Multiparty Session Types for Distributed and Failure-Prone Systems

Matthew Alan Le Brun

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Computer Science
College of Science and Engineering
University of Glasgow

Aug 2025

Abstract

Distributed computer systems inherently introduce notions of *concurrency*, *nondeterminism* and possibly *failure-prone communication* into programs. This makes designing, and implementing, correct distributed systems difficult, and it is widely agreed that such systems require rigorous methods of verification.

Types and type systems are lightweight techniques to formal verification whereby code implementations are annotated with meta-information about the data they use. This provides certain guarantees of correctness about the usage of data within a program, and helps developers design and implement code that is correct by construction. Specific to message-passing systems, *session types* not only describe information about *what* data is, but also about *how* it should be communicated. Essentially, they embed communication protocols as types within a program, ensuring that well-typed code abides by the specified protocol and benefits from any properties determined on said protocol, e.g. being *deadlock-free, terminating, live,* etc. *Multiparty session types* (MPST) are a generalisation of session type theory that allows for the description of communication protocols involving two or more named participants with possibly cyclic or interleaved communication patterns.

This thesis adapts MPST to describe protocols for distributed and failure-prone systems. The contributions presented are in the form of three novel languages: MPST!, MAGπ, and MAGπ!, each using a new MPST theory targetting client-server, failure-prone and fault-tolerant systems respectively; where the unit of failure is message loss and fault tolerance is introduced through the client-server paradigm.

This work establishes foundations for designing MPST systems targetted towards building multiparty client-server and fault-tolerant programs. For the first time, MPSTs are explored within programs designed to run over failure-prone networks, where messages are susceptible to loss, unbounded delay and total reordering. The construct of replication is shown to be expressively significant for building multiparty client-server systems. Lastly, it is demonstrated how combining replication, nondeterministic timeouts and standard MPST constructs, results in an expressive type system capable of modelling—and verifying properties on—fault-tolerant multiparty programs.

Contents

Abstract **i**

Acknowledgements **viii**

Declaration **x**

1 Introduction **1**
- 1.1 Research Questions . . . 4
- 1.2 Contributions . . . 5
- 1.3 Thesis Structure . . . 7

2 Preliminaries **8**
- 2.1 Process Calculi . . . 8
  - 2.1.1 Polyadic π-Calculus . . . 8
  - 2.1.2 Multiparty Session Calculus . . . 9
- 2.2 Session Types . . . 10
  - 2.2.1 Binary Session Types . . . 11
  - 2.2.2 Multiparty Session Types . . . 12

3 A Common Core: Generalised MPST **14**
- 3.1 Introduction . . . 14
- 3.2 Calculus . . . 16
  - 3.2.1 Syntax . . . 16
  - 3.2.2 Types . . . 18
  - 3.2.3 Semantics . . . 23
- 3.3 Metatheory . . . 28
  - 3.3.1 Type Context Properties . . . 29
  - 3.3.2 Subject Reduction . . . 30
  - 3.3.3 Session Fidelity . . . 33
  - 3.3.4 Behavioural Properties . . . 36
- 3.4 Conclusion . . . 37

4 Idealism: The Client-Server Paradigm **38**
4.1 Introduction . . . 38
4.2 Multiparty Session Types with a Bang! . . . 40
4.2.1 Language . . . 40
4.2.2 Process Semantics . . . 42
4.2.3 Types . . . 45
4.2.4 Type Semantics . . . 50
4.3 Metatheory . . . 53
4.3.1 Context Properties . . . 53
4.3.2 Role Substitution . . . 56
4.3.3 Subject Reduction . . . 58
4.3.4 Session Fidelity . . . 60
4.3.5 Behavioural Properties . . . 67
4.4 Expressivity and Decidability . . . 68
4.4.1 Expressivity . . . 68
4.4.2 Decidability . . . 74
4.5 Related Work . . . 79
4.6 Conclusion . . . 80

5 Nihilism: Failure-Prone Communication **82**
5.1 Introduction . . . 82
5.2 MAGπ . . . 85
5.2.1 Language . . . 85
5.2.2 Process Semantics . . . 88
5.2.3 Types . . . 89
5.2.4 Type Semantics . . . 95
5.3 Metatheory . . . 97
5.3.1 Subject Reduction . . . 98
5.3.2 Session Fidelity . . . 100
5.3.3 Decidability . . . 101
5.4 Key Observations . . . 103
5.4.1 Subtyping vs Compatibility . . . 103
5.4.2 Failure Models . . . 104
5.5 Network Assumptions . . . 105
5.5.1 User Datagram Protocol . . . 106
5.5.2 Reliable Data Protocol . . . 107
5.5.3 Real-Time Transport Protocol . . . 107
5.5.4 Transmission Control Protocol . . . 107
5.6 Related Work . . . 108

5.7 Conclusion 109

6 Realism: Typing Fault-Tolerant Servers 111
6.1 Introduction 111
6.2 MAGπ! 113
6.2.1 Syntax 113
6.2.2 Semantics 116
6.3 Metatheory 120
6.3.1 Subject Reduction and Session Fidelity 120
6.4 Examples 123
6.4.1 Ping Utility 123
6.4.2 Learning Switch 125
6.5 Conclusion 128

7 Conclusion 129

A Proofs for MPST! 130

B Further Definitions and Proofs for MAGπ and MAGπ! 148

List of Tables

3.1 Properties of a type context Γ. 29

4.1 Properties of a type context Γ, in MPST!. 55

5.1 Failure models as context paths. 106

List of Figures

2.1 Syntax of the finite polyadic π-calculus 9
2.2 Syntax of a multiparty session calculus 10

3.1 Syntax of Core Calculus 16
3.2 Syntax of Core Calculus Types 18
3.3 Type context operations. 20
3.4 Typing rules for the core calculus. 22
3.5 Operational semantics for the core calculus. 24
3.6 Type semantics. 26
3.7 Session Fidelity Restrictions on Core Calculus Syntax 34

4.1 Syntax of MPST! 41
4.2 Operational semantics for MPST!. 42
4.3 Process definitions for load balancer example 43
4.4 Syntax of Types 45
4.5 Typing contexts and context operations. 47
4.6 Typing rules. 48
4.7 Type semantics. 49
4.8 Free role variables and substitution in contexts 52
4.9 Session Fidelity Restrictions on MPST! 61

5.1 Syntax of MAGπ 86
5.2 Operational semantics MAGπ. 88
5.3 Syntax of MAGπ Types 89
5.4 Type context operations. 91
5.5 Typing rules. 93
5.6 Type semantics. 96
5.7 Session Fidelity Restricted Syntax for MAGπ 101

6.1 Syntax of MAGπ! 114
6.2 Syntax of MAGπ! Types 116

6.3 Operational semantics MAGπ!. 117
6.4 Type semantics. 119

B.1 Type context operations. 149
B.2 Typing rules. 150

Acknowledgements

This thesis represents four years of my life for which I am very grateful to have had. The PhD brought many exciting things with it: new country, weather, culture, hobbies; and most fantastic of all, connections with some incredible people. The following is a series of soppy and heartfelt thanks towards said people, attempting to describe the gratitude I feel.

I am privileged to have been supervised by three incredible researchers. Ornela, in our very first official meeting four years ago, we laid out expectations for our student-supervisor relationship. We both said that we wish to always be honest with one another. I want to thank you for making it so easy for me to uphold this promise. You have always been so approachable, caring and understanding. You have respected me as a fellow researcher from the start; you instilled within me a confidence in myself I was lacking; and you demonstrate by example the value of compassion, which I wish to emulate every day of my life. Simon, you have a work ethic I greatly admire and strive to imitate. You deeply care about the work you do and the people you work with, and I am so grateful that I got to be one of said people! Thank you for all the times you popped into my office, for all the lunches, for all the chats, for that one time you booked my leave for me. You made university a very safe, inviting and inspirational place to be. Paul, even though our time together was short, you always showed genuine interest in my research, my professional development and my well-being. Thank you for all the wisdom you have passed on to me, I look forward to spending more time with you. I am proud to call all of you my supervisors, and my friends.

I am sincerely grateful to my examiners, Professors Simon Gay and Alceste Scalas, who were very fair and thorough. We had an invigorating discussion during my viva, and I thoroughly enjoyed the experience. I hope to one day continue the examination train you have started!

Thank you to my office mates, Mathias, Arwa and Olivia, for making work an exciting place to be! It has been a privilege to get to know you and work by your sides. Thanks as well to all members of the FATA research group, past and present, for being such incredible, approachable and funny people; especially my fellow FATA PhD students: Laura, Ella, Frederik, Danielius, Matthew, Fabricio, Elena, Pepe, Nick. You are all just brilliant.

A special thank you goes to two of my best friends. From teaching me Erlang to hiking the Highlands' munros, our friendship has come a long way, my dear Duncan! Long will I cherish the memories of our shenanigans. Jakeyboy, though our friendship is relatively fresh, you have

so quickly become a big part of my life. I look forward to many more years of shared laughs and, of course, tea!

Just as I have established long-lasting new friendships, some older connections will always remain close to my heart, and have only got stronger despite the miles that separate us. Mummy and Daddy, my thanks to you stretches back 26 years! Not only have you supported me and encouraged me during the PhD, but you have always done so, and never have you once swayed me away from chasing my dreams and ambitions. Your Sproggy loves you! I am also very grateful of my incredibly supportive family, and extended family, who have cheered me on every step of the way. Thank you Ben and Den, Steffi and Elton, and all the Barry, Camilleri, Buttigieg, Croom, and Henderson families.

The main reason why my time as a PhD student will be special to me is that it overlapped with some major milestones in my relationship with the love of my life, Laura. The PhD marked the start of our life together under the same roof, in bonny Scotland. This thesis, although it brings back memories of long hours of work, will also always remind me of our travels together, our wedding day, and us moving into our own home. And so I dedicate my biggest thank you to my wonderful wife, Laura. You have provided me with unwavering support every single day from the very start, and you have believed in me at times when I did not believe in myself. Inħobbok Bubu tiegħi.

Lastly, I wish to thank my kitties, Milly and Toffee, for all their cuddles throughout the course of my PhD. Milly, thanks for all the early-morning "get out of bed" encouragement, keeping me accountable! Toffee, I love how you keep me company whilst I work, and always manage to step on my keyboarjgfsiud.

Declaration

Except where explicit reference is made to the contribution of others, all work in this thesis was carried out by the author and has not been submitted for any other degree at the University of Glasgow or any other institution. The novel contributions presented in this thesis have been published at peer-reviewed conference proceedings. Consequently, the contents of Chapters 4–6 are based on the author's publications [72, 69, 71].

Chapter 1

Introduction

In spite of large investments into fault-prevention techniques, failures still regularly occur in complex distributed applications [50, 121, 5]. It is widely agreed that traditional methods of verification using software testing do not provide high-enough levels of confidence in the correctness of distributed programs [24, 73, 105, 16]. This is due to the complexity of the environment in which they operate. Not only do distributed systems need to reason about the intricacies of *concurrency*, but they must also be built to withstand some degree of *failure*. This is because, realistically, networks can never be completely reliable: machines may lose power; wires linking systems may be broken; wireless messages are susceptible to interference and weak connections, etc. Therefore, it is often desirable for the software designed to run over distributed networks be built to provide some degree of *fault tolerance*—i.e., redundant operations are added to the program in order to still produce correct outputs even when some failures occur at runtime [68]. This merely increases the complexity of correctly designing and implementing such programs, reinforcing the need for more rigorous methods of verification for such systems.

