFSMs has been studied by Deniélou and Yoshida [DY13]. While we have not formally extended this connection with refinement types, the connection is helpful for us to design and generated refined APIs.


1
A?start(n_0){$0 \le n_0 < 100$}
2
A?limit(t_0){$0 \le t_0$}
$\langle n := n_0, t := t_0 \rangle$
A!higher
$\langle n := n, t := t-1 \rangle$
5
C!higher{$n > x \wedge t > 1$}
7
A!lose
3
C?guess(x){$0 \le x < 100$}
4
C!win{$n = x$}
C!lose{$n \ne x \wedge t = 1$}
$n\{0 \le n < 100\}$,
$t\{0 < t\}$
A!lower
$\langle n := n, t := t-1 \rangle$
6
C!lower{$n < x \wedge t > 1$}
8
A!win

(a) CFSM Representation

Generated F⋆ API

State	Transition	Generated Callback Type
1	A?start	`s1 → (n:int{0≤n<100}) → ML unit`
2	A?limit	`s2 → (t:int{0<t}) → ML unit`
3	C?guess	`s3 → (x:int{0≤x}) → ML unit`
4	[multiple]	`(s:s4) → ML (s4Cases s)`
5	A!higher	`s5 → ML unit`
6	A!lower	`s6 → ML unit`
7	A!lose	`s7 → ML unit`
8	A!win	`s8 → ML unit`

(b) Generated I/O Callback Types

```
type s4Cases (s:s4) =
| s4_lower of
  unit{s.n<s.x ∧ s.t>1}
| s4_lose of
  unit{s.n≠s.x ∧ s.t=1}
| s4_win of unit{s.n=s.x}
| s4_higher of
  unit{s.n>s.x ∧ s.t>1}
```

(c) Generated Sum Type for the Internal Choice at State 4

Figure 4.3: CFSM Representation and F⋆ API Generation for the Conductor **B** in `HigherLower`

Projection onto the Conductor **B**

We first look at the projection of the protocol onto the conductor **B**: it is unique among all participants, because the conductor **B** is involved in *every* interaction of the protocol, and (consequently) **B** has *explicit* knowledge of the value of *every* payload variable during execution. We take advantage of this unique circumstance to take away the consideration of multiplicities for the moment being. Handling *implicit* knowledge will be demonstrated later when we consider the player **C**.

We show in Fig. 4.3a the representation of local type used in our toolchain based on Communication Finite State Machines (CFSMs) [BZ83]. The transitions of the CFSMs are communication actions, where ! stands for output actions (i.e. sending), and ? stands for input actions (i.e. receiving). For example, **A**?start(n_0){$0 \le n_0 < 100$} is an input action of a start message from **A**, with an `int` payload n_0 with refinement $0 \le n_0 < 100$. Similarly, **C**!higher{$n > x \wedge t > 1$}

expresses a refinement on an output action of a `higher` message to **C**. For brevity, we omit the payload data types in the CFSM transitions, as this example features only `int` s; we omit empty payloads () likewise.

We show the recursion variables (declared on Line 7 of Fig. 4.2) on the corresponding CFSM state (shaded in grey, with an arrow to the state).

Refined API Generation for the Conductor B

CFSMs offer an intuitive understanding of the semantics of local types, and we use them as the basis for code generation, a technique commonly used in recent work [HY16; NHY+18; CHJ+19]. We highlight a novel and crucial development: we exploit the approach of *type* generation to produce functional-style *callback*-based APIs that *internalise* all of the actual communication channels and I/O actions.

In short, the transitions of the CFSM are rendered as a set of *transition-specific* function types to be implemented by the user — each of these functions take and return only the *user-level data* related to I/O actions and the running of the protocol. The transition functions of the CFSM are embedded into a generated entry point, exporting a clean interface to execute the protocol by calling the appropriate user-supplied functions according to the current CFSM state and I/O event occurrences.

We continue with our example, Fig. 4.3b lists the callback function types for **B**, detailed as follows. Note, a key characteristic of MPST-based CFSMs is that each non-terminal state (i.e. a state with outgoing transitions) is either input-only or output-only.

State Types For each state, we generate a separate type (named by enumerating the states, by default). Each state type is defined as a record type containing previously known payload values and any recursion variables, or `unit` if none. For example, the state type for state 3 is given as follows:

```
type s3 = { n0: int{0≤n0<100}; t0: int{0<t0}; n: int{0≤n<100}; t: int{0<t} }
```

Basic I/O Callbacks We explain how we generate callback function types for input (i.e. receiving) transitions and *singleton* output (i.e. sending) transitions. The generation of multiple output transitions (i.e. an *internal choice*) will be explained afterwards.

For each input transition we generate a function type:

```
st → x:t → ML unit
```

where `st` is the predecessor state type, and `x:t` is the refined payload type received. The return type is `unit` and the function can perform any side effects (under the permissive `ML` effect), so that the input handling callback is able to modify global state, interact with the console, instead of merely pure computation. However, it is not recommended to interfere with the control flow when implementing the callbacks (e.g. using exceptions or call/cc), as such effects may invalidate the guarantees about communication safety.

If an input transition is fired during execution, the generated entry point function will invoke a user-supplied function of this type with a value of the state type `st`, and the received payload value. Note that refinements in protocols are embedded into the refinement types of the payload variables, and stored under the appropriate state type.

Similarly, for each output state with a *single* outgoing transition, we generate a function type:

```
st → ML t
```

for the output providing callback, where `st` is the predecessor state type, `t` is the refined type for the output payload. If an output state with a single outgoing transition is reached during execution, the generated entry point function will invoke a user-supplied function of this type with a value of the state type `st`. The return value is used to fire an output transition. Note that output states with *multiple* output transitions are handled differently, explained below.

Internal Choices For each output state with more than one outgoing transition, we generate an additional sum type with the cases of the choice, e.g. Fig. 4.3c for state 4 of the CFSM. This sum type `s4Cases` is indexed by a value `s` of the corresponding state type `s4` to make variables inside the state type available refinements of each case. The constructors of the sum type corresponds to each label of the internal choice.

We then generate a single function type for this state for the output providing callback:

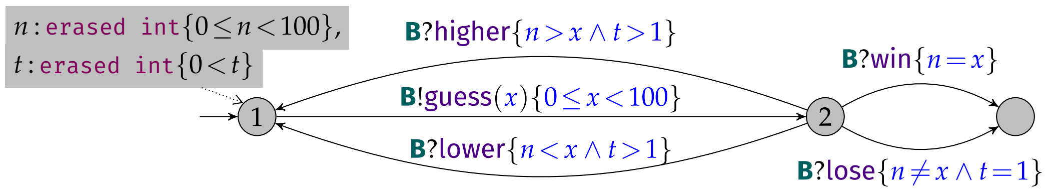
```
(s:st) → ML (stCases s)
```

where `st` is the predecessor state type, and `stCases` is the generated sum type for this output state. Recall that the generated case is indexed by the predecessor state, so we need to provide one in the type. The user-supplied function of that type makes the internal choice by returning a corresponding constructor. The constructors in the sum type incorporate the payload values, which need to satisfy refinements (if any). For example, when the player **C** provides a correct guess, we would have `s.n=s.x`, thus only a `s4_win` value can be constructed, since the refinements of all other constructors cannot be satisfied.

If an output state with a single outgoing transition is reached during execution, the generated entry point function will invoke a user-supplied function of this type with a value of the state type `st`. The return value is used to fire an output transition, and the state machine is advanced according to the constructor of the returned value.

Projection and API Generation for the Player C

The implementation for the player **C** raises an interesting question relating to the treatment of refinement: how should we handle refinements on variables that the target role does *not* itself *know*? The player **C** does not (and should not) know the value of the secret `n` (otherwise this game would be trivial), but they do know that related information *exists* in the protocol and is subject to the specified refinement. We have explained how we model these 'latent' information with silent prefixes in § 3.4, and now we need to model multiplicities of variables.



n : erased int{$0 \le n < 100$},
t : erased int{$0 < t$}
1
2
B?higher{$n > x \wedge t > 1$}
B!guess(x){$0 \le x < 100$}
B?lower{$n < x \wedge t > 1$}
B?win{$n = x$}
B?lose{$n \ne x \wedge t = 1$}

(a) CFSM Representation

State	Transition	Generated Callback Type	Example Implementation
1	**B**!guess	`s1 → ML (x:int{0≤x<100})`	`fun _ → !next (* Deref. next *)`
2	**B**?higher	`s2 → unit{^n>x∧^t>1} → ML unit`	`fun s → next := s.x + 1`
	B?lower	`s2 → unit{^n<x∧^t>1} → ML unit`	`fun s → next := s.x - 1`
	B?win	`s2 → unit{^n=x} → ML unit`	`fun _ → ()`
	B?lose	`s2 → unit{^n≠x∧^t=1} → ML unit`	`fun _ → ()`

where ^ is a shorthand notation for `reveal`, which extracts an erased value, and the mutable global reference `next` is defined as:

```
(* ‘ref‘ indicates a reference type, and ‘alloc‘ create a new reference *)
let next: ref (x:int{0≤x<100}) = alloc 50
```

(b) Generated I/O Callback Types and Example Implementation

Figure 4.4: CFSM Representation and F⋆ API Generation for the Player **C** in `HigherLower`

We utilise `erased` variables in F⋆ (described briefly in § 4.2) to retain the latent information, as illustrated by the annotation on state 1 in Fig. 4.4a. As mentioned earlier, the type-checker can still take advantage of the refined types of erased variables during type-checking for proving desired properties. Moreover, the type-checker will prevent the usage of `erased` variables in the implementation.

We show the generated callback types in Fig. 4.4b. The example implementation is to be explained in the next section, and readers can ignore the implementation for now. The callback type generation process is largely the same as before, yet we need to inject calls to the standard library `reveal` (noted as ^ for brevity) when describing the refinements. Again, `erased` variables can only be used at the type level, and we can safely do so in the generated callback types.

After explaining generated callback types, we show next how to use these generated APIs to implement an endpoint participant in the next section.

4.4 Implementation Via Generated Refinement-Typed APIs

We show how to use the generated APIs to implement the endpoint processes. As mentioned previously, we generate callback types for users to implement the program logic, and the user implements them as callback functions of the generated types. These functions are supplied to the generated entry point, along with primitives for communicating with other peers (i.e. connections).

Connection APIs

We have previously introduced the generation for callback APIs, but not the connection APIs. The connection APIs provide the communication primitives to interact with peers.

The *connection* type is a record type with functions for sending and receiving base types. The primitives for communications are collected in a record with fields as follows (`S` is ranged over by base types `int`, `bool`, `unit`, etc.):

$$\texttt{send\_S} : (\!|\,\mathbb{P}\,|\!) \to (\!|\,\texttt{S}\,|\!) \to \texttt{ML unit} \qquad\qquad \texttt{recv\_S} : (\!|\,\mathbb{P}\,|\!) \to \texttt{unit} \to \texttt{ML}\ (\!|\,\texttt{S}\,|\!)$$

where $(\!|\,\mathbb{P}\,|\!)$ is a generated sum type for other peers and $(\!|\,\texttt{S}\,|\!)$ is the data type for the base type `S` in F⋆. The communication primitives do not use refinement types in the type signature. We can safely do so by exploiting the property that refinements can be erased at runtime after static type-checking.

Generated Entry Point

The generated entry point (named `run`), takes the callbacks and connections to run the CFSM for the endpoint. We demonstrate here how to use it without explaining its internals (which we defer to § 4.5). The usage of the generated entry point is demonstrated by the code snippet in Fig. 4.5a, which bootstraps a player **C** endpoint. First, we establish a connection with the conductor **B** (note that the player does not need to communicate with role **A**), using Transmission Control Protocol (TCP). Then, we use a helper function `mk_conn` to convert the TCP connection to the communication primitives needed by the generated entry point. Assuming a record `callbacks` containing the required functions (static typing ensures all are covered), the entry point is invoked with both parameters, which will run the CFSM. The API takes care of endpoint execution by monitoring the channels, and calling the appropriate callback and communication primitives, based on the current state and I/O event occurrences.