Formal verification [15] refers to the use of rigorous mathematical techniques to determine whether a design specification conforms to a well-defined notion of correctness. Many such techniques exist. In the realm of concurrency and distributed systems, notable examples include *model checking* [8]—specifying the execution of programs as labelled transition systems and exhaustively exploring the model to check it against properties specified in a formal logic. For instance, the mCRL2 [49] model checking tool synthesises labelled transition systems of concurrent programs from a specification language, and can verify said models against properties written in the modal μ-calculus [64]. The benefit of model checking is that, assuming a correct model of a system, verification of desirable properties can be both *sound and complete* (given the model checker has access to the entirety of a program's state space). The natural drawback of this verification method being the mentioned assumption—any guarantees of correctness are determined for the model, as opposed to the code artifact.

On the other end of the formal verification spectrum is *runtime verification* [10, 39, 41]. This is lightweight technique for providing guarantees on a single execution trace; whereby,

properties of system behaviour are specified in a formal language and used to mechanically synthesise *monitors*, which are then *instrumented* with the system at runtime. A main benefit of runtime verification is that monitors are fed detailed information on the actual events observed in a system's runtime environment, allowing monitors to provide accurate feedback on the ongoing status of correctness post-deployment. A limitation inherent to the nature of monitorability is that guarantees of correctness cannot be determined for a system as a whole, but rather only for the observed execution trace up-to the latest observed event.

Types and type systems [91] lie in-between these two aforementioned techniques. They are lightweight forms of verification baked directly into programming languages. Types provide guarantees directly on handwritten code and aid developers in implementing software which is correct by construction. Specific to concurrent and distributed computing, *session types* [46] describe *what* and *how* data should be communicated within a program. They are a *behavioural* type system which encode communication protocols and ensure that implemented code observes its prescribed communication pattern. Thereby, session types bridge the gap between the verification of protocols and their implementation, whilst providing guarantees of properties which are proven to be upheld during a system's runtime. To this aim, session types have been implemented in various programming languages, including Java [63, 34], Go [67], Haskell [60, 87], Scala [100], Rust [65], Elixir [42], Links [38].

Since their inception [53, 55], most session type theories have been tailored to concurrent processes, as opposed to failure-prone distributed systems—i.e., it is commonly assumed that communication errors do not occur. The works that do consider failures in their communication model tend to assume a baseline degree of reliability, e.g. communication occurs over the Transport Control Protocol (TCP). This thesis strips away these reliability assumptions and explores the viability of session types for describing protocols of fault-tolerant programs, designed to operate over failure-prone networks. The following introduces session types as a means of describing communication protocols, and motivates why they should be considered for failure-prone and fault-tolerant systems.

Protocols as types. Session types are typically categorised into *binary* or *multiparty*.

The former describes the communication pattern of a binary endpoint channel. For example, the type of a channel which sends ($\oplus$) an integer and receives (&) back a string could be:

$$\oplus\texttt{Int}.\&\texttt{String}.\textbf{end}$$

This type would be associated with one endpoint of the channel, ensuring that communication stemming from that endpoint observes the prescribed pattern. The other end of the channel should observe the *dual* type:

$$\&\texttt{Int}.\oplus\texttt{String}.\textbf{end}$$

Adhering to this type guarantees that no *deadlocks* will occur at runtime w.r.t. the use of this channel. It does not however guarantee that an entire system is free from locks since deadlocks may still result from incorrect interleaving of actions on different channels. Additional mechanisms are required to prevent these issues, e.g. by prioritising the order in which channels can be used [31].

On the other hand, *multiparty* session types (MPST) assign a type to a multiparty session, rather than a binary channel, aiming to provide communication-centric guarantees for systems with interleaved communication between many named participants. For example, in a system with a *client*, *server* and *worker*, a multiparty protocol may be:

$$c : s \oplus \mathsf{request}(\mathtt{Int}) \,.\, w \,\&\, \mathsf{response}(\mathtt{String}) \,.\, \mathbf{end}$$
$$s : c \,\&\, \mathsf{request}(\mathtt{Int}) \,.\, w \oplus \mathsf{forward}(\mathtt{Int}) \,.\, \mathbf{end}$$
$$w : s \,\&\, \mathsf{forward}(\mathtt{Int}) \,.\, c \oplus \mathsf{response}(\mathtt{String}) \,.\, \mathbf{end}$$

The above types describe the communication protocol from the perspective of each participant in a multiparty session. For instance, the type of the *c*lient begins by making a request to the *s*erver ($s \oplus \mathsf{request}(\mathtt{Int})$) before waiting for a response from the *w*orker ($w \,\&\, \mathsf{response}(\mathtt{String})$). A key difference to observe in MPST is that output and input actions must be directed by a role name (e.g. *c*, *s*, and *w*) denoting to (resp. from) whom a message is sent (resp. received). Since distributed protocols often involve interleaved and cyclic communication between multiple participants, this thesis specifically focuses on MPST, as they are better suited for representing such configurations [110].

Computer networks. Networks are often described as a stack of protocols [107]. For instance, an application-level protocol may operate over the Transport Layer Protocol (TCP) [96], which in turn operates over the internet protocol (IP) [95], which in turn may be operating over Ethernet [35], etc. Each level on the stack aims to provide certain guarantees not available to the layers below it. For instance, TCP guarantees reliable and ordered transmission of IP packets, whereas IP guarantees that packets observe a standard pattern. Alternatively, the User Datagram Protocol (UDP) does away with reliability and ordering guarantees for applications which may wish to provide their own mechanisms for dealing with unreliability, or for applications valuing speed over accuracy. Typically, this protocol sees use in the transmission of voice and video as well as *client-server* applications [93]. Session type theories thus far typically employ reliable communication semantics, similar to communicating over TCP. This thesis takes the first steps towards exploring the use of session types in less reliable communication models as a means of verifying communication-centric properties of programs designed to operate over low-level network protocols.

The client-server paradigm. This paradigm refers to a method of programming concurrent and distributed applications in which a client consists of a process making a request to a service, and a server consists of an infinitely available process offering some service. The paradigm promotes reliable, compositional and modular programs. It is central to programming standards and is a common design pattern in, e.g., the Open Telecom Platform (OTP) [74] in Erlang [7]; which has proven to be effective at building highly-available and scalable distributed applications in practice [6]. Since this thesis explores session types within models of communication similar to the guarantees of UDP, which in turn has typical applications within client-server systems (e.g. Domain Name Systems [83]), then it is only natural that the client-server paradigm should be explored within session types as well. Such efforts have already been made for binary session typed systems [97, 88, 98], but are yet to be explored in MPST.

1.1 Research Questions

This thesis addresses the following questions.

How can multiparty session types be adapted to support the client-server paradigm?

The client-server paradigm is a widely adopted technique for writing distributed programs. The paradigm involves describing servers that are designed to be infinitely available to respond to requests made by clients. The technique promotes modular design of components, and allows for easy integration of services (as servers are allowed to act as clients to other servers). How can MPSTs be adapted to easily design multiparty client-server systems where servers are designed to be infinitely available? MPST introduce new challenges when answering this question. With binary channels, all parties involved in communication over the channel are immediately aware of the infinite status of the server. However, in a multiparty system, requests to a service may result in communication to participants other than the client or server; thus mechanisms are required to ensure that any middleman caught between a client and server is also capable of infinitely handling its communication. Can properties such as deadlock freedom and termination be partially determined for such systems? How can protocols be modularly designed without prior knowledge of the clients that will operate in the final system?

How can multiparty session types be adapted to verify behavioural properties of programs designed to run over failure-prone networks?

Applications designed to run over low-level network protocols have varying reliability assumptions. For example, protocols designed to run over the User Datagram Protocol (UDP) must take into account the possibility of message loss, delay and reordering. Is it possible for MPSTs to verify behavioural properties of protocols designed to run over such failure-prone modes of communication? Modelling such unreliable methods of communication introduces challenges

for verification which were previously unexplored within MPST. For example, by considering messages sent between any participants to have no notion of ordering, standard definitions of *safe communication* become unusable. This raises the question, what does safety of unreliable communication even mean? Lastly, given this unreliable setting, detecting and handling failure becomes crucial. What type-level mechanisms can be used to reason about the possibility of failure-prone communication?

Are multiparty session types suitable for formalising real-world fault-tolerant network protocols?

A method of establishing a degree of fault tolerance in network protocols is by utilising the client-server paradigm. If servers are designed to remain infinitely available, clients may reissue failed requests with minimal failure-handling required by the server. Various real-world network protocols observe this pattern, e.g., the Internet Control Message Protocol (ICMP), and Domain Name Systems (DNS). Can the client-server paradigm be used within MPSTs to design fault-tolerant protocols intended to run over failure-prone networks? Can any real-world descriptions of fault-tolerant protocols be formalised into MPST?

1.2 Contributions

This thesis serves as evidence towards the following statement.

Multiparty session types may be adapted to describe protocols for distributed and failure-prone systems. They are capable of verifying communication-centric properties of fault-tolerant programs, even when using imperfect failure detectors, by leveraging the client-server paradigm.

The evidence presented is in the form of three novel languages, each using a new MPST theory targetting client-server, failure-prone and fault-tolerant systems respectively; where the unit of failure is message loss and fault tolerance is introduced through the client-server paradigm. The languages and their novel contributions are listed below.

MPST! The first language with a MPST system to use *replication* and *first-class roles* as a means of describing client-server protocols (Chapter 4). The contributions of MPST! are:

1. Proofs of *subject reduction*, *session fidelity* and *property verification* for a generalised MPST system using replication and first-class roles. A key point of novelty in the type system is the use of replication and parallel composition in the *type semantics*.

2. Expressivity results. Replication is shown to increase the expressiveness of the type system to be capable of describing context-free protocols, thus serving as the first account of context-free MPST. Replication and recursion are also shown to be mutually non-inclusive, and a type system using both constructs is capable of describing protocols relying on *communication races* and *mutual exclusion*. These results are demonstrated through examples, such as *binary tree serialisation*, the *dining philosophers problem*, and an *online auction*.

3. Decidability results. The type system is shown to be non-representable as a finite state machine, and the decidability of the type system is proven to be dependent on the decidability of a co-inductive property used for type checking. Lastly, sublanguages of MPST! are identified for which this property is decidable.

Le Brun, M. A., Fowler, S., and Dardha O.
Multiparty Session Types with a Bang!.
European Symposium on Programming, 2025.

MAGπ The first language with a MPST system to use *asynchronous bag buffers*, *semantics describing message inconsistencies* and *timeouts as imperfect failure detectors* (Chapter 5). The contributions of MAGπ are:

1. Proofs of *subject reduction*, *session fidelity* and *property verification* for a generalised MPST system using timeouts as imperfect failure detectors. Type checking is shown to be *undecidable*, but sublanguages for which it becomes decidable are identified.