Example Implementations of Callbacks

For example, a minimal, well-typed implementation of the conductor **B** could comprise the internal choice callback in Fig. 4.5b (cf. state 4 of Fig. 4.3b), and an empty function for all other callbacks (i.e. `fun _ → ()`). We can highlight how protocol violations are ruled out by static refinement type-checking, which is ultimately the practical purpose of our work. In the above callback implementation, changing, for instance, the condition for the `lose` case to `s.t=0` (as an example of an off-by-one error) would violate the refinement on the `s4_lose` constructor, cf. Fig. 4.3c. Alternatively, omitting the `lose` case altogether would break both the `lower` and `higher` cases, as the type-checker would not be able to prove `s.t>1` as required by the subsequent constructors.

```
let main () =
  (* connect to B via TCP *)
  let server_B = connect ip_B port_B
    in
  (* Setup connection from TCP *)
  let conn = mk_conn server_B in
  (* Invoke the Entry Point 'run' *)
  let () = run callbacks conn in
  (* Close TCP connection *)
  close server_B
```

(a) Running the Player **C**

```
(* Signature (s:s4) → ML (s4Cases s) *)
fun (s:s4) →
  (* Win if guessed number is correct *)
  if s.x=s.n then s4_win ()
  (* Lose if running out of attempts *)
  else if s.t=1 then s4_lose ()
  (* Otherwise give hints accordingly *)
  else if s.n>s.x then s4_higher ()
  else s4_lower ()
```

(b) Implementing the Internal Choice for the Conductor **B**

Figure 4.5: Selected Snippets of Endpoint Implementation

For another example, Fig. 4.4b provides an example implementation for the player **C**. This implementation guesses the secret by a simple search, since we know the target value lies within the specified interval. We declare a global mutable reference `next`, initialised to 50, which will be updated on receiving any `higher` or `lower` hints.

In the input callback for `higher`, we increase the `next` value by 1. In order for the update of `next` to type-check, the new value must satisfy the refinement on its type (i.e. greater or equal to 0 and less than 100). Given that the value being assigned is one more than the existing value, it might have been the case that the new value falls out of the range (i.e. if `next` is 99), hence violating the prescribed type. Despite that the value of `n` is unknown, we have known from the refinement attached to the transition that `^n>x` holds, hence it must have been the case that our last guess `x` is strictly less than the secret `n`, which rules out the possibility that `x` can be 99 (the maximal value of `n`). Had the refinement and the erased variable not been present, the type-checker would not be able to accept this implementation, and it demonstrates that our encoding allows such reasoning with latent information from the protocol.

Moreover, the type and effect system of F* prevents the `erased` variables from being used in the callbacks, since the callbacks are computationally relevant. On one hand, `int` and `erased int` types are not compatible, because they are not the same type. This prevents an `erased` variable from being used in place of a concrete variable. On the other hand, the function `reveal` can convert a value of `erased 'a` to a value of `'a` with `GTot` effect. A function with `GTot` effect *cannot* be mixed with a function with `ML` effect (as in the case of our callbacks), so `erased` variables cannot be revealed in the callback implementation.

We invite interested readers to try the running example out with our accompanying artifact. In the artifact, we propose a few modifications on the implementation code and the protocol, and invite readers to observe errors being raised when implementations no longer conform to the prescribed protocol.

In the next section, we give inner working details of the generated entry point, and explain how it executes the CFSM.

4.5 Executing the Communicating Finite State Machine in the Generated Entry Point

As mentioned earlier, our API design separates the concern of program logic (with callbacks) and communication (with connections). A crucial piece of the generated code, the entry point, involves *threading* the two parts together: the execution function performs the communications actions and invokes the appropriate callbacks for handling. In this way, we do *not* expose explicit communication channels, so channel linearity (i.e. a communication channel is used exactly once) can be achieved with ease by construction in our generated code.

The entry point function, named `run`, takes callbacks and connections as arguments, and executes the CFSM for the specified endpoint. The signature uses the permissive `ML` effect, since communicating with the external world performs side effects. We traverse the states (denoted $\mathbb{Q}$) in the CFSM and generate appropriate code depending on the nature of the state and its outgoing transitions.

Internally, we define mutually recursive functions for each state $q \in \mathbb{Q}$, named $\mathtt{run}_q$. Each function takes the state record $(\!| q |\!)$ as argument, (where $(\!| q |\!)$ stands for the state type for a given state q), and performs the required actions at that state q. The run state function for a state q either (1) invokes callbacks and communication primitives for the correct transition, then calls the run state function for a successor state q'; or, (2) terminates if q is a terminal state (without outgoing transitions). Then, the generated entry point invokes the run function for the initial state q_0, to start the finite state machine.

The internal run state functions are not exposed to the developer, hence it is not possible to tamper with the internal state with usual means of programming. This allows us to guarantee linearity of communication channels by construction. We outline how to run each state, depending on whether the state is a sending state or a receiving state. Note that CFSMs constructed from local types do not have mixed states [DY13, Prop. 3.1]. We show the template for generating the run functions in Fig. 4.6. In this template, we assume the callbacks are provided by the variable `callbacks`, and connections are provided by the variable `comm`; and the state type $(\!| q |\!)$ is represented as $\mathtt{state}q$ (as seen in the argument of the run functions).

4.5.1 Running the CFSM at a Sending State

We give a template for the run function for a sending state in Fig. 4.6a. For a sending state $q \in \mathbb{Q}$, the developer makes an internal choice on how the protocol proceeds via the callback, among the possible outgoing transitions. This is done by invoking the sending callback $\mathtt{state}q\mathtt{\_send}$ with the current state record `st`, to obtain the choice made by the developer along with the

```
let rec run_q (st: stateq) =
  let choice =
    callbacks.stateq_send st
  in
  match choice with
  | Choiceqℓᵢ payload →
    comm.send_string p "ℓᵢ";
    comm.send_Sᵢ p payload;
    let st =
      { ⋯; xᵢ=payload }
    in
    run_q'ᵢ st
```

Repeat for $i \in I$

(a) Template for Running a Sending State q

```
let rec run_q (st: stateq) =
  let label = comm.recv_string p () in
  match label with
  | "ℓᵢ" →
    let payload = comm.recv_Sᵢ p () in
    assume Eᵢ;
    callbacks.stateq_receive_ℓᵢ st payload;
    let st =
      { ⋯; xᵢ=payload }
    in
    run_q'ᵢ st
```

(b) Template for Running Receiving State q

Figure 4.6: Template for Run State Function $\texttt{run}_q$

associated payload. We pattern match on the constructor with the label ℓ_i of the message, and find the corresponding successor state q_i'.

The label ℓ_i itself is encoded as a `string` (we discuss the encoding options later in § 4.5.3) and sent via the sending primitive `send_string` to the communication partner **p**. It is followed by the payload specified in the return value of the callback, via corresponding sending primitive $\texttt{send\_S}_i$ for the base type $\texttt{S}_i$. The sending primitive operates on the base types, and does not concern the refinement on the payload.

We then construct a state record of $⦇ q_i' ⦈$ from the existing record $⦇ q ⦈$, adding the new field x_i returned from the callback in the choice constructor. In the case of recursive protocols, we also update the recursion variable according to the definition in the protocol when constructing $⦇ q_i' ⦈$. Finally, we call the run state function $\texttt{run}_{q_i'}$ to continue the CFSM, effectively making the transition to state q_i'.

4.5.2 Running the CFSM at a Receiving State

We give a template for the run function for a receiving state in Fig. 4.6b. For a receiving state $q \in \mathbb{Q}$, how the protocol proceeds is determined by an external choice, among the possible outgoing actions. To know what choice is made by the other party, we first receive a string and decode it into a label ℓ_i, via the receiving primitive for string `recv_string`.

Subsequently, according to the label ℓ_i, we look up the label in the possible transitions, and find the corresponding successor state q_i'. By invoking the appropriate receiving primitive $\texttt{recv\_S}_i$, we obtain the payload value. We note that the receiving primitive has a return type *without* refinements, therefore the value would not type-check against the prescribed refinement type, due to the absence of refinements. We assume that the refinements have been validated on the sending side, following the procedure described in § 4.5.1. Therefore, we use the F* built-in `assume`[8] to reinstate the refinements according to the protocol before using the value: the built-in function `assume` allows the type-checker to *admit* the given boolean expression to be valid without verifying.

According the label ℓ_i received, we can now call the receiving callback `state`q`_receive_`ℓ_i with the received value `payload`. This allows the developer to process the received value and perform any relevant program logic. The same procedure for constructing the state record for the next state q_i' is followed, and then the run function for the next state q_i' is invoked to transition to the successor state.

4.5.3 Discussion: Encoding Message Labels

Encoding the message label ℓ and transmitting it as a `string` can be seen as a generalisation of the approach used by Pucella and Tov [PT08]. In [PT08, § 3], communication of binary choices (i.e. a choice of left and right), in binary session types, is encoded via a `bool` value being exchanged in the underlying channel. The unusual effect of the (unexposed) `bool` message is that it can affect the usage of the channel, since the choice, encoded as a `bool`, determines the continuation. Similarly, in our setup, the label ℓ of a message is encoded a `string`, and can determine the appropriate continuation.

A potential optimisation is whether we can safely omit the transmission of labels in the case of a singleton choice, as in done in [PT08] for the binary session type. Unfortunately, the question must be answered in negative due to the *merging* behaviour: consider a global type G as follows (omitting payloads for simplicity):

$$G = \mathbf{A} \to \mathbf{B} \begin{Bmatrix} \ell_{1a}.\mathbf{B} \to \mathbf{C} : \ell_{1b} \\ \ell_{2a}.\mathbf{B} \to \mathbf{C} : \ell_{2b} \end{Bmatrix}.$$

We note that the sending types for **B** with regards to labels ℓ_{1b} and ℓ_{2b}, are indeed singletons. According to the global type, the participant **B** is instructed to send *the* appropriate label according to what they have received. The picture is different from the receiving side at **C**, where they have *two* cases to consider: namely labels ℓ_{1b} and ℓ_{2b}. This would cause a mismatch in the communication and we have an unsafe use of the communication channel. This merging behaviour is tricky to handle, and poses interesting challenges for implementing MPST in practical programming (cf. [BHY+23, § 6.3])

As a consequence, we may not safely omit the transmission of the message labels in the general case. Failing so, we also note that encoding the labels as `int` locally at each choice, e.g. by numbering each choice in order, is also potentially unsafe, as it suffers from the same caveat. However, it is possible to do so globally by mapping all the labels in a given global type

[8] https://www.fstar-lang.org/tutorial/book/part1/part1_prop_assertions.html#assumptions

to `int`s instead of using them as `string`s directly.

To be on the safe side, our implementation encodes the label as a `string` and transmits the label with the payload at all times. For a possible optimisation, the labels can indeed be omitted for both when the *receiving* side is a singleton choice, where no mismatch would occur. In addition, all the labels in a global type can undergo a translation process into `int`s during code generation, in order to reduce the actual amount of data transferable.

4.6 Performance Evaluation

We evaluate the performance of our toolchain SESSION⋆. We describe the methodology and setup (§ 4.6.1), and comment on the compilation time (§ 4.6.2) and the execution time (§ 4.6.3). The source files of the benchmarks used in this section are included in our artifact, along with a script to reproduce the results.

4.6.1 Methodology and Setup

We measure the time to generate the CFSM representation from a SCRIBBLE protocol (*CFSM*), and the time to generate F⋆ code from the CFSM representation (F⋆ *APIs*). Since the generated APIs in F⋆ need to be type-checked before use, we also measure the type-checking time for the generated code (*Gen. Code*). Finally, we provide a simple implementation of the callbacks and measure the type-checking time for the callbacks against the generated type (*Callbacks*).

To execute the protocols, we need a network transport to connect the participants by providing appropriate sending and receiving primitives. In our experiment setup, we use the standard library module `FStar.Tcp` to establish TCP connections between participants, and provide a simple serialisation/de-serialisation module for base types. Due to the small size of our payloads, we set `TCP_NODELAY` to avoid the delays introduced by the congestion control algorithm[9]. Since our entry point to execute the protocol is parameterised by the connection/transport type, an implementation may use other connections if developers wish, e.g. an in-memory queue for local setups. We measure the execution time of the protocol (*Execution Time*).

To measure the execution overhead of our generated code, we compare the callback-based implementation against an implementation of the protocol without session types or refinement types, which we call *bare implementation*. In the bare implementation, we use the same sending and receiving primitives (i.e. `connection`) as in the generated code. The bare implementation uses a series of direct calls of sending and receiving primitives, for the same communication pattern, but does not use any generated callback APIs.

We use a Ping Pong protocol (Fig. 4.7), parameterised by the protocol length, i.e. the number of Ping Pong messages n in a protocol iteration. When the protocol length n increases, the number of CFSM states increases linearly, which gives rise to longer generated code and larger

[9]https://en.wikipedia.org/wiki/Nagle%27s_algorithm

```
global protocol PingPong_n(role A, role B) {
  choice at A { Ping(x_1:int) from A to B;
                Pong(y_1:int) from B to A; @"y_1>x_1"
                Ping(x_2:int) from A to B; @"x_2>y_1"
                Pong(y_2:int) from B to A; @"y_2>x_2"
                ...
                Ping(x_n:int) from A to B; @"x_n>y_{n-1}"
                Pong(y_n:int) from B to A; @"y_n>x_n"
                do PingPong_n(A, B); }
           or { Bye() from A to B;
                Bye() from B to A; } }
```

Figure 4.7: Ping Pong Protocol (Parameterised by Protocol Length n)

Table 4.1: Time Measurements for Ping Pong Protocol

Protocol	Generation Time		Type Checking Time		Execution Time
Length (n)	CFSM	F⋆ APIs	Gen. Code	Callbacks	(100 000 ping-pongs)
bare	n/a	n/a	n/a	n/a	28.79s
1	0.38s	0.01s	1.28s	0.34s	28.75s
5	0.48s	0.01s	3.81s	1.12s	28.82s
10	0.55s	0.01s	14.83s	1.34s	28.84s
15	0.61s	0.01s	42.78s	1.78s	n/a
20	0.69s	0.02s	98.35s	2.54s	28.81s
25	0.78s	0.02s	206.82s	3.87s	28.76s

generated types. In each Ping Pong message, we include payload of increasing numbers, and encode the constraints as protocol refinements.

We study its effect on the compilation time (§ 4.6.2) and the execution time (§ 4.6.3). We run the experiment on varying sizes of n, up to 25. Unfortunately, larger sizes of n leads to unreasonably large resource usage during type-checking in F⋆. Table 4.1 reports the results for the Ping Pong protocol in Fig. 4.7.

We run the experiments under a network of latency of 0.340ms (64 bytes ping), and repeat each experiment 30 times. Measurements are taken using a machine with Intel i7-7700K CPU (4.20 GHz, 4 cores, 8 threads), 16 GiB RAM, operating system Ubuntu 18.04, OCaml compiler version 4.08.1, F⋆ compiler commit `8040e34a`, Z3 version 4.8.5.

4.6.2 Compilation Time

CFSM and F⋆ Generation Time

We measure the time taken for SCRIBBLE to generate the CFSM from the protocol in Fig. 4.7, and for the code generation tool to convert the CFSM to F⋆ APIs. We observe from Table 4.1 that the generation time for CFSMs and F⋆ APIs is short. It takes less than 1 second to complete the generation phase for each case.

Type-checking Time of Generated Code and Callbacks

We measure the time taken for the generated APIs to type-check in F⋆. We provide a simple F⋆ implementation of the callbacks following the generated APIs, and measure the time taken to type-check the callbacks.

The increase of type-checking time is non-linear with regard to the protocol length. We encode CFSM states as records corresponding to local typing contexts. In this case, the size of local typing contexts and the number of type definitions grows linearly, giving rise to a non-linear increase. Moreover, the entry point function also is likely to cause non-linear increases in the type-checking time.

The long type-checking time of the generated code could be avoided, if the developer chooses to trust our toolchain to always generate well-typed F⋆ code for the entry point. The entry point would be available in an *interface file* that defines the signatures of functions, with the actual implementation is provided in OCAML instead of F⋆[10]. There would otherwise be no changes in the development workflow.

The type-checking time of the callback implementation does not fit a linear pattern either, yet it remains within a reasonable time frame.

4.6.3 Runtime Performance (Execution Time)

We measure the execution time taken for an exchange of 100,000 ping pongs for the generated code and bare implementation under the experiment network. The execution time is dominated by network communication, since there is little computation to be performed at each endpoint.

We provide a bare implementation using a series of direct invocations of sending and receiving primitives, in a compatible way to communicate with generated APIs. The bare implementation does not use callbacks, which we anticipate to run faster, since the bare implementation involves handling fewer function pointers when calling callbacks. Moreover, the bare implementation does not construct *state records*, which record a backlog of the communication, as the

[10]Defining signatures in an interface file, and then providing an implementation in the target language OCAML makes the F⋆ compiler to *assume* the implementation is correct, as the compiler cannot verify the implementation in the target language. This technique is used frequently in the standard library of F⋆. This is not to be confused with implementing the endpoints in OCAML instead of F⋆, as that would bypass the F⋆ type-checking.

protocol progresses. To measure the performance impact of book-keeping of callback and state records, we run the Ping Pong protocol from Fig. 4.7 for a protocol of increasing size (number of states and generated types), i.e. for increasing values of n. All implementations, including *bare* are run until 100,000 ping pong messages in total are exchanged[11].

We summarise the results in Table 4.1. Despite the different protocol lengths, there are *no significant changes* in execution time. Since the execution is dominated by time spent on communication, the measurements are subject to network fluctuations, which are difficult to avoid during the experiments. We conclude that our generated code does not impose a large overhead on the execution time.

4.7 Related Work

We begin with the most closely related work: Session Type Provider (STP) by Neykova et al. [NHY+18]. They implement MPST *with assertions* using the SCRIBBLE toolchain targeting F$^\#$. Their main contribution is to use *type providers* [PGS16], a language feature of F$^\#$ for compile-time code generation, for implementing multiparty protocols in SCRIBBLE. Additionally, they extend SCRIBBLE to allow interaction refinements in protocols for better expressivity, which makes their work closely related to ours.

Their implementation, the Session Type Provider (STP), relies on code generation of fluent (class-based) APIs, following the technique described by Hu and Yoshida [HY16]: each protocol state is implemented as a class, with methods corresponding to the possible transitions from that state. Fluent APIs encourage a programming style that does not only rely extensively on method chaining, but also requires dynamic checks to ensure the linearity of channel usage. and fits the workflow of MPST implementation neatly.

Our work differs from STP in multiple ways: *(1)* We extend the SCRIBBLE toolchain to support *recursion variables*, allowing refinements on recursions, hence improving expressivity. In this way, developers can specify dependencies across recursive calls, which is not supported in STP. *(2)* We depart from the class-based API generation, and generate a callback-based API. Our approach has the advantage that the linear usage of channels is ensured by construction, saving dynamic checks for channels. *(3)* We use refinement types in F* to verify refinements *statically*. In contrast, STP performs *dynamic* evaluations to validate assertions in protocols. *(4)* The metatheory of session types extended with refinements was not developed in their work.

Several other MPST works follow a similar technique of class-based API generation to overcome limitations of the type system in the target language, e.g. GO [CHJ+19], and C [NCY15]. The class-based API generation techniques suffer from the same limitation: they detect linearity violations at runtime, and offer limited static alternative.

For programming languages with more advances features, researchers utilise these features

[11]For $n = 1$, we run 100,000 iterations of recursion; for $n = 10$, we run 10,000 iterations, etc. Total number of ping pong messages exchanged by the two parties remain the same.

to implement MPST in novel techniques. Imai et al. [INY+20] provide an MPST implementation in OCAML that *statically* checks linearity violation, where global protocols are described using *combinators*. Their combinators can used in conjunction with an OCAML Preprocessor Extension (PPX) to ensure linearity at compilation time. The *affine* type system in RUST is also of particular interest with regards to implementing MPST [CYV22; LNY22], because affinity ensures a channel cannot be *duplicated*. To ensure a channel is actually used, Cutner et al. [CYV22] utilise visibility modifiers to require the user to construct a value `End` that can only be obtained by using channel as prescribed; and Lagaillardie et al. [LNY22] use a compiler annotation `#[must_use]` to require channels to be used.

In a broader picture, the majority of MPST implementation is based on Communicating Finite State Machine (CFSM), including our work, thanks to the connection between local types and CFSMs. Cledou et al. [CEJ+22] summarise the code generation technique based on CFSMs, and propose a new style of API using Set of Pomsets (SOP), utilising *match types* [BBK+22] in SCALA 3. SOP provides a more compact representation in the case of concurrent sub-protocols, and their generated APIs would not suffer from state explosions.

Our work proposes code generation using a callback-styled API. The callback style ensures linear channel usage by construction, by removing the need to expose manual channel management. Although our target language is F*, the callback-styled API code generation technique is applicable to other programming language. This approach is followed by Miu et al. [MFY+21] and Gheri et al. [GLS+22] for implementing MPST in TYPESCRIPT, and by Castro-Perez and Yoshida [CY23] for implementing MPST in GO; and also inspires the work by Viering et al. [VHE+21] for implementing fault-tolerant protocols using an event-driven style in SCALA.

4.8 Summary and Concluding Remarks

In this chapter, we present a toolchain, SESSION*, to apply our theory into practice. Our toolchain allows refined multiparty protocols to be specified using the extended SCRIBBLE language, and implemented using generated refinement-typed F* APIs. We demonstrated with our running example, `HigherLower`, how to implement a refined multiparty protocol with our toolchain SESSION*.

We use the usual CFSM-based technique to generate APIs from a multiparty protocol, but we present two novel changes. On one hand, the generated APIs contain refinement types, so that we can benefit from the advanced F* type system to verify refinements in the protocol statically. On the other hand, we use a callback-based style for generating APIs, through which we achieve the separation of program logic and communication. Moreover, the need to dynamically enforce linearity of communication channels is removed through our generated entry point.

We empirically evaluate our generated code by measuring execution time with experiments. While we impose negligible overhead on the execution time of protocol implementations, we notice possible scalability issues on the compilation time (for type-checking generated code) when a protocol becomes more complex. The scalability issues can be addressed with some workarounds, yet we left the search for an improved API design to address scalability as future work.

5 νSCR: An Extensible Toolchain for Multiparty Protocols

In this chapter, we introduce νSCR (alternatively written as NUSCR), an extensible toolchain for multiparty protocols, and highlight the correspondence to the multiparty session type theory. The name νSCR comes from the SCRIBBLE protocol description language, which we have introduced in § 2.2.3, combined with Greek letter ν, used as the symbol for the channel restriction operator in π-calculus.

νSCR is written in OCAML, implementing the core part of the SCRIBBLE language [YHN+14], with various extensions to the original Multiparty Session Types (MPST). νSCR also provides an interactive web interface (`https://nuscr.dev/`), so that users can perform quick prototyping in browsers, saving the need for installation. We show a screenshot[1] for the web interface in Fig. 5.1.

We introduce our motivation for this new toolchain in § 5.1, and give a brief description of the code layout in § 5.2. To demonstrate the extensibility of νSCR, we use the refinement type extension, the main topic of this thesis, as a case study in § 5.3.

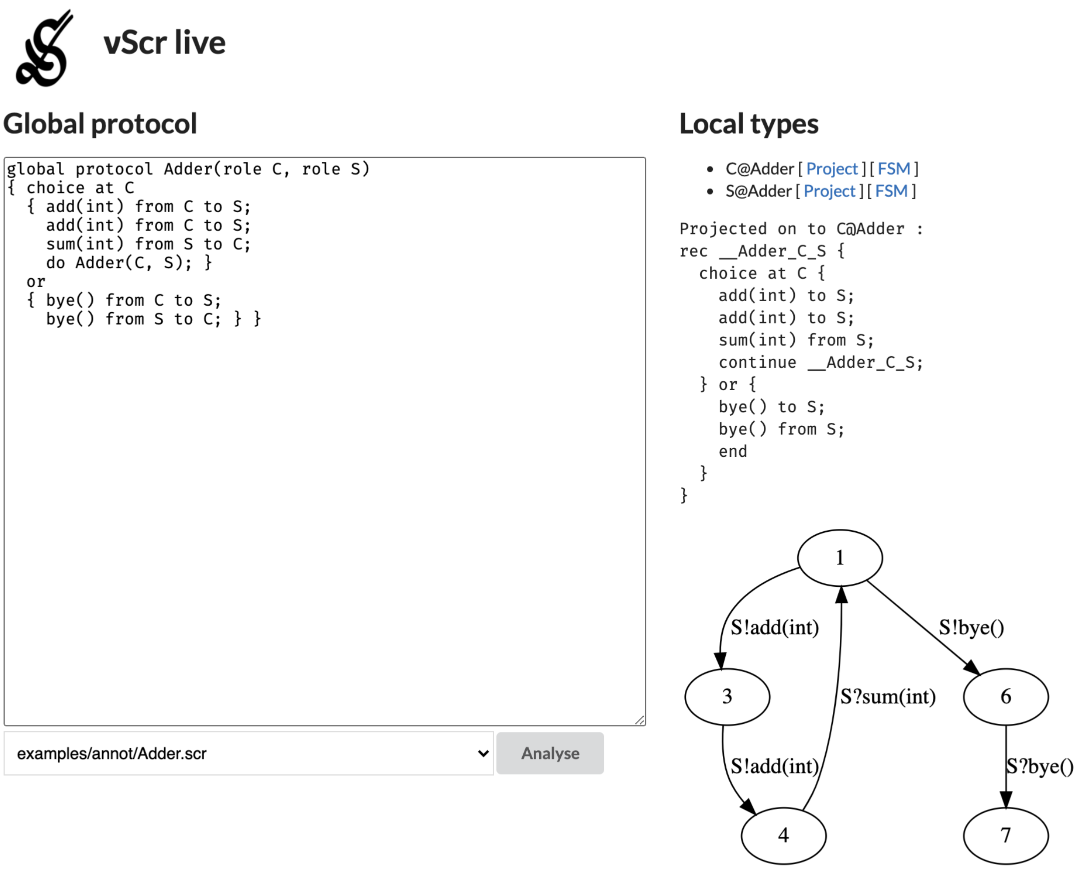
The source code of νSCR is hosted on GitHub at `https://github.com/nuscr/nuscr`. νSCR is open source software under the GNU General Public License (GPL) Version 3.