2. Key observations of using MPST with bag buffers and non-deterministic timeouts are discussed in detail. It is shown that standard notions of *session subtyping* and *safety* cannot be integrated into a MPST system with type-level bag buffers. Furthermore, non-deterministic timeouts in the type semantics are expressive enough to model various forms of failure. Methods of identifying subgraphs of the state space produced by the type semantics are presented, allowing the same type-specification of a protocol to be studied under various failure models.

3. Reconfigurations of MAGπ are discussed for communication models inspired by various network protocols, including e.g., the User Datagram Protocol (UDP) [93], the Real-time Transport Protocol (RTP) [44], and the Transmission Control Protocol (TCP) [96].

Le Brun, M. A., and Dardha O.
MAGπ: Types for Failure-Prone Communication.
European Symposium on Programming, 2023.

MAGπ! The first language with a MPST system to use *replication* as a means of describing *fault-tolerant protocols* (Chapter 6). The contributions of MAGπ! are:

1. Proofs of *subject reduction*, *session fidelity* and *property verification* for a generalised MPST system using timeouts as imperfect failure detectors and replication to describe fault-tolerant servers. Again, type checking is *undecidable*, but sublanguages are identified for which it becomes decidable.

2. The type system's ability to describe fault-tolerant programs is demonstrated through various examples inspired by real-world distributed protocols, e.g. a *load balancer*, the *ping utility*, and a *learning switch*.

Le Brun, M. A., and Dardha O.
MAGπ!: The Role of Replication in Typing Failure-Prone Communication.
Int. Conf. on Formal Techniques for Distributed Objects, Components and Systems, 2024.

1.3 Thesis Structure

Chapter 2 provides the minimal necessary background material for the subject area discussed in this thesis, namely expounding on process calculi and general concepts of session types. Chapter 3 serves two purposes. Firstly, the chapter introduces the *bottom-up* approach to multiparty session types [103]—the chosen approach for the type systems presented in this thesis. The chapter begins with a discussion on why this methodology was chosen for this work, and proceeds by demonstrating how such a type system is developed by example on a multiparty session calculus. This leads to the second purpose of the chapter, the *core calculus*. A calculus is designed to be easily extended in the following chapters, with the aim of reducing the cognitive load required to realise the novel languages and type systems presented.

The next three chapters present the novel contributions of the thesis, each by addressing a different extension to the core calculus and presenting a new language. The chapters follow similar structures: *(i)* an extension to the core calculus is identified targetting issues related to its use within fault-tolerant distributed systems; *(ii)* the language extension is formalised and the relevant metatheory is proven; and *(iii)* key features of the language are discussed in further detail. Chapter 4 addresses the integration of the client-server paradigm into MPST, and presents MPST!, an extension to the core calculus with *replication* and *first-class roles*. Chapter 5 addresses failure-prone communication by introducing asynchronous bag buffers and failure semantics. The chapter presents MAGπ, the first MPST theory to use timeouts as imperfect failure detectors. Chapter 6 blends both of the previous extensions into a single language, MAGπ!, capable of describing and verifying properties of fault-tolerant distributed client-server systems.

Lastly, Chapter 7 concludes the thesis by reiterating over the novel contributions and giving insight into future work.

Chapter 2

Preliminaries

The novel contributions of this thesis will be presented in Chapters 4–6, which have a common dependency on Chapter 3. Chapter 3, whilst being self-contained, assumes knowledge on the broader areas of *process calculi* and general concepts of *session types*. Hence, this chapter aims to provide a minimal set of information on these topics for unfamiliar readers to increase its accessibility.

2.1 Process Calculi

Calculi in programming languages are formal representations of a given language, used for rigorously reasoning about and analysing computation in a system. Many kinds of language calculi exist, each targeting a specific domain. For example, the λ-calculus [27, 28] is the foundation for reasoning about *functional* languages, and has given rise to programming languages now used on an industrial scale (e.g. Haskel [58, 43], Scala [85], Rocq [26]).

In the world of *concurrency*, process calculi provide the foundation for reasoning about concurrent and distributed *processes*. These calculi model computation as *communication* between processes running in parallel. One of the early process calculi is the Calculus of Communicating Systems (CCS) [77, 78], modelling communication over binary-endpoint channels. Session types are often integrated and studied within calculi similar to Milner's π-calculus [80, 99]; an extension to CCS permitting *mobility*—the ability to send and receive channels over channels.

2.1.1 Polyadic π-Calculus

A standard variation of the language is the *polyadic* π-calculus [80, 81, 82, 99], of which a finite version is presented in Figure 2.1. Polyadicity refers to the ability of sending and receiving any number of channels in payloads.

Processes are given by $P, Q, \ldots$, and may describe any of the following. *Prefixes* denote actions, specifically either *output* $\overline{x}\langle\vec{y}\rangle.P$ where channels $\vec{y}$ are sent over channel x, or *input*

$$
\begin{array}{rll}
P, Q ::= & \sum_{i \in I} \alpha_i.P_i & \text{(choice of prefix)} \\
\mid & P \mid Q & \text{(parallel composition)} \\
\mid & (\nu x)P & \text{(channel restriction)} \\
\mid & \mathbf{0} & \text{(inaction)} \\
\alpha ::= & \overline{x}\langle \vec{y} \rangle & \text{(output)} \\
\mid & x(\vec{y}) & \text{(input)}
\end{array}
$$

Figure 2.1: Syntax of the finite polyadic π-calculus

$x(\vec{y}).P$ where channels $\vec{y}$ are received via channel x. Both actions have a continuation process P, invoked after the action is completed. Actions are enclosed within a non-empty summation, denoting a nondeterministic choice. Singleton choices imply deterministic actions.

Processes may be *composed in parallel* $P \mid Q$, allowing both processes to simultaneously be available for communication. *Restriction* $(\nu x)P$ creates a new private channel to be used inside of P. Lastly, the empty process $\mathbf{0}$ denotes a process that is *inactive*.

The calculus given in Figure 2.1 so far has no means of expressing infinite computation. There are two standard constructs used for doing so in the π-calculus. The construct most familiar to other programming languages is *recursion*. By extending the previous finite calculus with the following, the calculus becomes capable of expressing infinite computations and is Turing-powerful.

$$P, Q ::= \quad \ldots \mid \mathrm{X}\langle \vec{y} \rangle \qquad \text{(recursive call)}$$

Given a set of process definitions in the form of $\mathrm{X}(\vec{x}) = P$, recursive calls reduce to the unrolled process $P\{\vec{y}/\vec{x}\}$, substituting free occurrences of any $\vec{x}$ in P with the corresponding $\vec{y}$.

The alternative construct is that of *replication*.

$$P, Q ::= \quad \ldots \mid !P \qquad \text{(recursive call)}$$

Replication $!P$ represents infinite copies of process P composed in parallel. It is key to note that for a polyadic π-calculus with binary-endpoint channels, both constructs are equivalent in expressive power [4], whereas for a calculus with strictly empty payloads (akin to CCS) recursion is more expressive than replication [18].

2.1.2 Multiparty Session Calculus

A generalisation of binary-endpoint channels is *multiparty sessions*, allowing for two or more participants to communicate over the same object—a *session*. A simplified multiparty session

$$
\begin{array}{rll}
P, Q ::= & \sum_{i \in I} \alpha_i.P_i & \text{(choice of prefix)} \\
\mid & P \mid Q & \text{(parallel composition)} \\
\mid & (\nu s)P & \text{(session restriction)} \\
\mid & \mathbf{0} & \text{(inaction)} \\
\mid & \mathrm{X}\langle \vec{x} \rangle & \text{(recursive call)} \\
\alpha ::= & c[\boldsymbol{p}] \oplus \mathsf{m}\langle \vec{d} \rangle & \text{(output)} \\
\mid & c[\boldsymbol{p}] \& \mathsf{m}(\vec{x}) & \text{(input)} \\
c, d ::= & s[\boldsymbol{p}] \mid x & \text{(endpoints)}
\end{array}
$$

Figure 2.2: Syntax of a multiparty session calculus

calculus [56, 57, 100] (based on that originally proposed by Bettini et al. [14], later revised by Coppo et al. [30]) is given in Figure 2.2.

The main difference is the use of *role indexed sessions* in prefixes. Given actions no longer occur over binary-endpoint channels, the source and destination of an action need to be specified. This is achieved by indexing sessions with *roles* ($\boldsymbol{p}$, $\boldsymbol{q}$, ...), representing the participants in a session. For example, $s[\boldsymbol{p}][\boldsymbol{q}] \oplus \mathsf{m}\langle \vec{d} \rangle$ denotes sending a message with label m and payload $\vec{d}$ on session s from $\boldsymbol{p}$ to $\boldsymbol{q}$. Similarly, $s[\boldsymbol{p}][\boldsymbol{q}] \& \mathsf{m}(\vec{x})$ denotes receiving a message with label m and payloads bound to variables $\vec{x}$ on session s at $\boldsymbol{p}$ from $\boldsymbol{q}$. An endpoint in this calculus therefore is a tuple of (*i*) the session in which the action should occur, and (*ii*) the role performing the action.

It is typical for choice in this calculus to be restricted in some way, e.g. by requiring prefixes in a choice to be all of the same action stemming from the same endpoint. These restrictions are usually required by the type system to guarantee *session fidelity*, i.e., ensuring that processes adhere to the types of sessions. Relaxing these restrictions is a topic of ongoing research. Sender-driven choice—the ability to choose between sending to different participants—can be permitted whilst still maintaining type safety [75, 106]. Furthermore, there are strict hierarchies of expressivity when it comes to relaxing such restrictions and allowing undirected or mixed choice [90]. This topic of discussion will be revisited in reference to the languages later developed in this thesis.

2.2 Session Types

When session types were initially introduced [53, 55, 120], they were designed to operate over a calculus slightly different to that presented in Figure 2.1. A polished calculus was later presented with the fundamentals of designing and using session types by Vasconcelos [114, 113]. The key difference between the calculus used in these works and that of Figure 2.1 is that binary-endpoint channels were modelled using two names as opposed to one (a name for each endpoint of the

channel). This makes attaching a binary session type more natural, and lends itself well to the concept of *duality*.

2.2.1 Binary Session Types

Consider a channel used to send an integer and receive back a Boolean. The operation on the sent data is, for now, abstracted using a function $f(a)$.

$$(\nu xy)\ \bar{x}\langle 5\rangle.x(b).\mathbf{0} \mid y(a).\bar{y}\langle f(a)\rangle.\mathbf{0}$$

Using the syntax presented by Vasconcelos [114], channel restriction binds two names x and y, representing each endpoint in the binary channel. Hence, when the number 5 is sent over x, it is received over y in the parallel process. In other words, the actions observed at x should be *dual* to those observed at y, and vice-versa.

Binary session types can be used to give a type to the channel xy. Consider the following type associated to the endpoint x.

$$\oplus\texttt{Int}.\&\texttt{Bool}.\mathbf{end}$$

This can be read as "send ($\oplus$) an integer, then receive ($\&$) a Boolean, then end".[1] The type of y can be computed from the type of x using the dual function.

$$\begin{aligned}\overline{\oplus\texttt{Int}.\&\texttt{Bool}.\mathbf{end}} &= \&\texttt{Int}.\overline{\&\texttt{Bool}.\mathbf{end}}\\ &= \&\texttt{Int}.\oplus\texttt{Bool}.\overline{\mathbf{end}}\\ &= \&\texttt{Int}.\oplus\texttt{Bool}.\mathbf{end}\end{aligned}$$

The metatheory for session types then provides a number of guarantees on processes that are well-typed. For instance, the above example will only type-check if function $f(a)$ produces a Boolean. Therefore, e.g. if $f = \texttt{is\_prime}$ then the process type-checks, but $f = \texttt{to\_string}$ produces a typing error.