5.1 Motivation and Overview

The original SCRIBBLE toolchain [YHN+14] has been the traditional choice for multiparty session type practitioners. What we call SCRIBBLE is not only a protocol description language, but also a toolchain for code generation. Moreover, the entire SCRIBBLE project engages closely with the industry, with the goal of making multiparty protocol design and implementation easy to use for all. Over time, the SCRIBBLE toolchain has been used by many researchers, and has resulted in numerous works [NYH13; DHH+15; HY16; HY17; SDH+17; NHY+18; KNY19; CHJ+19; VDG20; ZFH+20; HFD+21; MFY+21; LNY22] (the list is not exhaustive).

The inventors of SCRIBBLE designed the toolchain in a way to promote adoption of multiparty session types for industrial uses. In the meantime, researchers may find it difficult to build their research prototypes, i.e. their extensions of MPST, upon this industrial-grade toolchain. There exists a need to provide a toolchain with MPST practitioners as target users, so that they build their prototype MPST implementations easily. We take the modern core of MPST theory and the core of SCRIBBLE language, to create a brand new toolchain νSCR. νSCR is designed

[1]The logo for νSCR, as shown on the top left corner of the screenshot, is designed by Francisco Ferreira.



```
global protocol Adder(role C, role S)
{ choice at C
  { add(int) from C to S;
    add(int) from C to S;
    sum(int) from S to C;
    do Adder(C, S); }
  or
  { bye() from C to S;
    bye() from S to C; } }
```

- C@Adder [Project] [FSM]
- S@Adder [Project] [FSM]

```
Projected on to C@Adder :
rec __Adder_C_S {
  choice at C {
    add(int) to S;
    add(int) to S;
    sum(int) from S;
    continue __Adder_C_S;
  } or {
    bye() to S;
    bye() from S;
    end
  }
}
```

Figure 5.1: A Screenshot of the νSCR Web Interface, Showing an `Adder` Protocol

to be extensible, so that researchers working on MPST theories can find it easy to implement their extensions upon the code base of νSCR.

To control different language extensions, we take inspiration from the HASKELL programming language and its compiler Glasgow HASKELL Compiler (GHC). A *language pragma*[2] appears at the beginning of an input file, and controls the set of enabled language features. We use a similar design in νSCR to allow users to selectively enable different features in their multiparty protocols. This design has a practical benefit that various extensions are implemented in the same νSCR toolchain, which saves a user the need to download different versions of the toolchain for different extensions. More broadly speaking, we invite researchers to think νSCR as a 'kitchen sink' or a playground for MPST, and encourage researchers to implement their prototypes and contribute new extensions to νSCR.

Multiparty protocols in the core SCRIBBLE description language are accepted by νSCR, and then converted into an MPST global type. From a global type, νSCR is able to project upon a specified participant to obtain their local type, and subsequently obtain the corresponding

[2]`https://wiki.haskell.org/Language_Pragmas`

Communicating Finite State Machine (CFSM). Moreover, νSCR is able to generate code for implementing the participant in various programming languages, from their local type or CFSM. We bear modularity in mind in the design of νSCR, so that each stage in the aforementioned workflow can be customisable. Moreover, νSCR can be used either as a standalone command line application, or as an OCAML library for manipulating multiparty protocols.

Currently, νSCR contains two major language extensions: nested protocols[3] [DH12], implemented by Echarren Serrano [Ech20] (with a code generation back-end for GO), and refinement types (with a code generation back-end for F*), which we use as a case study later in § 5.3.

5.2 Code Layout

The code base of νSCR can be briefly split into 4 major components: parsing and the concrete syntax tree (`syntaxtree`), multiparty session types (`mpst`), code generation (`codegen`) and utilities (`utils`). We introduce the components in detail.

5.2.1 Utilities (`utils`)

We first introduce the the `utils` component, which contains miscellaneous modules fulfilling various utility functions. While this component mainly plays a supportive role, there are a few concepts that need some introduction, as they would be relevant in other components.

- The `Names` module defines *separated* namespaces (as modules) for all kinds of identifiers that may appear throughout the code base, e.g. payload type names (`PayloadTypeName`), payload label names (`LabelName`), type variable names (`TypeVariableName`).

  Each namespace uses a distinct type, in order to prevent namespace mixing, e.g. using a label name (of type `LabelName.t`) as a type variable (of type `TypeVariableName.t`) would result in a type error. Only explicit conversions across namespaces are allowed (via the function `of_other_name`).

- The `Pragma` module defines language pragmas, and contains other configuration options. Language pragmas appear at the beginning of an input file as a special comment `(*# #*)`, and may control which extensions are enabled when processing the input file. We give more details when we demonstrate how we extend νSCR in § 5.3.

5.2.2 Parsing and the Concrete Syntax Tree (`syntaxtree`)

The first stage of processing the input SCRIBBLE protocol is parsing the input into a concrete syntax tree, which is handled by the `syntaxtree` component. This component contains the

[3]This work has subsequently lead to the development of *dynamically updatable* MPST by Castro-Perez and Yoshida [CY23]. Their implementation is based on νSCR, but has been not merged into the main branch of νSCR.

```
1 global protocol Adder(role C, role S)
2 { choice at C
3   { add(int) from C to S;
4     add(int) from C to S;
5     sum(int) from S to C;
6     do Adder(C, S); }
7   or
8   { bye() from C to S;
9     bye() from S to C; } }
```

(a) Adder Protocol in Scribble

$$G_{\mathsf{Adder}} = \mu \mathbf{t}.\mathbf{C} \to \mathbf{S} \left\{ \begin{array}{l} \mathsf{add}(\mathsf{int}). \\ \quad \mathbf{C} \to \mathbf{S} : \mathsf{add}(\mathsf{int}). \\ \quad \mathbf{S} \to \mathbf{C} : \mathsf{sum}(\mathsf{int}). \\ \quad \mathbf{t}; \\ \mathsf{bye}(). \\ \quad \mathbf{S} \to \mathbf{C} : \mathsf{bye}(). \\ \quad \mathsf{end} \end{array} \right\}$$

(b) Global Type of Adder Protocol

Figure 5.2: Adder Protocol and its Corresponding Global Type

syntax trees of the Scribble protocol description language, the core part of which was shown in Fig. 2.1 (in § 2.2.3); as well as the lexing and parsing rules for the Scribble syntax.

As indicated previously, we support a *core* part of the Scribble language, covering the protocol description aspects. The full Scribble language, as implemented in the Scribble toolchain, covers some declarations for engineering considerations. For example, each input file must contain a *module declaration* (analogous to a package declaration in Java at the beginning of a file); and all payload types must be mapped into a type in the target programming language using a *type declaration*. We make the design choice not to support these constructs, as our focus is to allow easy prototyping for researchers, instead of providing an industrial-grade toolchain.

We implement lexing and parsing using generators. The lexer is implemented using the Sedlex[4] package. The parser is generated using the Menhir[5] package, a popular choice among OCaml programmers. The input Scribble file is parsed into a concrete syntax tree as a Scribble module, and can be processed by later stages.

As a simple running example, we show a simple Scribble protocol describing an `Adder` protocol in Fig. 5.2a, where a `C`lient is able to make various requests to `add` two `int`s, before they decide to finish the protocol with a `bye` message. Notice that the Scribble protocol here uses `do` constructs for recursion (Line 6), which is not included in the core syntax given in Fig. 2.1. However, `do` constructs can be converted to recursive μ-types after expansion[6], which takes place in later stages.

[4]`https://github.com/ocaml-community/sedlex`

[5]`http://cambium.inria.fr/~fpottier/menhir/`

[6]In fact, `do` constructs can be more versatile, the list of roles in the protocol call does not need to be the same as the declared roles: suppose the client and server wish to swap role after each computation, this can be expressed via `do Adder(S, C)`, which expands the `Adder` protocol with roles swapped.

```
global protocol NonDirected (role A, role B, role C)
{ choice at A // A sends to either B or to C in this choice
    { Foo() from A to B; // either send to B
      Bar() from A to C; }
  or { Bar() from A to C; // or send to C
      Foo() from A to B; } }
```

Figure 5.3: Non-directed Choice in SCRIBBLE

5.2.3 Multiparty Session Types (`mpst`)

After an input file is parsed into a concrete syntax tree, the concrete syntax tree can be reduced into an abstract syntax tree for further processing. This is implemented inside the `mpst` component, as well as other operations implementing the MPST theory.

Each global protocol in the concrete syntax tree can be converted into a global type (as an abstract syntax tree, defined in the `Gtype` module). During this process, we perform validation and transformation on the concrete syntax tree, e.g. ensuring there are no unbound names, and converting the `do` constructs into `rec` / `continue` constructs that map to μ-types. For our running example, the `Adder` global protocol can be converted into the global type shown in Fig. 5.2b.

It is important to note that, syntactically correct protocols may fall out of the expressivity of the original MPST theory. For example, the protocol shown in Fig. 5.3 uses a *non-directed* choice. The role `A` makes an internal choice of either sending `Foo` to `B` first, or sending `Bar` to `C` first. These non-directed choices in global types are not supported by the original MPST theory (cf. Def. 2.1). This is a consequence of the original SCRIBBLE language design, and many recent works [CDG19; vGHH21; MMS+21; LSW+23] extend MPST to support non-directed choices.

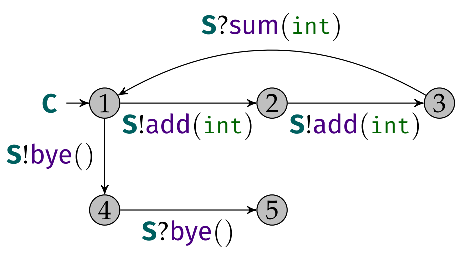
Keeping extensibility in mind, we make the design decision to keep the `choice` constructs *as is* in the abstract syntax of global types. In the meantime, we provide another module named `LiteratureSyntax` for the *literature syntax* of constructs in the original MPST, using the connection between the SCRIBBLE language and the original MPST [NY19]. If a developer prefers the syntax used in the literature, the abstract syntax of global type (as defined in `Gtype` module) can be converted into the literature syntax, and used for further processing.

Subsequently, given a specific role, a global type can be *projected* into a *local type* (defined in `Ltype` module). Additionally, we can construct a corresponding Communicating Finite State Machine (CFSM) (defined in `Efsm` module, which stands for *endpoint finite state machine*) for the local type, using the technique described by Deniélou and Yoshida [DY13]. We use the graph library OCAMLGRAPH[7] [CFS07], to represent the CFSM as a directed graph.

[7] `http://ocamlgraph.lri.fr/index.en.html`

$$L_{\mathsf{C}} = \mu \mathbf{t}.\mathsf{S}\oplus \left\{ \begin{array}{l} \mathsf{add}(\mathsf{int}). \\ \quad \mathsf{S}\oplus\mathsf{add}(\mathsf{int}). \\ \quad \mathsf{S}\&\mathsf{sum}(\mathsf{int}). \\ \quad \mathbf{t}; \\ \mathsf{bye}(). \\ \quad \mathsf{S}\&\mathsf{bye}(). \\ \quad \mathsf{end} \end{array} \right\}$$

(a) Local Type for Role **C**



(b) CFSM for role **C**

Figure 5.4: Local Type and CFSM for Role **C** in the `Adder` Protocol

As a bonus, νSCR supports exporting global and local types in multiple formats, including S-expressions, LaTeX (using macros from `mpstmacros`[8]), and MPSTK[9] [SY19].

For our running example `Adder`, we show the local type for `Client` in Fig. 5.4a, and its corresponding CFSM in Fig. 5.4b. Both local types and CFSMs can be used for code generation purposes, which will be introduced in the next.

5.2.4 Code Generation (`codegen`)

As the name suggests, the `codegen` component concerns generating code for implementing multiparty protocols using the MPST theory. Following the usual top-down design methodology, code generation follows from the projected local type (or their equivalent CFSMs), and ensures that implementations using the generated code will be correct by construction.

Currently, νSCR supports code generation in 3 different target programming languages: OCAML, GO (with the nested protocol extension [Ech20]), and F* (with the refinement type extension, to be explained later in § 5.3). Moreover, νSCR can export the CFSM in the GRAPHVIZ DOT format[10], and other tools may use the exported CFSM to perform their own code generation, independent of νSCR. νSCR has been used in this way to support code generation in RUST [CY21; CYV22], TYPESCRIPT [MFY+21; GLS+22]. In the subsequent text, we briefly describe the code generation technique for OCAML. The overall approach is similar to what we describe in § 4.3, saving the need for considering any refinements.

To generate code in OCAML, we use a CFSM-based generation technique, as proposed by Hu and Yoshida [HY16]; however, we do not use class-based APIs, i.e. states in the CFSMs are classes, and state transitions are methods on classes. While it would be possible to implement a similar object oriented approach in OCAML, it does not fit well in the functional programming paradigm. νSCR uses a *callback-based* approach (as described in § 4.3) for API generation. We generate functions for CFSM transitions, and maintain the state *internally* in a CFSM runner.

[8]`https://github.com/fangyi-zhou/mpstmacros`
[9]`https://github.com/alcestes/mpstk`
[10]`https://graphviz.org/doc/info/lang.html`

```
module type Callbacks = sig
  type t (* An abstract type for user-maintained state *)

  (* Sending callbacks return the state and a labelled value to send *)
  val state1Send : t → t * [‘bye of unit | ‘add of int]
  val state2Send : t → t * [‘add of int]

  (* Receiving callbacks take received value as arguments,
   * and return the state *)
  val state3Receivesum : t → int → t
  val state4Receivebye : t → unit → t
end
```

Figure 5.5: Generated Module Type in OCAML for role `C`

Callback API Style for OCAML

The generated API separates the program logic and communication aspects of the endpoint program. We generate type signatures of *callback functions*, corresponding to state transitions in the CFSM, for handling the program logic. The signatures are collected in the form of a *module type*, named `Callbacks`. We show the generated module signature in Fig. 5.5, for the `Client` role of the `Adder` protocol (Fig. 5.4b). Since we use a graph representation for CFSMs, the generation process is done by iterating through the edges of the graph (i.e. the transitions of the CFSM).

For a complete endpoint implementation, we generate an OCAML *functor*[11] taking a module of type `Callbacks` to return an implementation module. The module exposes a runner, which executes the CFSM when provided connections to other communicating roles. The runner handles the communication with other roles, so that the callback module does not need to involve any sending and receiving primitives.

Optional Monadic APIs

We optionally generate monadic OCAML function types for communication primitives and callback functions. This monadic API allows users to implement the endpoint program with popular asynchronous execution libraries in OCAML that provide monadic interfaces, such as LWT[12] or ASYNC[13], for better performance.

[11] A *functor* in OCAML is a module that is parameterised by another module.
[12] `https://ocsigen.org/lwt/latest/manual/manual`
[13] `https://github.com/janestreet/async`

```
(*# RefinementTypes #*)

global protocol HigherLower(role A, role B, role C)
{ start(n:int{0 <= n && n < 100}) from A to B;
  limit(t0:int{0 < t0}) from A to B;
  rec Loop [t<B>: (t: int{0 < t}) = t0]
  { guess(x:int{0 <= x && x < 100}) from C to B;
    choice at B
        { higher(ignore:unit{n > x && t > 1}) from B to C;
          higher() from B to A;
          continue Loop [t - 1]; }
    or  { win(ignore:unit{n = x}) from B to C;
          lose() from B to A; }
    or  { lower(ignore:unit{n < x && t > 1}) from B to C;
          lower() from B to A;
          continue Loop [t - 1]; }
    or  { lose(ignore:unit{n < >x && t = 1}) from B to C;
          win() from B to A; } } }
```

Figure 5.6: `HigherLower` Protocol in νSCR with refinement type extension

5.3 Extending νSCR

We design νSCR in a modular way to allow extensions of the MPST theory to be implemented easily. The language pragmas, implemented as a special comment at the beginning of an input file, control which extensions are enabled when processing the protocols. So far, two major extensions have been added to νSCR: nested protocols (implemented by Echarren Serrano [Ech20]), and refinement types.

We use the refinement type extension as a case study, to demonstrate how an extension can be implemented in νSCR. The refined MPST theory has been introduced in § 3 of the thesis, and we have presented a toolchain, SESSION⋆, to implement the theory in § 4. While SESSION⋆ uses the SCRIBBLE toolchain for parsing the input protocol, projecting a global protocol to a local type, and generate the CFSM representation of the local type, the code generation back-end is implemented separately from the SCRIBBLE toolchain in OCAML. Thus, we implement the earlier stages for the refinement type extension as an interesting exercise, and port the existing code generator to νSCR. Again, we use the `HigherLower` game (cf. § 4.1.1) as our example, as shown in Fig. 5.6.

5.3.1 Using Pragmas to Control the Extension

Despite describing the same protocol, the syntax in Fig. 5.6 looks quite different from Fig. 4.2. Noticeably, the first line of Fig. 5.6 enables the refinement type extension using the *language pragma* `RefinementTypes` (wrapped in a special comment[14] `(*# ... #*)`).

Pragmas appear on the first line of the input file, and are parsed before the rest of the file. Pragmas are handled by the `Pragma` module in the `utils` component, and can toggle different extensions. Each extension is tracked by a flag in the global configuration, and can be toggled and queried using appropriate setter and getter functions. To add a new pragma, we create a constructor for the new pragma, and a flag for the corresponding extension in the global configuration, and provide the getter and setter functions for the extension. An implementer may wish to check for conflicting pragmas/extensions when processing all pragmas at the beginning, so that incompatible extensions are not enabled at the same time. For our case, we add the `RefinementTypes` pragma, and a corresponding function `refinement_types_enabled` for querying whether the extension is enabled.

In order to preserve the program behaviour when the extension is not enabled, it is essential that subsequent implementations of the extension should query whether the extension is enabled before proceeding.

5.3.2 Extending the Syntax Tree and Parser

The `HigherLower` protocol in Fig. 5.6 uses a number of syntactic constructs added by the refinement type extension. The biggest extensions are the refinement types themselves, appearing in the position of payload types; in contrast, refinements are attached as annotations in Fig. 4.2. Moreover, the recursion variable declarations (Line 6) and updates look closer (Line 11) to the RMPST syntax in Def. 3.1.

To implement these syntactic changes, we extend the `syntaxtree` component. We need to update the concrete syntax tree (in the `Syntax` module) and add new lexing and parsing rules (in the `Lexer` module and the `Parser` module respectively).

Recall that our parser is generated using the MENHIR library. Generated parsers are easier to maintain in general, as a developer only needs to maintain the grammar and leaves the rest to the generator. Compared with a hand-crafted parser, our generated parser does not have a flexibility that a specific grammar rule cannot be enabled or disabled on the fly.

In an ideal situation, we would want the new parsing rules introduced by an extension to be enabled only when the corresponding pragma is set. For example, without the `RefinementTypes` pragma, the protocol in Fig. 5.6 would result in a parse error. This is unfortunately difficult to achieve with parser generators, yet it could be done with hand-crafted parsers, which would in turn be more difficult to maintain.

[14]As a νSCR syntax extension, we support ML style comments: `(* ... *)`.

As a consequence, we make the design decision to use a single parser regardless of the pragmas enabled. Instead, we check whether an extension is enabled during the conversion from the concrete syntax tree to the abstract syntax. Using the same example, without the `RefinementTypes` pragma, the protocol in Fig. 5.6 will still be parsed, but raises an error that new constructs relating to refinement types can only be used with the pragma set.

5.3.3 Implementing the MPST Theory Extension

The crucial part of the extension is to implement the extended theory in the `mpst` component, where global and local types are defined. Within the component, global (resp. local) types are defined using the OCAML type `Gtype.t` (resp. `Ltype.t`). In addition, extending the global and local types does not complete the picture, an implementer needs to connect the concrete syntax obtained from parsing to the abstract syntax of global types.

For the refinement type extensions, one of the major changes is to incorporate refinement types and payload variables in the global and local types. There is no new constructor to be added for global types, but we need to modify constructors concerning recursive types, in order to add recursion variable declarations and updates. For local types, we need to perform similar modifications for recursion constructs, and add a new constructor `SilentL` for silent prefixes.

For converting the abstract syntax from the concrete syntax tree, we modify the extraction function defined at `Gtype.of_protocol`. When processing the new syntactic constructs added by our extension, we must remember to call `Pragma.refinement_types_enabled` (which will return `true` when the pragma is set) to avoid adding unwanted behaviour when the extension is not enabled.

We do not dive into details about how to implement the actual theory extensions. The major changes usually take place in the implementation of projection, which is defined at `Ltype.project`.

5.3.4 Extending the Code Generation

Lastly, we extend the code generation in the in the `codegen` component. In general, a code generation back-end is free to choose any representation in the `mpst` component, so an implementer may pick whichever representation that suits best their code generation approach. As described in § 4.3, we opt for a traditional CFSM-based approach for code generation.

When generating code for refined protocols, we require additional information to be present (for handling recursion variables and their updates). To do so, we need to enrich the CFSM representation in the `Efsm` module, and update the conversion process from local types to CFSMs, which is defined at `Efsm.of_local_type`.

We have previously introduced the API generation process in § 4.3, so we will not repeat here. As mentioned previously, the code generator part of SESSION* is written in OCAML, and takes the CFSM output from the extended SCRIBBLE as the input for the code generator. The

```
(*# RefinementTypes, ValidateRefinementSatisfiability, ValidateRefinementProgress #*)

global protocol Problematic(role A, role B)
{ Num(x: int{x >= 0}) from A to B;
  choice at B // This choice has a progress issue when x = 0
     { Pos(ignore: unit{x > 0}) from B to A; }
  or { // this branch has unsatisfiable conditions
        Neg(ignore: unit{x < 0}) from B to A; } }
```

Figure 5.7: A Problematic Protocol with Satisfiability and Progress Issues

code generator is now ported to νSCR, taking the CFSM produced by νSCR directly, without the intermediate step to export the CFSM in a DOT graph format.

5.3.5 Implementing Selective Checks on Protocols

Up to this point, we have described how the refinement type extension is implemented in νSCR, which is enabled via the `RefinementTypes` pragma. On top of that extension, we use some additional pragmas for enabling selective checks on refined protocols, taking inspiration from the Session Type Provider (STP) [NHY+18].

When defining semantics of refined global and local types, we do not consider whether the refinement type in the message payload is empty (see Rem. 3.35). For practical implementations, it would be beneficial to identify protocols that cannot be implemented (i.e. empty types) at an early stage. We implement a *satisfiability* check for refined multiparty protocols, as is done in STP [NHY+18, § 3]). This check can be enabled via the `ValidateRefinementSatisfiability` pragma, and validates that all global typing contexts are satisfiable, using an Satisfiability Modulo Theories (SMT) solver. We show a problematic protocol in Fig. 5.7: when given an initial *non-negative* integer `x` in a `Num` message, the `Neg` branch has unsatisfiable conditions and cannot be selected.

As explained in Rem. 3.13, lacking variable knowledge can also lead to empty types; however, such empty types cannot be detected via a satisfiability check. Therefore, the satisfiability check can only act as a *best effort* check to exclude unimplementable protocols.

In addition to satisfiability checks, we also implement a *progress* check to ensure a choice is not stuck, which is enabled via the `ValidateRefinementProgress` pragma. This check relates to implementability issues described in Rem. 3.15, and is also implemented by STP [NHY+18, § 3]. For the problematic protocol shown in Fig. 5.7, we notice that the `choice` for `B` gets stuck when the number `x` is zero, since neither branch has satisfiable conditions.

In § 4, we have described a code generation technique targeting F* for *static* verification of the refinements. Instead of static verification, refinements can be evaluated *dynamically*, as is done by STP for the F# language [NHY+18, § 5.2]. To ensure a refinement is *evaluable*, the role that needs to evaluate the refinement expression must know all the variables (cf. history sensitivity [BHT+10], discussed on Page 71). We implement optional checks to ensure the sending (resp. receiving) side of each message is able to evaluate the refinements, which is enabled via the `SenderValidateRefinements` (resp. `ReceiverValidateRefinements`) pragma.

5.4 Related Work

We summarise tools for analysing or implementing multiparty session types. As we have mentioned in § 5.1, the SCRIBBLE toolchain has been highly influential in the tooling area. We begin with some tools that are not based on the SCRIBBLE toolchain, and continue with different extensions of the SCRIBBLE toolchain.

Scalas and Yoshida [SY19] introduce a *generalised* multiparty session type theory, where the type system is parameterised on a safety property. As an accompanying artifact, they implement a toolkit, named MPSTK, for analysing (synchronous) multiparty protocols. The toolkit uses a model checker (MCRL2 [BGK+19]) to decide whether the desired safety property holds. This toolkit is later extended to include a model of crash-stop failures by Barwell et al. [BSY+22].

Imai et al. [INY+20] implement multiparty session types in OCAML with protocol *combinators*, as we have previously discussed in § 4.7. Their tool uses language features such as variant and object types in OCAML to encode external and internal choices in the local types, and supports session delegation.

Majumdar et al. [MMS+21] introduce a new *generalised* projection for MPST with non-directed choices. Their improved projection involves a causality analysis of messages, and expands the expressivity of MPST. The study of the projection is continued by Li et al. [LSW+23], where an automata-based approach is proposed. Both works provide a prototype implementation of the new projection algorithms.

Jongmans and Ferreira [JF23] propose a theory of *synthetic* MPST, where they introduce a flexible *regular* syntax of global and local types, and propose *synthetic* typing rules based on operational semantics. The accompanying tool, named OVEN, implements their synthetic MPST theory, as well as a code generation back-end targeting SCALA.