Furthermore, session type theory is powerful enough to provide communication-centric guarantees, such as *deadlock-freedom*—i.e., processes do not get stuck waiting for messages that will never arrive. Such guarantees are possible due to *linearity* constraints; meaning that objects given a session type cannot be (prematurely) destroyed or duplicated.

This notion of linearity stems from Girard's linear logic [47], a logical system for reasoning about consumable resources. Linear resources are ones which must be used exactly once.

'I have one apple, therefore I can eat an apple.'

[1] The syntax used to denote send and receive types here is non-standard. Typically, these are written as ! and ? respectively. However, these symbols will be used differently later on, and so $\oplus$ and $\&$ are used to stick to the conventions established in this thesis and to avoid overloading type constructors.

The above proposition demonstrates a linear resource (the apple), and how it can only be used once. Sadly, one cannot eat an apple many times if they only have *one* apple.

In fact, linear logic and session types can be seen as two sides of the same coin. The seminal work by Caires and Pfenning [20] shows how dual intuitionistic linear logic [9] can be used as a (session) type system for the π-calculus, and a classical interpretation was later given by Wadler [117]. Furthermore, Dardha et al. [32] provide semantic- and type-preserving encodings between the session calculus of Vasconcelos [114] and the linear π-calculus [59].

Session type theories typically concern themselves with studying the communication-centric properties that can be statically enforced on a programming language. The most common of which include:

Safety No unexpected messages are ever received;

Deadlock freedom Processes do not get stuck waiting for messages that will never arrive;

Termination Processes are guaranteed to reach the inactive process;

Lock freedom Under fair scheduling, all input and output actions are eventually triggered;

Liveness All input and output actions are eventually triggered, regardless of scheduler.

2.2.2 Multiparty Session Types

As previously mentioned, binary-endpoint channels were later generalised into multiparty sessions which can be typed using Multiparty Session Types (MPST). Originally introduced by Honda et al. [56, 57], and later elaborated by Bettini et al. [14] and Coppo et al. [30], MPST could be described using a *top-down* approach. Also referred to as *global types*, top-down MPST allow the interactions between multiple participants to be described from a global view of the system.

Recalling the *client*, *server*, *worker* example from the introduction, the same protocol may be described using the following *global type*.

$$c \to s : \mathsf{request}(\texttt{Int}) \,.\, s \to w : \mathsf{forward}(\texttt{Int}) \,.\, w \to c : \mathsf{response}(\texttt{String}) \,.\, \mathbf{end}$$

The types for individual participants, called *local types*, can then be generated from the global type using the *projection* operation. Top-down MPST systems aim to provide well-formedness restrictions on global types which guarantee communication-centric properties on the local types they produce through projection. Scribble [54, 119] is a tool for protocol descriptions that achieves exactly this outcome.

Given the name "top-down" for this previous approach to MPST, the existence of a *bottom-up* approach should be of no surprise. Also referred to as *generalised* MPST, the bottom-up approach [103] proves its metatheory directly on local types, removing the reliance on global types and their projections. The bottom-up approach is central to this thesis, and as such the next chapter is dedicated to giving an in-depth technical background on the subject; demonstrating the

approach by example on a core calculus used for developing the novel contributions presented in the subsequent chapters.

Chapter 3

A Common Core: Generalised MPST

3.1 Introduction

This thesis aims to determine whether multiparty session types can be used to statically verify behavioural properties for distributed and failure-prone message-passing systems.

Such a setting is complex and broad. It would not make much sense to immediately attempt to apply MPST theory to a failure-prone calculus capable of modelling complex distributed protocols as the causality of results could be unclear. It is important that, e.g., if an extension to a language were to increase the expressiveness of the type system, the exact feature causing the added expressiveness should be identified. Furthermore, if the decidability of the type system were to be affected, the cause of undecidability should be known. To this aim, the coming chapters modularly introduce new features into a language under scrutiny, allowing for a more granular understanding of dependencies between language features and the capabilities of types.

"Distributed systems" covers a wide range of protocols, with massively varying assumptions, guarantees, and requirements. A single traditional MPST system would not be suitable for a language intended to be used for such a broad setting. For instance, a system used to write peer-to-peer protocols guaranteeing termination would not be suitable for client-server protocols where servers are designed to never terminate. One could address this by designing a plethora of languages, each with their own assumptions and guarantees, to be used for a targetted group of protocols. This thesis takes a more *general* approach.

Generalised MPST [103, 118] are an approach to MPST theory which, instead of syntactically guaranteeing properties from a well-formed global specification, provide a framework for verifying runtime properties directly from the local protocols of each participant. Importantly, verification of properties is decoupled from protocol design, allowing the same generalised framework to be used to verify protocols of different requirements. The approach involves proving the metatheory parametric on some largest property on type contexts, and offloads the responsibility of verifying that the types of a program are captured by this property. The benefit is that a language can be made more expressive and general purpose as the verification of prop-

erties is handled *a posteriori*. The trade-off is that the theory no longer produces a standalone tool, as the generalised approach requires users to manually verify the adherence of types to properties. This could be achieved, e.g., by generating local types from global type projections (as they are guaranteed to observe liveness [118]), or by exhaustively traversing the state-spaces produced by local types (for instance, via model checking). The latter being more expressive at the cost of complexity [111]. This thesis adopts the generalised approach to MPST for the following reasons:

1. **Protocol-specific requirements.** Distributed protocols have a wide range of assumptions, outcomes, and desired properties. Why design a language that enforces termination when many distributed protocols are designed to never terminate? Why relax the termination restriction for the protocols that should? A generalised theory allows a *single* language to be used for protocols of multiple purposes with ranging requirements of communication properties; all whilst keeping the language tailored to distributed communication.

2. **Minimising bias.** The generalised approach is more expressive than standard global types w.r.t. the number of safe protocols they capture [103]. Since this work will involve examining the effects of features added into a language, using a most expressive type system as a starting point aims to minimise bias for any expressiveness gained through added features.

3. **Localised behaviours.** The constructs that are studied in this thesis focus on describing local behaviours of individual participants. They specifically are not meant to provide global information. For example, Chapter 5 explores the use of *timeouts* as *imperfect failure detectors*, i.e., a timeout can by used by a participant to assume a failure of some message. Importantly, participants should not be immediately aware of timeouts occurring at different locations. (This would be inherently embedded into a global type.) Furthermore, in Chapter 4, the use of *replication* is studied in a system where the full list of participants may be unknown when designing protocols. Global types are antithetical to such an approach since they assume knowledge on all participants at play.

This chapter serves as a background on generalised MPST [103] and introduces the approach *by example* on a core calculus—a general starting point for language investigation—on which the novel contributions of the thesis are subsequently built upon. The core calculus models session-based communication between multiple participants; it is an adaptation of standard multiparty session π-calculi [29, 57, 101]. Although it does not present novelty through language features, the exact calculus presented is unique in design and presentation as it is built to be seamlessly extended and includes concepts focused on making it more usable.

$$
\begin{array}{rl}
\textit{Concretes} & d ::= s[\boldsymbol{p}] \mid v \\
\textit{Binders} & b ::= x \\
\textit{Channels} & c ::= x \mid s[\boldsymbol{p}] \\
\textit{Values} & V ::= c \mid v \\
\\
\textit{Processes} & P,Q ::= (\nu s)P \mid P|Q \mid \textstyle\sum_{i\in I} P_i^{\oplus} \mid P^{\&} \mid \mathrm{X}\langle\vec{V}\rangle \mid \mathbf{0} \\
\textit{Send Process} & P^{\oplus} ::= c[\boldsymbol{q}] \oplus \mathsf{m}\langle\vec{V}\rangle \,.\, P \\
\textit{Receive Process} & P^{\&} ::= c[\boldsymbol{q}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, P_i \\
\\
\textit{Programs} & \mathscr{P} ::= (P, D) \\
\textit{Definitions} & D ::= \emptyset \mid D, \mathrm{X} \mapsto (\vec{x}, \vec{T}, P) \\
\textit{Protocols} & \Psi ::= \{\boldsymbol{p} : S_{\boldsymbol{p}}\}_{\boldsymbol{p}\in I}
\end{array}
$$

Figure 3.1: Syntax of Core Calculus

3.2 Calculus

This section introduces the *core calculus*, serving as a common starting point for investigation. Subsequent chapters will refer to this calculus, using it as a building block for novel features.

3.2.1 Syntax

Figure 3.1 shows the syntax of the *core calculus*.

Names, Values, and Binders. The *concrete* primitives of the core calculus are the: *(i) session with role*, represented by a *session name* (ranged over by $s, s', \ldots$) indexed with a *role name* (ranged over by $\boldsymbol{p}, \boldsymbol{q}, \ldots$)—together, $s[\boldsymbol{p}]$ denotes a participant $\boldsymbol{p}$'s endpoint within the multiparty session s; and *(ii) basic values* v (naturals, reals, units, strings, etc.). A *binder* b captures all kinds of *variables*; for the core calculus, there is only one, ranged over by $x, y, \ldots$. *Channels* c can be represented by a session with role or a variable, and *values* V capture channels and basic values.

Processes. Processes $P, Q, \ldots$ describe how channels can be used to send and receive information within multiparty sessions. Session *restriction* $(\nu s)P$ denotes the creation of a new multiparty session s used inside of P. Sessions can be associated with a *protocol* Ψ (e.g. $(\nu s : \Psi)P$), but more on this later. Processes can be *composed in parallel* $P|Q$, denoting the concurrent computation of processes P and Q. *Choice of sends* $\sum_{i\in I} P_i^{\oplus}$ refers to a non-deterministic choice between sending messages; with a singleton choice representing a deterministic send process. (A well-formedness condition is assumed where all channels in the choice are the same.) A

send process $c[q] \oplus \mathsf{m}\langle \vec{V} \rangle \,.\, P$ denotes sending a message over channel c towards participant q with label m and payload $\vec{V}$, before continuing according to P. Similarly, a *receive process* $c[q] \,\&_{i \in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, P_i$ denotes a *branching* receipt of a message over channel c from role q; with the chosen branch depending on the received message label m_i, then binding received payloads to $\vec{b}_i$, before continuing according to P_i. Lastly, *process call* $\mathrm{X}\langle \vec{V} \rangle$ models recursion, passing arguments $\vec{V}$ to a process with name X; and the *inactive process* $\mathbf{0}$ is a terminated process.

Programs, Definitions, and Protocols. A *program* is what is written by an end-user of the language; it is a tuple (P, D) of a "main" process P and process *definitions* D. Definitions map process names (ranged over by X,Y,...) to process bodies $(\vec{x}, \vec{T}, P)$ to be used for recursion, where $\vec{x}$ is a list of accepted arguments, $\vec{T}$ is a list of types for said arguments, and P is the process in which the arguments are used. Finally, *protocols* are *session type* specifications of roles in a given session. Formally, Ψ is a set of role-session type pairs, $\{p : S_p\}_{p \in I}$ for a non-empty I, defining the communication patterns that each participant of a session should follow.