As previously introduced, the SCRIBBLE toolchain provides a language-agnostic description language for multiparty protocols, and, through various extensions, generates code targeting a variety of programming languages: GO [CHJ+19], JAVA [HY16], PURESCRIPT [KNY19], RUST [LNY22], SCALA [SDH+17], etc. In addition, the SCRIBBLE toolchain incorporates a number of MPST extensions, e.g. explicit connections [HY17], and interruptible protocols [DHH+15]. Moreover, the SCRIBBLE toolchain uses other validation techniques (e.g. checking multiparty compatibility [DY13]) to verify the safety of the multiparty protocol for extra expressivity.

Based on the SCRIBBLE toolchain, Voinea et al. [VDG20] provide a tool, named STMUNGO, to translate a SCRIBBLE multiparty protocol to a type-state specification in JAVA. The type-state specification can be checked via MUNGO, a static type-checker for type-states in JAVA. Developers can use the generated type-state APIs to implement the multiparty protocol safely. Harvey et al. [HFD+21] use the SCRIBBLE toolchain with explicit connections [HY17] to develop a tool, named ENSEMBLES, to implement multiparty session types within an actor language with adaptations.

5.5 Summary and Concluding Remarks

In this chapter, we have presented a new toolchain, named νSCR, designed for multiparty session type researchers. The toolchain is designed with extensibility in mind, and uses language pragmas for controlling different language extensions. We use the refinement type extension as a case study, to walk through the process of implementing a new extension.

We invite researcher to use νSCR as a basis to implement their MPST theory extensions, and we already see some works doing so [GLS+22; CY23]. In the future, we would like to integrate more development of MPST into νSCR, e.g. explicit connections [HY17], automata-based projection [LSW+23], as well as adding more code generation back-ends.

6 Conclusion

We conclude this thesis by summarising the contributions, and proposing possible directions for future work.

6.1 Summary of Contributions

The title of this thesis is 'Refining Multiparty Session Types'. As explained in the introduction (§ 1), the overall goal is to improve the expressivity of multiparty session types, and to allow developers to describe and implement multiparty protocols in finer details. In particular, we aimed to support specifying constraints on payload data in protocols, and verifying these properties in the protocol implementations.

To achieve this goal, we integrate multiparty session types with refinement types. On the theoretical side, we propose a theory of *refined* multiparty session types (§ 3), allowing value-level properties to be specified using refinement types; on the practical side, we propose a toolchain, SESSION*, for implementing refined multiparty protocols in F* (§ 4), enabling the verification of these properties in protocol implementations with the help of the F* compiler.

The theory of *refined* multiparty session (§ 3) incorporates refinement types in the type of message payloads. Using refinement types, data and control flow constraints can be expressed in a global type, including constraints across different messages (e.g. the number payload in the second message is greater than the first one). We extend the semantics of global and local types to include typing contexts that model knowledge of variables available to each participant. We show the relation between the semantics of the global type and projected local types, which is important in the framework of top-down multiparty protocol design.

The toolchain, SESSION*, addresses the practical side of protocol implementations. We choose F* as the target language, in order to utilise its refinement type system for validating protocol implementations. We use a code generation approach, using a callback-styled Application Programming Interface (API), to generate F* code for a given *refined* multiparty protocol. Developers can use the generated APIs to implement a participant by providing callback functions handling relevant program logic. Protocols implemented using our toolchain can benefit from the safety guarantees of the multiparty session type theory, i.e. correctness-by-construction.

In addition to the main goal, we present an extensible toolchain for multiparty protocols, named νSCR, in § 5. The ambition for νSCR is to enable researchers of multiparty session types to implement their extensions easily. We take inspiration from other programming languages, and use language pragmas in νSCR to control different extensions. As a case study, we show how to implement the refinement type extension in the toolchain.

6.2 Future Work

In addition to proposals of future work at the end of each chapter, we propose some additional streams of further investigation.

6.2.1 Subtyping over Refined Local Types

In the background chapter, we introduced a subtyping relation over local types in § 2.4.3, yet we have not introduced a similar relation over *refined* local types. By studying the subtyping relation more carefully, we may be able to strengthen the results obtained from associating global types and configurations, and thus allow more expressivity in our refined global types. We briefly discuss some technical challenges involved in defining such a relation.

Firstly, we can incorporate the subtyping relation over refinement types into the subtyping of local types. However, a natural lifting of the subtyping relation over refinement types (as shown in [RKJ08, Fig. 3]) onto the local types requires a typing context, in order to capture type information about variables. As a consequence, the subtyping relation over refined local types would also need a context.

Secondly, branches with *uninhabited* payload types may sometimes be ignored safely. If a refinement type T is uninhabited under a given typing context, and that type T is used as a payload in a sending prefix, then that branch can be omitted. For example, under the empty typing context, a local type $\mathbf{A}\oplus\left\{\begin{array}{l}\mathsf{Something}(x:\mathtt{int}).\mathtt{end}\\ \mathsf{Nothing}(x:\mathtt{int}\{\mathtt{false}\}).\mathtt{end}\end{array}\right\}$ can be considered a subtype of $\mathbf{A}\oplus\mathsf{Something}(x:\mathtt{int}).\mathtt{end}$, since the $\mathsf{Nothing}$ branch contains an uninhabited payload type. This idea is captured by a subtyping relation in a call-by-push-value system in [LDD+22, Fig. 3, Rule ⊕Sub]. While their work does not involve session types directly, there is sufficient connection between variant record types and internal choices (both represented as ⊕), and between record types and external choices (both represented as &).

6.2.2 Interaction between Refined and Non-Refined Participants

In § 3, we presented an implementation of refined multiparty session types in F⋆, a programming language designed for verification. Researchers have implemented refinement type extensions to general-purpose programming languages, e.g. HASKELL [VSJ+14], TYPESCRIPT [VCJ16], RUBY [KVB+18], RUST [LGV+23]. While we can utilise those extensions to integrate with existing MPST implementations as a future work itself, we achieve better usability if we allow a mixture of implementations: how 'refined' participants (i.e. those implemented using a type system supporting refinement types) can interact with 'non-refined' participants (i.e. those implemented otherwise), without loss of correctness guarantees.

A simple starting point would be to remove the generated `assume` expressions (see § 4.5.2) from refined participants whenever they receive from non-refined participants. The original assumptions are justified because the refinements would be validated by sender, so there would be no need to validate them again. When receiving non-refined participants, the assumptions are no longer justified, and the (refined) receiver of the message must perform validations.

For a more theoretically-based approach, we can take inspirations from gradual typing, especially from gradual refinement types [LT17]. A recent piece of work on this topic proposes dynamic assertion generation [HKS23], where dynamic validations can be generated automatically whenever a best-effort type system cannot validate some refinements.

6.2.3 Endpoint Calculi and Type Systems

In § 3, we have given the syntax and semantics of global and local session types, without introducing any endpoint calculi and/or type systems. The boundary between local types and processes are often blurred, and several recent works [vGHH21; DGD23] describe only one instead of establishing a connection between the two. A simplistic single-session endpoint calculus, can be derived in a similar fashion as demonstrated in works such as [GJP+19; YG20]. However, given that our local typing contexts incorporates multiplicities (cf. § 3.3), the typing system for endpoint calculus needs to support them, on top of supporting refinement types.

A more complex endpoint calculus, e.g. with support for multiple sessions (à la Scalas and Yoshida [SY19]), would require advanced handling of multiplicities, e.g. using *quantitative type theory* [Atk18]: In [SY19], the endpoint calculus does not include data values (i.e. only channel endpoints are communicated), the typing system is thus completely linear. To incorporate a refinement layer in the endpoint calculus, it would be less meaningful if no data values are modelled, and thus the type system would require the handling of linear multiplicity (for the channels), as well as irrelevant (0) and unrestricted (ω) multiplicities for the data values.

6.2.4 Incorporating Failure Models in Semantics

We have used the standard communication model in the semantics of global and local types (§ 3.5), which assumes perfect environment where processes and communication links never fail. In contrast, a key feature of distributed programming is fault tolerance, requiring a system to continue to operate in spite of some failures.

The global and local type semantics can be extended to incorporate various failure models [CGR11, §§ 2.2 and 2.7], in order to capture different failures in real-world distributed systems. In particular, we have explored crash-stop failures in the context of (basic) multiparty session types, in publications not included in this thesis [BSY+22; BHY+23], which can be incorporated to refined MPST easily. Incorporating failure models in semantics can further extend the applicability of the refined multiparty session types into practical distributed programming.

Bibliography

[ABB+16] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos and Nobuko Yoshida. 'Behavioral Types in Programming Languages'. In: *Found. Trends Program. Lang.* 3.2–3 (July 2016), pp. 95–230. ISSN: 2325-1107. DOI: 10.1561/2500000031.

[AS12] Andreas Abel and Gabriel Scherer. 'On Irrelevance and Algorithmic Equality in Predicative Type Theory'. In: *Logical Methods in Computer Science* Volume 8, Issue 1 (Mar. 2012). DOI: 10.2168/LMCS-8(1:29)2012. URL: https://lmcs.episciences.org/1045.

[Atk18] Robert Atkey. 'Syntax and Semantics of Quantitative Type Theory'. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 56–65. DOI: 10.1145/3209108.3209189.

[BBF+11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon and Sergio Maffeis. 'Refinement types for secure implementations'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.2 (2011), 8:1–8:45. DOI: 10.1145/1890028.1890031.

[BBK+22] Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer and Martin Odersky. 'Type-level programming with match types'. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–24. DOI: 10.1145/3498698.

[BCD+09] Karthikeyan Bhargavan, Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet and James J. Leifer. 'Cryptographic Protocol Synthesis and Verification for Multiparty Sessions'. In: *2009 22nd IEEE Computer Security Foundations Symposium*. July 2009, pp. 124–140. DOI: 10.1109/CSF.2009.26.

[BCE+15] Michele Bugliesi, Stefano Calzavara, Fabienne Eigner and Matteo Maffei. 'Affine Refinement Types for Secure Distributed Programming'. In: *ACM Trans. Program. Lang. Syst.* 37.4 (2015), 11:1–11:66. DOI: 10.1145/2743018.

[BDY13] Laura Bocchi, Romain Demangeon and Nobuko Yoshida. 'A Multiparty Multi-session Logic'. In: *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers*. Ed. by Catuscia Palamidessi and Mark Dermot Ryan. Vol. 8191. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 97–111. ISBN: 978-3-642-41157-1. DOI: 10.1007/978-3-642-41157-1_7.

[BFG10] Karthikeyan Bhargavan, Cédric Fournet and Andrew D. Gordon. 'Modular Verification of Security Protocol Code by Typing'. In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 445–456. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706350.

[BGH+12] Gavin M Bierman, Andrew D Gordon, Cătălin Hriţcu and David Langworthy. 'Semantic subtyping with an SMT solver'. In: *Journal of Functional Programming* 22.1 (2012), pp. 31–105. DOI: 10.1017/S0956796812000032.

[BGK+19] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs and Tim A. C. Willemse. 'The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 21–39. DOI: 10.1007/978-3-030-17465-1_2.

[BHT+10] Laura Bocchi, Kohei Honda, Emilio Tuosto and Nobuko Yoshida. 'A Theory of Design-by-Contract for Distributed Multiparty Interactions'. In: *CONCUR 2010 - Concurrency Theory*. Ed. by Paul Gastin and François Laroussinie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 162–176. ISBN: 978-3-642-15375-4. DOI: 10.1007/978-3-642-15375-4_12.

[BHY+23] Adam D. Barwell, Ping Hou, Nobuko Yoshida and Fangyi Zhou. 'Designing Asynchronous Multiparty Protocols with Crash-Stop Failures'. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 1:1–1:30. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.1. URL: https://drops.dagstuhl.de/opus/volltexte/2023/18194.

[BLT20] Franco Barbanera, Ivan Lanese and Emilio Tuosto. 'Choreography Automata'. In: *Coordination Models and Languages*. Ed. by Simon Bliudze and Laura Bocchi. Cham: Springer International Publishing, 2020, pp. 86–106. ISBN: 978-3-030-50029-0. DOI: 10.1007/978-3-030-50029-0_6.

[Bro07] Stephen Brookes. 'A semantics for concurrent separation logic'. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 227–270. DOI: 10.1016/j.tcs.2006.12.034.

[BSY+22] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida and Fangyi Zhou. 'Generalised Multiparty Session Types with Crash-Stop Failures'. In: *33rd International Conference on Concurrency Theory (CONCUR 2022)*. Ed. by Bartek Klin, Sławomir Lasota and Anca Muscholl. Vol. 243. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 35:1–35:25. ISBN: 978-3-95977-246-4. DOI: 10.4230/LIPIcs.CONCUR.2022.35.