Example 3.1 (Load Balancer: Program). A load balancer architecture typically involves a *client* (c) making requests, a server (srv) accepting requests, and some number of workers (w_i) servicing requests. The below is a simplified example program of a load balancer $\mathscr{P}_{lb} = (P, D)$, where a client makes one request to a load balancer with two workers.

$$
\begin{aligned}
P = {} & (\nu s) \\
& \mathrm{Srv}\langle s[srv] \rangle \mid \mathrm{Wrk}\langle s[w_1] \rangle \mid \mathrm{Wrk}\langle s[w_2] \rangle \\
& \mid s[c][srv] \oplus \mathsf{req}\langle 42 \rangle \,.\, s[c][srv] \,\&\, \left\{ \begin{array}{l} \mathsf{worker1}() \,.\, s[c][w_1] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \\ \mathsf{worker2}() \,.\, s[c][w_2] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \end{array} \right. \\
D = {} & \{ \\
& \mathrm{Srv}(r : S_{srv}) \mapsto r[c] \,\&\, \mathsf{req}(x) \,.\, \left(\begin{array}{c} r[w_1] \oplus \mathsf{fw}\langle x \rangle \,.\, r[c] \oplus \mathsf{worker1}\langle\rangle \,.\, \mathbf{0} \\ + \\ r[w_2] \oplus \mathsf{fw}\langle x \rangle \,.\, r[c] \oplus \mathsf{worker2}\langle\rangle \,.\, \mathbf{0} \end{array} \right) \\
& \mathrm{Wrk}(w : S_w) \mapsto w[srv] \,\&\, \mathsf{fw}(y) \,.\, w[c] \oplus \mathsf{ans}\langle f(y) \rangle \,.\, \mathbf{0} \\
& \}
\end{aligned}
$$

The main process P begins by restricting a new session s. A server and two worker processes are composed in parallel using process calls, passing the respective channel to each process name. The client behaviour is written directly into the main process. It starts by offering a request to the server with a payload. The server begins listening for a request from the client, subsequently choosing a worker to handle the request and then informing the client of the choice of worker. A worker handles a forwarded request if chosen by the server. The client then waits for a message from the server informing it of the worker that

$$
\begin{array}{rl}
\textit{Static types} & T ::= S \mid B \\
\textit{Basic types} & B ::= \texttt{Nat} \mid \texttt{Real} \mid \texttt{String} \mid \cdots \\
\textit{Session types} & S ::= S^{\oplus} \mid S^{\&} \mid \mu \mathrm{t}.S \mid \mathrm{t} \mid \mathbf{end} \\
\textit{Selection type} & S^{\oplus} ::= \oplus_{i \in I} \boldsymbol{p_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \\
\textit{Branching type} & S^{\&} ::= \&_{i \in I} \boldsymbol{p}{:}\mathsf{m}_i(\vec{T}_i).S_i
\end{array}
$$

Figure 3.2: Syntax of Core Calculus Types

has been chosen, and lastly waits to receive the final answer from said worker.

Note. Though this example serves for demonstrative purposes, it is not quite realistic given that the client needs to be aware of the server-side architecture. This is a limitation of the underlying formalism the core calculus is based upon [103]. The issue will be addressed in Chapter 4, where novel mechanisms are introduced to make clients agnostic of the server-side architecture.

Expressions and Conditional Branching. The calculus does not out-of-the-box formalise arithmetic expressions or conditional branching, as these are orthogonal to the investigation carried out in this thesis. All of the presented calculi can be routinely extended to handle these constructs, making them more of usable languages rather than research calculi and guaranteeing that they are Turing-complete.

3.2.2 Types

Figure 3.2 shows the syntax of types for the core calculus.

Static Types. The grammar for the type syntax is given by the type T, being either a *session type* S, or a *basic type* B. Session types S describe patterns of communication. *Selection* $\oplus_{i \in I} \boldsymbol{p_i}{:}\mathsf{m}_i(\vec{T}_i).S_i$ represents a role's *internal* choice of sending a message to one of a set of participants $\boldsymbol{p_i}$, having label m_i and payload of types $\vec{T}_i$. The channel should then be used in accordance to the continuation type of the selected path S_i. *Branching* $\&_{i \in I} \boldsymbol{p}{:}\mathsf{m}_i(\vec{T}_i).S_i$ refers to an *external* choice stemming from role $\boldsymbol{p}$, where any one of a set of messages could be received, described by the labels m_i and payloads $\vec{T}_i$. Again, the communication pattern after the branch follows the description of the continuation type of the branched path, by S_i. Type $\mu \mathrm{t}.S$ binds a new type-level tail-recursive variable t inside of continuation type S. Lastly, type **end** symbolises the termination of a communication pattern.

Conventions. It is assumed that branching and selection types contain a non-empty set of messages with pairwise-distinct message labels. It is required that all recursion variables are guarded

by a communication action.

Example 3.2 (Load Balancer: Types). To type the program in Example 3.1, session s can be annotated with the protocol Ψ below.

$$\Psi = \{srv : S_{srv},\ w_1 : S_w,\ w_2 : S_w,\ c : S_c\}$$
$$S_{srv} = \&c : \mathsf{req}(\mathtt{Int}).\oplus \begin{cases} w_1 : \mathsf{fw}(\mathtt{Int}).\oplus c : \mathsf{worker1}().\mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}).\oplus c : \mathsf{worker2}().\mathbf{end} \end{cases}$$
$$S_w = \&srv : \mathsf{fw}(\mathtt{Int}).\oplus c : \mathsf{ans}(\mathtt{String}).\mathbf{end}$$
$$S_c = \oplus srv : \mathsf{req}(\mathtt{Int}).\&srv : \begin{cases} \mathsf{worker1}().\&w_1 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \mathsf{worker2}().\&w_2 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \end{cases}$$

The protocol abstracts away the specifics of the main program whilst still accurately describing the communication patterns that can be observed at runtime.

Definition 3.3 (Subtyping). The *subtyping relation* $\leqslant$ is co-inductively defined on types by the following inference rules:

$$\frac{(\vec{T_i} \leqslant \vec{T_i'})_{i\in I} \qquad (S_i \leqslant S_i')_{i\in I}}{\&_{i\in I} p{:}\mathsf{m}_i(\vec{T_i}).S_i \leqslant \&_{i\in I\cup J} p{:}\mathsf{m}_i(\vec{T_i'}).S_i'} \qquad \frac{(\vec{T_i} \geqslant \vec{T_i'})_{i\in I} \qquad (S_i \leqslant S_i')_{i\in I}}{\oplus_{i\in I\cup J} p_i{:}\mathsf{m}_i(\vec{T_i}).S_i \leqslant \oplus_{i\in I} p_i{:}\mathsf{m}_i(\vec{T_i'}).}$$

$$\frac{S\{\mu \mathrm{t}.S/\mathrm{t}\} \leqslant S'}{\mu \mathrm{t}.S \leqslant S'} \qquad \frac{S \leqslant S'\{\mu \mathrm{t}.S'/\mathrm{t}\}}{S \leqslant \mu \mathrm{t}.S'} \qquad \frac{}{T \leqslant T}$$

In a programming language, subtyping refers to a relation between types that allows for some degree of substitutability—i.e., an operation that works for some type should also work for its subtypes [91, Part III]. Definition 3.3 defines a subtyping relation $\leqslant$ for the core calculus. The chosen convention treats "smaller" session types as ones with *less external* and *more internal* choice (in the style of Gay and Hole [45]). Subtypes are related up-to their recursive unfoldings, and the relation is *reflexive*. Subtyping is coinductively defined as it relates types up-to their infinite unfoldings, hence the rules in Definition 3.3 may be used to build infinite proof trees. Subtyping is extended pointwise to lists of types $\vec{T_1} \leqslant \vec{T_2}$, assuming the lists are of equal lengths.

Contexts. For types to be useful, they must be associated with an entity within a program. Type contexts do exactly this. Definition 3.4 defines the type context for the core calculus.

Context splitting $\boxed{\Gamma = \Gamma_1 \cdot \Gamma_2}$

$$\frac{}{\emptyset = \emptyset \cdot \emptyset} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, x : B = (\Gamma_1, x : B) \cdot (\Gamma_2, x : B)}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : S = (\Gamma_1, c : S) \cdot \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : S = \Gamma_1 \cdot (\Gamma_2, c : S)}$$

Context addition $\boxed{\Gamma_1 + \Gamma_2 = \Gamma}$

$$\frac{}{\Gamma + \emptyset = \Gamma} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{(\Gamma_1, x : B) + (\Gamma_2, x : B) = \Gamma, x : B}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad c \notin \mathsf{dom}(\Gamma_2)}{(\Gamma_1, c : T) + \Gamma_2 = \Gamma, c : T} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad c \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + (\Gamma_2, c : T) = \Gamma, c : T}$$

Figure 3.3: Type context operations.

Definition 3.4 (Type Contexts). The type context Γ maps channels to session types and variables to static types, and is defined as:

$$\Gamma ::= \emptyset \mid \Gamma, s[\boldsymbol{p}] : S \mid \Gamma, x : T$$

Context *composition* Γ, Γ' is defined iff Γ and Γ' have disjoint domains.
Contexts are subtypes $\Gamma \leqslant \Gamma'$ iff $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ and $\forall c \in \mathsf{dom}(\Gamma) : \Gamma(c) \leqslant \Gamma'(c)$.

As previously seen, *linearity* plays a key part in session types. Since type context Γ holds information on both linear session types and unrestricted basic types, Figure 3.3 defines additional operations to help enforce linearity only where it is needed (*à la* Vasconcelos [114]).

Context splitting $\Gamma = \Gamma_1 \cdot \Gamma_2$ splits a context into sub-environments, Γ_1 and Γ_2. The operation enforces linearity on session types by keeping session-typed entities in only *one* of the splits; whereas, basic types (which have no linearity restrictions) are *shared* between splits. *Context addition* $\Gamma_1 + \Gamma_2 = \Gamma$, merges two contexts into a single one. It is synonymous to context composition when the added environments have disjoint domains, and is defined for non-unique domains iff the shared entities have basic types.

Proposition 3.5. Context addition is the left inverse of context splitting, i.e., $\Gamma = \Gamma_1 \cdot \Gamma_2$ *implies* $\Gamma_1 + \Gamma_2 = \Gamma$.

Proof. By rule induction on the definition of $\Gamma = \Gamma_1 \cdot \Gamma_2$.

BC $\Gamma = \emptyset = \emptyset \cdot \emptyset$ and by Figure 3.3, $\emptyset + \emptyset = \emptyset$.

IC1 $\dfrac{\Gamma' = \Gamma'_1 \cdot \Gamma'_2}{\Gamma = \Gamma', x : B = \Gamma'_1, x : B \cdot \Gamma'_2, x : B}$ and by the inductive hypothesis and Figure 3.3, $\dfrac{\Gamma'_1 + \Gamma'_2 = \Gamma'}{\Gamma'_1, x : B + \Gamma'_2, x : B = \Gamma', x : B = \Gamma}$.

IC2 $\dfrac{\Gamma' = \Gamma'_1 \cdot \Gamma'_2}{\Gamma = \Gamma', c : S = \Gamma'_1, c : S \cdot \Gamma'_2}$. By the definition of context composition (Definition 3.4), $c \notin \mathsf{dom}(\Gamma')$, and since splitting does not grow contexts, then $c \notin \mathsf{dom}(\Gamma'_2)$. This is used with the ind. hyp. and Figure 3.3 to conclude $\dfrac{\Gamma'_1 + \Gamma'_2 = \Gamma' \qquad c \notin \mathsf{dom}(\Gamma'_2)}{\Gamma'_1, c : S + \Gamma'_2 = \Gamma', c : S = \Gamma}$. The mirrored case follows similar reasoning.

□

Proposition 3.5 states that an original context can be re-obtained after splitting by adding together the split sub-environments. This inverse property only holds in one direction, since adding *new* basic types into an environment is not reversible.

Definition 3.6 (Session Association). Contexts can be obtained from protocols by *associating* each prescribed session type to a *session with role*. Formally:

$$\mathtt{assoc}_s(\{\boldsymbol{p} : S_{\boldsymbol{p}}\}_{\boldsymbol{p} \in I}) := \{s[\boldsymbol{p}] : S_{\boldsymbol{p}}\}_{\boldsymbol{p} \in I}$$

As shown by the definitions of *session restriction*, users define session types using a protocol. Type contexts are extracted from a protocol using the `assoc` function (Definition 3.6), to be used for type checking.

The final ingredient needed for type checking is Definition 3.7. Due to the *linearity* restriction on session types, the end predicate is needed to identify contexts that can be considered as "entirely used", i.e., all the session types in that context have reached **end**.

Definition 3.7 (End-typed environment). A context is *end-typed*, written $\mathsf{end}(\Gamma)$, iff all of its session-typed entities have the type **end**.

$$\frac{}{\mathsf{end}(\emptyset)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(c : \mathbf{end}, \Gamma)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(x : B, \Gamma)}$$

Typing Rules for Values $\boxed{\Gamma \vdash V : T}$

$$\text{T-S}\ \frac{T \leqslant T'}{c : T \vdash c : T'} \qquad \text{T-Wkn}\ \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad \Gamma_1 \vdash V : T \quad \mathsf{end}(\Gamma_2)}{\Gamma \vdash V : T} \qquad \text{T-B}\ \frac{v \in B}{\emptyset \vdash v : B}$$

Typing Rules for Programs and Processes $\boxed{\vdash \mathscr{P}}$ $\boxed{\Gamma \vdash_D P}$

$$\text{T-}\mathscr{P}\ \frac{\emptyset \vdash_D P \quad \forall i \in 1..n : \{\overrightarrow{x_i : T_i}\} \vdash_D Q_i}{\vdash (P, D = \{\mathrm{X}_i \mapsto (\vec{x}_i, \vec{T}_i, Q_i)\}_{i \in 1..n})} \qquad \text{T-}\nu\ \frac{\varphi(\mathtt{assoc}_s(\Psi)) \quad \varphi \text{ is a } \textit{safety} \text{ property} \quad s \notin \Gamma \quad \Gamma + \mathtt{assoc}_s(\Psi) \vdash_D P}{\Gamma \vdash_D (\nu s : \Psi)\ P}$$

$$\text{T-X}\ \frac{D(\mathrm{X}) = (\vec{x}, \vec{T}, P) \quad \forall i \in 1..n : \Gamma_i \vdash V_i : T_i \quad \mathsf{end}(\Gamma_0)}{\Gamma_0(\cdot \Gamma_i)_{i \in 1..n} \vdash_D \mathrm{X}\langle (V_i)_{i \in 1..n} \rangle} \qquad \text{T-}\oplus\ \frac{\Gamma_\oplus \vdash c : \oplus \boldsymbol{q}{:}\mathsf{m}(\vec{T}).S \quad (\Gamma_i \vdash V_i : T_i)_{i \in 1..n} \quad \Gamma + c : S \vdash_D P}{\Gamma \cdot \Gamma_\oplus(\cdot \Gamma_i)_{i \in 1..n} \vdash_D c[\boldsymbol{q}] \oplus \mathsf{m}\langle (V_i)_{i \in 1..n} \rangle\ .\ P}$$

$$\text{T-\&}\ \frac{\Gamma_\& \vdash c : \&_{i \in I} \boldsymbol{q}{:}\mathsf{m}_i(\vec{T}_i).S_i \quad (\Gamma + c : S_i + \overrightarrow{b_i : T_i} \vdash_D P_i)_{i \in I}}{\Gamma \cdot \Gamma_\& \vdash_D c[\boldsymbol{q}]\ \&_{i \in I}\ \mathsf{m}_i(\vec{b}_i)\ .\ P_i} \qquad \text{T-}|\ \frac{\Gamma_1 \vdash_D P_1 \quad \Gamma_2 \vdash_D P_2}{\Gamma_1 \cdot \Gamma_2 \vdash_D P_1 | P_2} \qquad \text{T-}+\ \frac{(\Gamma \vdash_D P_i^{\oplus})_{i \in I}}{\Gamma \vdash_D \sum_{i \in I} P_i^{\oplus}} \qquad \text{T-}\mathbf{0}\ \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash_D \mathbf{0}}$$

Figure 3.4: Typing rules for the core calculus.

Type Checking

Figure 3.4 shows the typing rules for the core calculus. There are three typing judgements: *(i)* the *value judgement* $\Gamma \vdash V : T$ assigns type T to value V under context Γ; *(ii)* the *program judgement* $\vdash \mathscr{P}$ checks whether a program is self-enclosed; and *(iii)* the *process judgement* $\Gamma \vdash_D P$ checks whether a process P is well-typed under a context Γ and process definitions D.

The value judgement has three rules. Rule T-S types variables and role indexed sessions to static types, allowing for subtyping. Rule T-Wkn allows weakening, i.e., if value V has type T under some environment Γ_1, and there exists an end-typed environment Γ_2 such that $\Gamma_1 + \Gamma_2 = \Gamma$, then the rule allows the value to be typed under the larger context Γ. Rule T-B types a basic value v to type B iff v belongs to B.

The program judgement is only used by rule T-$\mathscr{P}$ to type a program (P, D). This judgement does not use a type environment, and ensures that programs are *closed*. This is achieved by requiring the main process P to be well-typed under an empty context. Furthermore, process definitions are also required to be closed by typing them under a context containing only the arguments defined for that process name.

The final judgement types processes under type context Γ.

Rule T-ν is what makes the type system *generalised*. Instead of forcing a specific property on types, session restriction requires that the defined protocol adheres to the minimum requirements for subject reduction. Specifically, the rule requires proof of $\varphi(\mathtt{assoc}_s(\Psi))$, where φ is any *safety* property—the *largest* safety property will be defined later. If the session protocol adheres to these requirements, and the session has not been previously established, then the restricted process can be typed using the newly added context. Rule T-X checks that the type of every value used in a process call match the expected types defined in the process definition D.

Rule T-$\oplus$ types a send process. First, the channel used for sending should be typed to a selection session type with a matching role and message label (it is key to note that this accounts for subtyping). Then, the values sent as the message payload should be typed to the types specified in the selection session type, under some split context. Lastly, the continuation process should be well typed under the remainder of the original context, with the channel updated to the continuation session type. Dually, rule T-& types the receive process by checking that the channel used to receive can be mapped to a branching session type with matching role and branch labels (also accounting for subtyping). Then, every continuation process should be typed using the respective branch from the continuation session type, along with any new binders specified in the payload.

Rule T-| states that process composition is well typed under a context if that context can be split into two sub-environments that type each of the composed processes. Rule T-+ requires all processes in a nondeterministic choice to be typed under the same environment. Lastly, a terminated process is typed by rule T-**0** iff any remaining session types in the type environment are *end-typed*.

3.2.3 Semantics

The core calculus is enriched with operational semantics to model reduction via communication.

> **Definition 3.8** (Process reduction). A process P reduces parametric on some process definitions D, written $P \rightarrow_D P'$, via the operational semantics defined in Figure 3.5. Multistep reductions are expressed using the reflexive and transitive closure, i.e., $P \rightarrow_D^* P'$.

Definition 3.8 defines process reduction ($\rightarrow_D$) up-to structural congruence ($\equiv$), both given in Figure 3.5. Structural congruence for the core calculus is similar to what is standard for the π-calculus. Parallel composition is commutative and associative with identity element **0**; session restriction is commutative and has absorbing element **0**; and scope extrusion allows restrictions to move in/out of parallel compositions provided there are no naming collision.

The main reduction rule for processes is rule R-C; it models *communication* between two roles. Concretely, given a sending process $s[\boldsymbol{p}][\boldsymbol{q}] \oplus \mathsf{m}\langle\vec{d}\rangle \,.\, P$ in parallel with a receiving process

Structural Congruence $\boxed{P \equiv Q}$

$$P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \qquad P \mid \mathbf{0} \equiv P \qquad (\nu s)\mathbf{0} \equiv \mathbf{0}$$

$$(\nu s_1)(\nu s_2)P \equiv (\nu s_2)(\nu s_1)P \qquad \frac{s \notin \mathsf{fs}(Q)}{(\nu s)(P \mid Q) \equiv (\nu s)P \mid Q}$$

Process reduction $\boxed{P \to_D P'}$

$$\text{R-C}\quad s[p][q] \oplus \mathsf{m}_k\langle \vec{d} \rangle \,.\, P \mid s[q][p] \,\&_{i \in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, Q_i \;\to_D\; P \mid Q_k\{\vec{d}/\vec{b}_k\} \quad \text{if } k \in I$$

$$\text{R-+}\quad \frac{j \in I}{\sum_{i \in I} P_i^{\oplus} \;\to_D\; P_j^{\oplus}} \qquad \text{R-X}\quad \frac{D(\mathrm{X}) = (\vec{x}, \vec{T}, P)}{\mathrm{X}\langle \vec{d} \rangle \;\to_D\; P\{\vec{d}/\vec{x}\}} \qquad \text{R-}\mid\quad \frac{P \to_D P'}{P \mid Q \to_D P' \mid Q}$$

$$\text{R-}\nu\quad \frac{P \to_D P'}{(\nu s)P \;\to_D\; (\nu s)P'} \qquad \text{R-}\equiv\quad \frac{P \equiv P' \qquad P \;\to_D\; Q \qquad Q \equiv Q'}{P' \;\to_D\; Q'}$$

Figure 3.5: Operational semantics for the core calculus.

$s[q][p]\ \&_{i \in I}\ \mathsf{m}_i(\vec{b}_i)\ .\ Q_i$, where both processes operate on the same session (s), have matching roles (p and q), and the sender message label is within the expected labels on the receiver side ($\exists k \in I$ s.t. $\mathsf{m} = \mathsf{m}_k$), then both processes reduce to their continuations (P and Q_k resp.). The receiver also substitutes the binding payloads in its continuation with the concrete values in the sender payload. Rule R-+ non-deterministically reduces a choice of sending processes to any one of its continuations. The call rule, rule R-X, models recursion by looking up the called process name in the parametrised definitions D, reducing to the mapped process and substituting any passed arguments. The remaining rules are administrative. Rules R-$\mid$ and R-ν allow processes to reduce under parallel composition and session restriction respectively. Lastly, rule R-$\equiv$ allows for reduction modulo structural congruence.