[BW81] Manfred Broy and Martin Wirsing. 'On the Algebraic Specification of Nondeterministic Programming Languages'. In: *CAAP '81, Trees in Algebra and Programming, 6th Colloquium, Genoa, Italy, March 5-7, 1981, Proceedings*. Ed. by Egidio Astesiano and Corrado Böhm. Vol. 112. Lecture Notes in Computer Science. Springer, 1981, pp. 162–179. DOI: 10.1007/3-540-10828-9_61.

[BZ83] Daniel Brand and Pitro Zafiropulo. 'On Communicating Finite-State Machines'. In: *J. ACM* 30.2 (Apr. 1983), pp. 323–342. ISSN: 0004-5411. DOI: 10.1145/322374.322380.

[CDG19] Ilaria Castellani, Mariangiola Dezani-Ciancaglini and Paola Giannini. 'Reversible sessions with flexible choices'. In: *Acta Informatica* 56.7-8 (2019), pp. 553–583. DOI: `10.1007/s00236-019-00332-y`.

[CEJ+22] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans and José Proença. 'API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3'. In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 27:1–27:28. ISBN: 978-3-95977-225-9. DOI: `10.4230/LIPIcs.ECOOP.2022.27`. URL: `https://drops.dagstuhl.de/opus/volltexte/2022/16255`.

[CFG+21] 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 the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 237–251. ISBN: 9781450383912. DOI: `10.1145/3453483.3454041`.

[CFS07] Sylvain Conchon, Jean-Christophe Filliâtre and Julien Signoles. 'Designing a Generic Graph Library Using ML Functors'. In: *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007*. Ed. by Marco T. Morazán. Vol. 8. Trends in Functional Programming. Intellect, 2007, pp. 124–140.

[CGR11] Christian Cachin, Rachid Guerraoui and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. ISBN: 978-3-642-15259-7. DOI: `10.1007/978-3-642-15260-3`.

[CHJ+19] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng and Nobuko Yoshida. 'Distributed Programming Using Role-parametric Session Types in Go: Statically-typed Endpoint APIs for Dynamically-instantiated Communication Structures'. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 29:1–29:30. ISSN: 2475-1421. DOI: `10.1145/3290342`.

[CM13] Marco Carbone and Fabrizio Montesi. 'Deadlock-freedom-by-design: multiparty asynchronous global programming'. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 263–274. DOI: `10.1145/2429069.2429101`.

[CY21] Zak Cutner and Nobuko Yoshida. 'Safe Session-Based Asynchronous Coordination in Rust'. In: *Coordination Models and Languages*. Ed. by Ferruccio Damiani and Ornela Dardha. Cham: Springer International Publishing, 2021, pp. 80–89. ISBN: 978-3-030-78142-2. DOI: `10.1007/978-3-030-78142-2_5`.

[CY23] David Castro-Perez and Nobuko Yoshida. 'Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols'. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 6:1–6:30. ISBN: 978-3-95977-281-5.

DOI: 10.4230/LIPIcs.ECOOP.2023.6. URL: https://drops.dagstuhl.de/opus/volltexte/2023/18199.

[CYV22] Zak Cutner, Nobuko Yoshida and Martin Vassor. 'Deadlock-free asynchronous message reordering in rust with multiparty session types'. In: *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*. Ed. by Jaejin Lee, Kunal Agrawal and Michael F. Spear. ACM, 2022, pp. 246–261. DOI: 10.1145/3503221.3508404.

[Dag19] Francesco Dagnino. 'Coaxioms: flexible coinductive definitions by inference systems'. In: *Log. Methods Comput. Sci.* 15.1 (2019). DOI: 10.23638/LMCS-15(1:26)2019.

[DB08] Leonardo De Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24.

[DdY07] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro and Nobuko Yoshida. 'On Progress for Structured Communications'. In: *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*. Ed. by Gilles Barthe and Cédric Fournet. Vol. 4912. Lecture Notes in Computer Science. Springer, 2007, pp. 257–275. DOI: 10.1007/978-3-540-78663-4_18.

[DGD23] Francesco Dagnino, Paola Giannini and Mariangiola Dezani-Ciancaglini. 'Deconfined Global Types for Asynchronous Sessions'. In: *Logical Methods in Computer Science* Volume 19, Issue 1 (Jan. 2023). DOI: 10.46298/lmcs-19(1:3)2023. URL: https://lmcs.episciences.org/10809.

[DH12] Romain Demangeon and Kohei Honda. 'Nested Protocols in Session Types'. In: *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*. Ed. by Maciej Koutny and Irek Ulidowski. Vol. 7454. Lecture Notes in Computer Science. Springer, 2012, pp. 272–286. DOI: 10.1007/978-3-642-32940-1_20.

[DHH+15] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova and Nobuko Yoshida. 'Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python'. In: *Formal Methods in System Design* 46.3 (June 2015), pp. 197–225. ISSN: 1572-8102. DOI: 10.1007/s10703-014-0218-8.

[dMBV19] Jan de Muijnck-Hughes, Edwin Brady and Wim Vanderbauwhede. 'Value-Dependent Session Design in a Dependently Typed Language'. In: Proceedings *Programming Language Approaches to Concurrency- and Communication-cEntric Software,* Prague, Czech Republic, 7th April 2019. Ed. by Francisco Martins and Dominic Orchard. Vol. 291. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2019, pp. 47–59. DOI: 10.4204/EPTCS.291.5.

[dMBV20] Jan de Muijnck-Hughes, Edwin Brady and Wim Vanderbauwhede. 'A Framework for Resource Dependent EDSLs in a Dependently Typed Language (Pearl)'. In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 20:1–20:31. ISBN: 978-3-95977-154-2. DOI: `10.4230/LIPIcs.ECOOP.2020.20`. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/13177`.

[DP20] Ankush Das and Frank Pfenning. 'Session Types with Arithmetic Refinements'. In: *31st International Conference on Concurrency Theory (CONCUR 2020)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 13:1–13:18. ISBN: 978-3-95977-160-3. DOI: `10.4230/LIPIcs.CONCUR.2020.13`. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/12825`.

[DP22a] Ornela Dardha and Jorge A. Pérez. 'Comparing type systems for deadlock freedom'. In: *J. Log. Algebraic Methods Program.* 124 (2022), p. 100717. DOI: `10.1016/j.jlamp.2021.100717`.

[DP22b] Ankush Das and Frank Pfenning. 'Rast: A Language for Resource-Aware Session Types'. In: *Log. Methods Comput. Sci.* 18.1 (2022). DOI: `10.46298/LMCS-18(1:9)2022`. URL: `https://lmcs.episciences.org/8954`.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. 'Dynamic Multirole Session Types'. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 435–446. ISBN: 978-1-4503-0490-0. DOI: `10.1145/1926385.1926435`.

[DY13] Pierre-Malo Deniélou and Nobuko Yoshida. 'Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types'. In: *40th International Colloquium on Automata, Languages and Programming*. Vol. 7966. LNCS. Berlin, Heidelberg: Springer, 2013, pp. 174–186. DOI: `10.1007/978-3-642-39212-2_18`.

[Ech20] Benito Echarren Serrano. 'Nested Multiparty Session Programming in Go'. available on `https://becharrens.files.wordpress.com/2020/07/final_report.pdf`. MA thesis. Imperial College London, 2020.

[FP91] Tim Freeman and Frank Pfenning. 'Refinement Types for ML'. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, 1991, pp. 268–277. ISBN: 0-89791-428-7. DOI: `10.1145/113445.113468`.

[FRS+21] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux and Tahina Ramananandro. 'Steel: proof-oriented programming in a dependently typed concurrent separation logic'. In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: `10.1145/3473590`.

[GH05] Simon J. Gay and Malcolm Hole. 'Subtyping for session types in the pi calculus'. In: *Acta Informatica* 42.2-3 (2005), pp. 191–225. DOI: `10.1007/s00236-005-0177-z`.

[GJP+19] Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas and Nobuko Yoshida. 'Precise subtyping for synchronous multiparty sessions'. In: *J. Log. Algebraic Methods Program.* 104 (2019), pp. 127–173. DOI: 10.1016/j.jlamp.2018.12.002.

[GLL+22] Yanjie Gao, Zhengxian Li, Haoxiang Lin, Hongyu Zhang, Ming Wu and Mao Yang. 'REFTY: Refinement Types for Valid Deep Learning Models'. In: *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1843–1855. DOI: 10.1145/3510003.3510077.

[GLS+22] Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto and Nobuko Yoshida. 'Design-By-Contract for Flexible Multiparty Session Protocols'. In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 8:1–8:28. ISBN: 978-3-95977-225-9. DOI: 10.4230/LIPIcs.ECOOP.2022.8. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16236.

[HBK19] Jonas Kastberg Hinrichsen, Jesper Bengtson and Robbert Krebbers. 'Actris: Session-Type Based Reasoning in Separation Logic'. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371074.

[HBK22] Jonas Kastberg Hinrichsen, Jesper Bengtson and Robbert Krebbers. 'Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic'. In: *Log. Methods Comput. Sci.* 18.2 (2022). DOI: 10.46298/lmcs-18(2:16)2022. URL: https://lmcs.episciences.org/9689.

[HFD+21] Paul Harvey, Simon Fowler, Ornela Dardha and Simon J. Gay. 'Multiparty Session Types for Safe Runtime Adaptation in an Actor Language'. In: *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Ed. by Anders Møller and Manu Sridharan. Vol. 194. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 10:1–10:30. ISBN: 978-3-95977-190-0. DOI: 10.4230/LIPIcs.ECOOP.2021.10. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14053.

[HKS23] Momoko Hattori, Naoki Kobayashi and Ryosuke Sato. 'Gradual Tensor Shape Checking'. In: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*. Ed. by Thomas Wies. Vol. 13990. Lecture Notes in Computer Science. Springer, 2023, pp. 197–224. DOI: 10.1007/978-3-031-30044-8_8.

[Hu17] Raymond Hu. 'Distributed Programming Using Java APIs Generated from Session Types'. In: *Behavioural Types: from Theory to Tools* (2017), pp. 287–308.

[HVH19] Martin A. T. Handley, Niki Vazou and Graham Hutton. 'Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell'. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371092.

[HVK98] Kohei Honda, Vasco T. Vasconcelos and Makoto Kubo. 'Language primitives and type discipline for structured communication-based programming'. In: *Programming Languages and Systems*. Ed. by Chris Hankin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. ISBN: 978-3-540-69722-0. DOI: 10.1007/BFb0053567.

[HY16] Raymond Hu and Nobuko Yoshida. 'Hybrid Session Verification through Endpoint API Generation'. In: *19th International Conference on Fundamental Approaches to Software Engineering*. Vol. 9633. LNCS. Berlin, Heidelberg: Springer, 2016, pp. 401–418. DOI: `10.1007/978-3-662-49665-7_24`.

[HY17] Raymond Hu and Nobuko Yoshida. 'Explicit Connection Actions in Multiparty Session Types'. In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Marieke Huisman and Julia Rubin. Vol. 10202. Lecture Notes in Computer Science. Springer, 2017, pp. 116–133. DOI: `10.1007/978-3-662-54494-5_7`.

[HYC08] Kohei Honda, Nobuko Yoshida and Marco Carbone. 'Multiparty Asynchronous Session Types'. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 273–284. ISBN: 978-1-59593-689-9. DOI: `10.1145/1328438.1328472`.

[INY+20] Keigo Imai, Rumyana Neykova, Nobuko Yoshida and Shoji Yuen. 'Multiparty Session Programming With Global Protocol Combinators'. In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 9:1–9:30. DOI: `10.4230/LIPIcs.ECOOP.2020.9`.

[JF23] Sung-Shik Jongmans and Francisco Ferreira. 'Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types (Pearl/Brave New Idea)'. In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 42:1–42:30. DOI: `10.4230/LIPIcs.ECOOP.2023.42`.

[JM16] Peter Jipsen and M. Andrew Moshier. 'Concurrent Kleene algebra with tests and branching automata'. In: *J. Log. Algebraic Methods Program.* 85.4 (2016), pp. 637–652. DOI: `10.1016/j.jlamp.2015.12.005`.

[JV21] Ranjit Jhala and Niki Vazou. 'Refinement Types: A Tutorial'. In: *Found. Trends Program. Lang.* 6.3-4 (2021), pp. 159–317. DOI: `10.1561/2500000032`.

[JvdB22] Sung-Shik Jongmans and Petra van den Bos. 'A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming'. In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 520–547. DOI: `10.1007/978-3-030-99336-8_19`.

[Kel76] Robert M. Keller. 'Formal Verification of Parallel Programs'. In: *Commun. ACM* 19.7 (1976), pp. 371–384. DOI: `10.1145/360248.360251`.

[KKK+20] Wen Kokke, Ekaterina Komendantskaya, Daniel Kienitz, Robert Atkey and David Aspinall. 'Neural Networks, Secure by Construction - An Exploration of Refinement Types'. In: *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*. Ed. by Bruno C. d. S. Oliveira. Vol. 12470. Lecture Notes in Computer Science. Springer, 2020, pp. 67–85. DOI: 10.1007/978-3-030-64437-6_4.