Example 3.9 (Load Balancer: Reduction). The following shows step by step reductions on the main process for load balancer $\mathscr{P}_{lb}$. The initial process is identical to that specified in the program (cf. Example 3.1) sans session restriction, which is omitted for simplicity.

$$\mathrm{Srv}\langle s[srv] \rangle \mid \mathrm{Wrk}\langle s[w_1] \rangle \mid \mathrm{Wrk}\langle s[w_2] \rangle$$

$$\mid s[c][srv] \oplus \mathsf{req}\langle 42 \rangle \,.\, s[c][srv]\ \& \begin{cases} \mathsf{worker1}() \,.\, s[c][w_1]\ \&\ \mathsf{ans}(z) \,.\, \mathbf{0} \\ \mathsf{worker2}() \,.\, s[c][w_2]\ \&\ \mathsf{ans}(z) \,.\, \mathbf{0} \end{cases}$$

The only immediately available reductions are via rule R-X. A possible trace is one where the server process unrolls first, substituting in the channel $s[\mathit{srv}]$.

$$\rightarrow_D$$

$$s[\mathit{srv}][c] \,\&\, \mathsf{req}(x) \,.\, \left(\begin{array}{c} s[\mathit{srv}][w_1] \oplus \mathsf{fw}\langle x \rangle \,.\, s[\mathit{srv}][c] \oplus \mathsf{worker1}\langle\rangle \,.\, \mathbf{0} \\ + \\ s[\mathit{srv}][w_2] \oplus \mathsf{fw}\langle x \rangle \,.\, s[\mathit{srv}][c] \oplus \mathsf{worker2}\langle\rangle \,.\, \mathbf{0} \end{array} \right) \mid \mathrm{Wrk}\langle s[w_1] \rangle$$
$$\mid \mathrm{Wrk}\langle s[w_2] \rangle \mid s[c][\mathit{srv}] \oplus \mathsf{req}\langle 42 \rangle \,.\, s[c][\mathit{srv}] \,\&\, \left\{ \begin{array}{l} \mathsf{worker1}() \,.\, s[c][w_1] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \\ \mathsf{worker2}() \,.\, s[c][w_2] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \end{array} \right.$$

Communication is now possible between the client and the server, and thus the process may reduce by rule R-C. It is key to note that communication substitutes the sent payload in the continuation of the receiver process.

$$\rightarrow_D$$

$$\left(\begin{array}{c} s[\mathit{srv}][w_1] \oplus \mathsf{fw}\langle 42 \rangle \,.\, s[\mathit{srv}][c] \oplus \mathsf{worker1}\langle\rangle \,.\, \mathbf{0} \\ + \\ s[\mathit{srv}][w_2] \oplus \mathsf{fw}\langle 42 \rangle \,.\, s[\mathit{srv}][c] \oplus \mathsf{worker2}\langle\rangle \,.\, \mathbf{0} \end{array} \right) \mid \mathrm{Wrk}\langle s[w_1] \rangle \mid \mathrm{Wrk}\langle s[w_2] \rangle$$
$$\mid s[c][\mathit{srv}] \,\&\, \left\{ \begin{array}{l} \mathsf{worker1}() \,.\, s[c][w_1] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \\ \mathsf{worker2}() \,.\, s[c][w_2] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \end{array} \right.$$

The server must now non-deterministically choose between sending to either of the works. Consider that the branch for worker w_1 is chosen.

$$\rightarrow_D^* \; s[\mathit{srv}][w_1] \oplus \mathsf{fw}\langle 42 \rangle \,.\, s[\mathit{srv}][c] \oplus \mathsf{worker1}\langle\rangle \,.\, \mathbf{0}$$
$$\mid s[w_1][\mathit{srv}] \,\&\, \mathsf{fw}(y) \,.\, s[w_1][c] \oplus \mathsf{ans}\langle f(y) \rangle \,.\, \mathbf{0}$$
$$\mid \mathrm{Wrk}\langle s[w_2] \rangle \mid s[c][\mathit{srv}] \,\&\, \left\{ \begin{array}{l} \mathsf{worker1}() \,.\, s[c][w_1] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \\ \mathsf{worker2}() \,.\, s[c][w_2] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \end{array} \right.$$

Communication can now continue between the participants until each role (except for w_2) reaches the empty process.

$$\rightarrow_D^* \; \mathbf{0} \mid \mathbf{0} \mid s[w_2][\mathit{srv}] \,\&\, \mathsf{fw}(y) \,.\, s[w_2][c] \oplus \mathsf{ans}\langle f(y) \rangle \,.\, \mathbf{0} \mid \mathbf{0}$$
$$\equiv \; s[w_2][\mathit{srv}] \,\&\, \mathsf{fw}(y) \,.\, s[w_2][c] \oplus \mathsf{ans}\langle f(y) \rangle \,.\, \mathbf{0}$$

At this stage, there are no further possible reductions. Such a program is said to be *deadlocked*, since it reaches a process with no further reductions that is not $\mathbf{0}$. MPST aim to catch such errors at compile time, by statically checking for properties such as

Type LTS $\boxed{\Gamma \xrightarrow{A} \Gamma'}$

$$\Gamma\text{-}\& \quad \frac{S = \&_{i\in I}\boldsymbol{q}{:}\mathsf{m}_i(\vec{T}_i).S'_i \quad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S'_k} \qquad \Gamma\text{-}\oplus \quad \frac{S = \oplus_{i\in I}\boldsymbol{q}_i{:}\mathsf{m}_i(\vec{T}_i).S'_i \quad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}_k:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S'_k} \qquad \Gamma\text{-}\mu \quad \frac{\Gamma \cdot c : S\{\mu \mathrm{t}.S/\mathrm{t}\} \xrightarrow{A} \Gamma'}{\Gamma \cdot c : \mu \mathrm{t}.S \xrightarrow{A} \Gamma'}$$

$$\Gamma\text{-Com} \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \vec{T} \leqslant \vec{T}' \qquad \Gamma_1 \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \Gamma'_1 \qquad \Gamma_2 \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}(\vec{T}')} \Gamma'_2}{\Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \Gamma'_1 + \Gamma'_2} \qquad \Gamma\text{-Cong}_1 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma, \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''} \qquad \Gamma\text{-Cong}_2 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma \cdot \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''}$$

Figure 3.6: Type semantics.

deadlock-freedom.

Integral to the generalised approach to MPST, type context Γ is enriched with a labelled transition system (LTS) aiming to simulate the communication protocol of a program. This LTS is what will be used to define type-level properties and allow for protocol verification.

Definition 3.10 (Context reduction). Labels for the type LTS are given by actions A, representing *output*, *input*, and *communication*.

$$\text{Actions} \quad A ::= s{:}\boldsymbol{p}\oplus\boldsymbol{q}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{p},\boldsymbol{q}{:}\mathsf{m}$$

The roles involved in an action are referred to by the function $\mathtt{roles}(A)$, defined as $\mathtt{roles}(s{:}\boldsymbol{p}\oplus\boldsymbol{q}{:}\mathsf{m}(\vec{T})) = \mathtt{roles}(s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m}(\vec{T})) = \mathtt{roles}(s{:}\boldsymbol{p},\boldsymbol{q}{:}\mathsf{m}) = \{\boldsymbol{p},\boldsymbol{q}\}$.

The type LTS is defined by the transitions listed in Figure 3.6. Context *reduction*, $\Gamma \to \Gamma'$, is defined iff $\Gamma \xrightarrow{A} \Gamma'$ where $A = s{:}\boldsymbol{p},\boldsymbol{q}{:}\mathsf{m}$ for any $s, \boldsymbol{p}, \boldsymbol{q}, \mathsf{m}$—i.e., a context reduces iff it can transition via a communication action. The transitive and reflexive closure of context reduction is written as $\to^*$; and $\Gamma \to$ represents the existence of some transition from Γ.

Definition 3.10 begins by listing the labels used in the type LTS: (*i*) *output* ($s{:}\boldsymbol{p}\oplus\boldsymbol{q}{:}\mathsf{m}(\vec{T})$) on session s from $\boldsymbol{p}$ to $\boldsymbol{q}$ of message m with payloads $\vec{T}$; (*ii*) *input* ($s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m}(\vec{T})$) on session s at $\boldsymbol{p}$ from $\boldsymbol{q}$ of message m with payloads $\vec{T}$; and (*iii*) *communication* ($s{:}\boldsymbol{p},\boldsymbol{q}{:}\mathsf{m}$) on session s of message m between $\boldsymbol{p}$ (the sender) and $\boldsymbol{q}$ (the receiver).

The LTS itself is given in Figure 3.6. As expected, using rule Γ-& (resp. rule Γ-$\oplus$), a branching session type (resp. selection) can transition via an input (resp. output) label to reach any one of its continuation types. Importantly, the session, roles, message labels and payload

types of the action correspond to what is dictated in type. Rule Γ-Com models communication—if a context can be split in a way such that one partition fires an output action, and the other fires an input action with corresponding roles, message label and super-typed payloads, then the entire context can transition via a communication action to the addition of each respective reductum. The remaining rules allow for transitions modulo context composition, context splitting, and recursive unfolding.

Example 3.11 (Load Balancer: Type Reduction). By examining the LTS generated from the load balancer protocol in Example 3.2, the program can be determined to be *deadlocked* without requiring process execution. Consider the initial context used for type checking.

$$s[\mathit{srv}] : \&\mathit{c}{:}\mathsf{req}(\mathtt{Int}).\oplus \begin{cases} \mathit{w}_1{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{worker1}().\mathbf{end} \\ \mathit{w}_2{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{worker2}().\mathbf{end} \end{cases},$$

$$s[\mathit{w}_1] : \&\mathit{srv}{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{ans}(\mathtt{String}).\mathbf{end},$$

$$s[\mathit{w}_2] : \&\mathit{srv}{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{ans}(\mathtt{String}).\mathbf{end},$$

$$s[\mathit{c}] : \oplus\mathit{srv}{:}\mathsf{req}(\mathtt{Int}).\&\mathit{srv}{:} \begin{cases} \mathsf{worker1}().\&\mathit{w}_1{:}\mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \mathsf{worker2}().\&\mathit{w}_2{:}\mathsf{ans}(\mathtt{String}).\mathbf{end} \end{cases}$$

There is only one reduction for this context, via the communication action $s{:}\mathit{c},\mathit{srv}{:}\mathsf{req}$.

$$\rightarrow$$

$$s[\mathit{srv}] : \oplus \begin{cases} \mathit{w}_1{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{worker1}().\mathbf{end} \\ \mathit{w}_2{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{worker2}().\mathbf{end} \end{cases},$$

$$s[\mathit{w}_1] : \&\mathit{srv}{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{ans}(\mathtt{String}).\mathbf{end},$$

$$s[\mathit{w}_2] : \&\mathit{srv}{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathit{c}{:}\mathsf{ans}(\mathtt{String}).\mathbf{end},$$

$$s[\mathit{c}] : \&\mathit{srv}{:} \begin{cases} \mathsf{worker1}().\&\mathit{w}_1{:}\mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \mathsf{worker2}().\&\mathit{w}_2{:}\mathsf{ans}(\mathtt{String}).\mathbf{end} \end{cases}$$

The LTS now branches into two paths, one for each possible communication action (i.e.,

s:srv,w_1:fw and s:srv,w_2:fw). Consider the path proceeding from the former.