[KNY19] Jonathan King, Nicholas Ng and Nobuko Yoshida. 'Multiparty Session Type-safe Web Development with Static Linearity'. In: *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*. Ed. by Francisco Martins and Dominic Orchard. Vol. 291. EPTCS. 2019, pp. 35–46. DOI: 10.4204/EPTCS.291.4.

[Kob06] Naoki Kobayashi. 'A New Type System for Deadlock-Free Processes'. In: *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*. Ed. by Christel Baier and Holger Hermanns. Vol. 4137. Lecture Notes in Computer Science. Springer, 2006, pp. 233–247. DOI: 10.1007/11817949_16.

[Koz97] Dexter Kozen. 'Kleene Algebra with Tests'. In: *ACM Trans. Program. Lang. Syst.* 19.3 (1997), pp. 427–443. DOI: 10.1145/256167.256195.

[KVB+18] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster and Emina Torlak. 'Refinement Types for Ruby'. In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*. Ed. by Isil Dillig and Jens Palsberg. Vol. 10747. Lecture Notes in Computer Science. Springer, 2018, pp. 269–290. DOI: 10.1007/978-3-319-73721-8_13.

[KWR+20] Tristan Knoth, Di Wang, Adam Reynolds, Jan Hoffmann and Nadia Polikarpova. 'Liquid Resource Types'. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408988.

[KY14] Dimitrios Kouzapas and Nobuko Yoshida. 'Globally Governed Session Semantics'. In: *Log. Methods Comput. Sci.* 10.4 (2014). DOI: 10.2168/LMCS-10(4:20)2014.

[LDD+22] Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido and Frank Pfenning. 'Polarized Subtyping'. In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 431–461. DOI: 10.1007/978-3-030-99336-8_16.

[LGV+23] Nico Lehmann, Adam T. Geller, Niki Vazou and Ranjit Jhala. 'Flux: Liquid Types for Rust'. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591283.

[LKB+21] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan and Ranjit Jhala. 'STORM: Refinement Types for Secure Web Applications'. In: *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. Ed. by Angela Demke Brown and Jay R. Lorch. USENIX Association, 2021, pp. 441–459. URL: https://www.usenix.org/conference/osdi21/presentation/lehmann.

[LNY22] Nicolas Lagaillardie, Rumyana Neykova and Nobuko Yoshida. 'Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types'. In: *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Ed. by Karim Ali and Jan Vitek. Vol. 222. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 4:1–4:29. ISBN: 978-3-95977-225-9. DOI: 10.4230/LIPIcs.ECOOP.2022.4. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16232.

[LSW+23] Elaine Li, Felix Stutz, Thomas Wies and Damien Zufferey. 'Complete Multiparty Session Type Projection with Automata'. In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*. Ed. by Constantin Enea and Akash Lal. Vol. 13966. Lecture Notes in Computer Science. Springer, 2023, pp. 350–373. DOI: 10.1007/978-3-031-37709-9_17.

[LT17] Nico Lehmann and Éric Tanter. 'Gradual Refinement Types'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2017. Paris, France: Association for Computing Machinery, 2017, pp. 775–788. ISBN: 9781450346603. DOI: 10.1145/3009837.3009856.

[MFY+21] Anson Miu, Francisco Ferreira, Nobuko Yoshida and Fangyi Zhou. 'Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types'. In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. CC 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 94–106. ISBN: 9781450383257. DOI: 10.1145/3446804.3446854.

[Mil71] Robin Milner. 'An Algebraic Definition of Simulation Between Programs'. In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*. Ed. by D. C. Cooper. William Kaufmann, 1971, pp. 481–489. URL: http://ijcai.org/Proceedings/71/Papers/044.pdf.

[Mil78] Robin Milner. 'A theory of type polymorphism in programming'. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: http://www.sciencedirect.com/science/article/pii/0022000078900144.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3.

[MMS+21] Rupak Majumdar, Madhavan Mukund, Felix Stutz and Damien Zufferey. 'Generalising Projection in Asynchronous Multiparty Session Types'. In: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*. Ed. by Serge Haddad and Daniele Varacca. Vol. 203. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 35:1–35:24. DOI: 10.4230/LIPIcs.CONCUR.2021.35.

[Nak00] Hiroshi Nakano. 'A Modality for Recursion'. In: *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 2000, pp. 255–266. DOI: 10.1109/LICS.2000.855774.

[NCY15] Nicholas Ng, Jose G.F. Coutinho and Nobuko Yoshida. 'Protocols by Default: Safe MPI Code Generation based on Session Types'. In: *24th International Conference on Compiler Construction*. Vol. 9031. LNCS. Springer, 2015, pp. 212–232. DOI: 10.1007/978-3-662-46663-6_11.

[NHY+18] Rumyana Neykova, Raymond Hu, Nobuko Yoshida and Fahd Abdeljallal. 'A Session Type Provider: Compile-time API Generation of Distributed Protocols with Refinements in F#'. In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: ACM, 2018, pp. 128–138. ISBN: 978-1-4503-5644-2. DOI: 10.1145/3178372.3179495.

[NSC+22] Yuki Nishida, Hiromasa Saito, Ran Chen, Akira Kawata, Jun Furuse, Kohei Suenaga and Atsushi Igarashi. 'Helmholtz: A Verifier for Tezos Smart Contracts Based on Refinement Types'. In: *New Gener. Comput.* 40.2 (2022), pp. 507–540. DOI: 10.1007/s00354-022-00167-1.

[NY19] Rumyana Neykova and Nobuko Yoshida. '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*. Ed. by Michele Boreale, Flavio Corradini, Michele Loreti and Rosario Pugliese. Cham: Springer International Publishing, 2019, pp. 236–259. ISBN: 978-3-030-21485-2. DOI: 10.1007/978-3-030-21485-2_14.

[NYH13] Rumyana Neykova, Nobuko Yoshida and Raymond Hu. 'SPY: Local Verification of Global Protocols'. In: *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*. Ed. by Axel Legay and Saddek Bensalem. Vol. 8174. Lecture Notes in Computer Science. Springer, 2013, pp. 358–363. DOI: 10.1007/978-3-642-40787-1_25.

[Pfe01] Frank Pfenning. 'Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory'. In: *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 2001, pp. 221–230. DOI: 10.1109/LICS.2001.932499.

[PGS16] Tomas Petricek, Gustavo Guerra and Don Syme. 'Types from Data: Making Structured Data First-class Citizens in F#'. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: ACM, 2016, pp. 477–490. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908115.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091.

[PSY+20] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance and Armando Solar-Lezama. 'Liquid Information Flow Control'. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408987.

[PT08] Riccardo Pucella and Jesse A. Tov. 'Haskell Session Types with (Almost) No Class'. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: ACM, 2008, pp. 25–36. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411290.

[PZR+17] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet and Nikhil Swamy. 'Verified low-level programming embedded in F*'. In: *Proc. ACM Program. Lang.* 1.ICFP (2017), 17:1–17:29. DOI: 10.1145/3110261.

[Rey02] John C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[RKJ08] Patrick M. Rondon, Ming Kawaguchi and Ranjit Jhala. 'Liquid Types'. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 159–169. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375602.

[RW94] Arend Rensink and Heike Wehrheim. 'Weak Sequential Composition in Process Algebras'. In: *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*. Ed. by Bengt Jonsson and Joachim Parrow. Vol. 836. Lecture Notes in Computer Science. Springer, 1994, pp. 226–241. DOI: 10.1007/978-3-540-48654-1_20.

[SCF+13] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan and Jean Yang. 'Secure distributed programming with value-dependent types'. In: *J. Funct. Program.* 23.4 (2013), pp. 402–451. DOI: 10.1017/S0956796813000142.

[SDH+17] Alceste Scalas, Ornela Dardha, Raymond Hu and Nobuko Yoshida. 'A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming'. In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Ed. by Peter Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 24:1–24:31. DOI: 10.4230/LIPIcs.ECOOP.2017.24.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue and Santiago Zanella-Béguelin. 'Dependent Types and Multi-monadic Effects in F*'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655.

[SK16] Georg Stefan Schmid and Viktor Kuncak. 'SMT-based Checking of Predicate-qualified Types for Scala'. In: *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*. SCALA 2016. Amsterdam, Netherlands: ACM, 2016, pp. 31–40. ISBN: 978-1-4503-4648-1. DOI: 10.1145/2998392.2998398.

[SRF+20] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman and Guido Martínez. 'SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs'. In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3409003.

[Stu23] Felix Stutz. 'Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts'. In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 32:1–32:31. DOI: 10.4230/LIPIcs.ECOOP.2023.32.

[SY19] Alceste Scalas and Nobuko Yoshida. 'Less is More: Multiparty Session Types Revisited'. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290343.

[TML+22] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig and Yu Feng. 'SolType: refinement types for arithmetic overflow in solidity'. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–29. DOI: 10.1145/3498665.

[TY17] Bernardo Toninho and Nobuko Yoshida. 'Certifying data in multiparty session types'. In: *J. Log. Algebraic Methods Program.* 90 (2017), pp. 61–83. DOI: `10.1016/j.jlamp.2016.11.005`.

[VCE+18] Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu and Lukasz Ziarek. 'A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems'. In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Amal Ahmed. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 799–826. DOI: `10.1007/978-3-319-89884-1_28`.

[VCJ16] Panagiotis Vekris, Benjamin Cosman and Ranjit Jhala. 'Refinement types for TypeScript'. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 2016, pp. 310–325. DOI: `10.1145/2908080.2908110`.

[VDG20] A. Laura Voinea, Ornela Dardha and Simon J. Gay. 'Typechecking Java Protocols with [St]-Mungo'. In: *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*. Ed. by Alexey Gotsman and Ana Sokolova. Vol. 12136. Lecture Notes in Computer Science. Springer, 2020, pp. 208–224. DOI: `10.1007/978-3-030-50086-3_12`.

[vGHH21] Rob van Glabbeek, Peter Höfner and Ross Horne. 'Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom'. In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 2021, pp. 1–13. DOI: `10.1109/LICS52264.2021.9470531`.

[VHE+21] Malte Viering, Raymond Hu, Patrick Eugster and Lukasz Ziarek. 'A multiparty session typing discipline for fault-tolerant event-driven distributed programming'. In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–30. DOI: `10.1145/3485501`.

[VSJ+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis and Simon Peyton-Jones. 'Refinement Types for Haskell'. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 269–282. ISBN: 978-1-4503-2873-9. DOI: `10.1145/2628136.2628161`.

[VTC+17] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler and Ranjit Jhala. 'Refinement Reflection: Complete Verification with SMT'. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 53:1–53:31. ISSN: 2475-1421. DOI: `10.1145/3158141`.

[YDB+10] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri and Raymond Hu. '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.* Ed. by C.-H. Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer, 2010, pp. 128–145. DOI: `10.1007/978-3-642-12032-9_10`.

[YG20] Nobuko Yoshida and Lorenzo Gheri. 'A Very Gentle Introduction to Multiparty Session Types'. In: *Distributed Computing and Internet Technology*. Ed. by Dang Van Hung and Meenakshi D´Souza. Cham: Springer International Publishing, 2020, pp. 73–93. ISBN: 978-3-030-36987-3. DOI: `10.1007/978-3-030-36987-3_5`.

[YHN+14] Nobuko Yoshida, Raymond Hu, Rumyana Neykova and Nicholas Ng. 'The Scribble Protocol Language'. In: *8th International Symposium on Trustworthy Global Computing - Volume 8358*. TGC 2013. Buenos Aires, Argentina: Springer-Verlag, 2014, pp. 22–41. ISBN: 978-3-319-05118-5. DOI: `10.1007/978-3-319-05119-2_3`.

[YZF21] Nobuko Yoshida, Fangyi Zhou and Francisco Ferreira. 'Communicating Finite State Machines and an Extensible Toolchain for Multiparty Session Types'. In: *Fundamentals of Computation Theory*. Ed. by Evripidis Bampis and Aris Pagourtzis. Cham: Springer International Publishing, 2021, pp. 18–35. ISBN: 978-3-030-86593-1. DOI: `10.1007/978-3-030-86593-1_2`.

[ZFH+20] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova and Nobuko Yoshida. 'Statically Verified Refinements for Multiparty Protocols'. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: `10.1145/3428216`.

Acronyms

API Application Programming Interface. 74, 108

CFSM Communicating Finite State Machine. 73, 74, 79, 93, 97, 99

EDSL Embedded Domain Specific Language. 72

GHC Glasgow HASKELL Compiler. 96

GPL GNU General Public License. 95

LTS Labelled Transition System. 16, 47, 56

MPST Multiparty Session Types. 8, 95

PPX Preprocessor Extension. 93

RMPST Refined Multiparty Session Types. 27

SMT Satisfiability Modulo Theories. 4, 29, 37, 70, 105

SOP Set of Pomsets. 93

STP Session Type Provider. 92, 105

TCP Transmission Control Protocol. 84