$\rightarrow$

$s[\mathit{srv}] : \oplus c$:worker1().**end**, $s[w_1] : \oplus c$:ans(String).**end**,

$s[w_2] : \&\mathit{srv}$:fw(Int).$\oplus c$:ans(String).**end**,

$s[c] : \&\mathit{srv}$: $\begin{cases} \text{worker1}().\&w_1\text{:ans(String).}\mathbf{end} \\ \text{worker2}().\&w_2\text{:ans(String).}\mathbf{end} \end{cases}$

This context proceeds via deterministic reductions, reaching the following.

$\rightarrow^*$

$s[\mathit{srv}] : \mathbf{end}$, $s[c] : \mathbf{end}$, $s[w_1] : \mathbf{end}$, $s[w_2] : \&\mathit{srv}$:fw(Int).$\oplus c$:ans(String).**end**

As will be shown in the next section (cf. Proposition 3.18), this context is guaranteed to type processes that are deadlocked, therefore load balancer $\mathscr{P}_{lb}$ can be determined to be deadlocked directly from its protocol.

3.3 Metatheory

Unlike traditional MPST theory, the *generalised* approach does not syntactically guarantee any properties on typed processes. Rather, it serves as a framework for verifying customisable runtime properties. This is accomplished by verifying properties on the type LTS for a given protocol, from which process-level properties may be inferred. For this to be possible, the metatheory of a generalised MPST system should provide two guarantees—*subject reduction* and *session fidelity*. The former guarantees that process reduction preserves typing, whilst the latter guarantees that processes follow at least one path of the type LTS. Both theorems together allow for runtime properties to be inferred from contexts, since no communication will occur at runtime that is not modelled by the type semantics. Furthermore, the theorems should hold for some largest set of contexts which can be described with a property φ. (This is the requirement made for typechecking in rule T-v.) Essentially, φ should describe the minimum (and most general) requirements for *subject reduction* or *session fidelity* to hold. By proving these theorems parametric on some largest φ, the property can be re-instantiated with more specific properties without having to reprove any of the base metatheory. (In the core calculus, φ should be instantiated with a *safety* property to guarantee subject reduction, and a *fidelity* property for session fidelity.) This is what makes the type system *generalised*, allowing for verification of redefinable properties as long as they conform to the minimum requirements.

safe	$\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge\ \Gamma \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}'(\vec{T}')} \implies \Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}}$
fidelitous	$\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge\ \Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{r}:\mathsf{m}'} \implies \forall \vec{A} : \Gamma \xrightarrow{\vec{A}}{}^{*}\, \Gamma' \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \wedge\ \boldsymbol{p} \notin \mathtt{roles}(\vec{A})$
deadlock-free	$\Gamma \to^{*} \Gamma' \not\to \implies \mathsf{end}(\Gamma')$
terminating	Γ is deadlock free and all paths from Γ are terminating
never-terminating	$\Gamma \to^{*} \Gamma' \implies \Gamma' \to$
lock-free	$\Gamma \to^{*} \Gamma' \implies$ all fair paths from Γ' are also live
live	$\Gamma \to^{*} \Gamma' \implies$ all paths from Γ' are live

Table 3.1: Properties of a type context Γ.

3.3.1 Type Context Properties

Before presenting the main theorems of the calculus, some auxiliary definitions are required. The following formalises the notion of a *path*—i.e., a sequence of transitions of a context—and defines properties on *type contexts*. The main theorems are then presented, and are subsequently used to determine runtime properties on programs directly from the properties defined on types.

Definition 3.12 (Path). A type context *path* is a possibly infinite sequence of type contexts, $\vec{\Gamma}$, s.t. $\forall \Gamma_i \in \vec{\Gamma} : \Gamma_i \to \Gamma_{i+1}$. A path $\vec{\Gamma}$ is said to be:

1. *terminating* iff $\vec{\Gamma}$ is finite;
2. *fair* iff $\forall \Gamma_i \in \vec{\Gamma} : \Gamma_i \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \implies \exists \Gamma_k \in \vec{\Gamma}, \boldsymbol{r}, \mathsf{m}' : k \geq i \wedge \Gamma_k \xrightarrow{s:\boldsymbol{p},\boldsymbol{r}:\mathsf{m}'} \Gamma_{k+1}$
3. *live* iff $\forall \Gamma_i \in \vec{\Gamma}$:
   (a) $\Gamma_i \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \implies \exists \Gamma_k \in \vec{\Gamma}, \boldsymbol{r}, \mathsf{m}' : k \geq i \wedge \Gamma_k \xrightarrow{s:\boldsymbol{p},\boldsymbol{r}:\mathsf{m}'} \Gamma_{k+1}$; and
   (b) $\Gamma_i \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}(\vec{T})} \implies \exists \Gamma_k \in \vec{\Gamma}, \mathsf{m}' : k \geq i \wedge \Gamma_k \xrightarrow{s:\boldsymbol{q},\boldsymbol{p}:\mathsf{m}'} \Gamma_{k+1}$.

A context *path* (Definition 3.12) is a possibly infinite sequence of context reductions. A path is said to be (*i*) *terminating* iff it is finite; (*ii*) *fair* iff at any point on the path, if a communication action is possible for some sender role, then there exists a future on that path where the context reduces via communication stemming from the same sender; and (*iii*) *live* iff at any point on the path, if a sender (resp. receiver) is blocked, then there exists a future on that path where that role becomes unblocked. Table 3.1 uses paths to define properties on type contexts. Each property can be informally described as follows.

Safe If a message m can be output from $\boldsymbol{p}$ to $\boldsymbol{q}$, and an input can occur at $\boldsymbol{q}$ from $\boldsymbol{p}$, then m should be able to be communicated.

Fidelitous If a message can be output from $\boldsymbol{p}$, and communication stemming from $\boldsymbol{p}$ is possible, then every message that can be output should be always eventually capable of communication (possibly after some reductions not involving $\boldsymbol{p}$).

Deadlock-free If a context can no longer reduce, then it must be end-typed.

Terminating The context is deadlock-free and all of its paths are finite.

Never-terminating The context can always reduce.

Lock-free Under a fair scheduler, all outputs and inputs eventually become unblocked.

Live All outputs and inputs eventually become unblocked, regardless of the scheduler.

Note on the fidelitous property. A fidelitous property is stronger than one that is safe, as it requires the communication of *all* messages in a selection type irrespective of the destination role. Essentially, if at least one destination role in a selection can be reached, then all of them should be able to be reached. This is a novel property that is only needed because of undirected selection. (Details of why it is needed are discussed later, cf. Section 3.3.3.) The chosen name for the property "*fidelitous*", an old English word, is an adjective form of the noun fidelity—which in turn is a noun form of the adjective faithful. Since the words "faithful" and "fidelity" already have defined meanings in this area of research, this old word was chosen to distinguish the novel property whilst still nodding at its connections to fidelity.

3.3.2 Subject Reduction

Subject reduction (Theorem 3.15) informally states that, if a well-typed program can reduce, then its reductums are also well-typed. As previously alluded to, this implication depends on a *safety property*. Definition 3.13 formalises the *largest* safety property for which subject reduction is dependent upon.

Definition 3.13 (Safety). φ is a *safety* property on type environment Γ iff:
φ**-S** $\varphi(\Gamma)$ implies Γ is *safe*;
φ**-μ** $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu \mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu \mathrm{t}.S/\mathrm{t}\})$;
φ**-$\rightarrow$** $\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$.
$\mathsf{safe}(\Gamma)$ is written iff $\varphi(\Gamma)$ and φ is a safety property.

A property φ is a safety property on type contexts iff it guarantees that all reductions, and their unfoldings, of the context are *safe*—referring to the type context property in table 3.1. This essentially enforces programs to not contain unexpected messages, and further requires sent payloads to be subtypes of expected payloads; since a *safe* context implies the capability of communication, and by rule Γ-Com payloads must correspond in order to communicate.

Lemma 3.14. For all contexts Γ_1, Γ_2:

1. if $\mathsf{safe}(\Gamma_1 \cdot \Gamma_2)$, then $\mathsf{safe}(\Gamma_1)$;
2. if $\mathsf{safe}(\Gamma_1)$ and $\Gamma_1 \leqslant \Gamma_2$, then $\mathsf{safe}(\Gamma_2)$;
3. if $\mathsf{safe}(\Gamma_1)$ and $\exists \Gamma_2' : \Gamma_1 \leqslant \Gamma_2 \to \Gamma_2'$, then $\exists \Gamma_1' : \Gamma_1 \to \Gamma_1' \leqslant \Gamma_2'$.

Proof.

1. By contradiction. Assume Γ_1 is *not* safe, then there must exist some Γ' such that $\Gamma_1 \to^* \Gamma''$ and $\Gamma' = \mathsf{unf}^*(\Gamma'')$ and Γ' violates φ-S. But by rule Γ-Cong_2 $\Gamma_2 \cdot \Gamma_1 \to^* \Gamma_2 + \Gamma''$ and after some possible applications of rule Γ-μ, $\Gamma_2 + \Gamma'$ violates φ-S. Hence, $\Gamma_2 + \Gamma'$ is not safe, meaning $\Gamma_2 + \Gamma''$ is not safe via φ-μ, and thus $\Gamma_2 + \Gamma_1$ is not safe by φ-$\to$. By Proposition 3.5 and commutativity of $\cdot$, $\Gamma_1 \cdot \Gamma_2$ is not safe. This contradicts the thesis. □

2. By coinduction on $\mathsf{safe}(\Gamma_1)$. Observe that by $\mathsf{safe}(\Gamma_1)$, Γ_1 is safe. Thus, if $\Gamma_1 \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})}$ and $\Gamma_1 \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}'(\vec{T}')}$ then $\Gamma_1 \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}}$. Assume communication to be enabled on Γ_1, and without loss of generality, assume the above output and input actions to also be enabled. By rule Γ-$\oplus$ and rule Γ-&, the shape of Γ_1 can be inferred to be:

$$\Gamma_1 = \Gamma_1', s[\boldsymbol{p}] : S^{\oplus}, s[\boldsymbol{q}] : S^{\&}$$
$$S^{\oplus} = \oplus_{i \in I} \boldsymbol{r_i}:\mathsf{m}_i(\vec{T}_i).S_i \quad \text{for some } I$$
$$\exists I' \subseteq I \text{ such that } \forall i \in I' : \boldsymbol{r_i} = \boldsymbol{q}$$
$$S^{\&} = \&_{j \in J} \boldsymbol{p}:\mathsf{m}_j'(\vec{T}_j').S_j'$$

By φ-S there are no unexpected messages in $S^{\oplus}$. Consider now a $S^{\oplus\prime} \geqslant S^{\oplus}$ and $S^{\&\prime} \geqslant S^{\&}$. By the definition of $\leqslant$, $S^{\oplus\prime}$ has (at most) less internal choices, and $S^{\&\prime}$ has (at most) more external choices. Therefore, there are still no unexpected messages in $S^{\oplus\prime}$. Hence, given $\mathsf{safe}(\Gamma_1)$ and a $\Gamma_2 \geqslant \Gamma_1$, it follows that φ-S holds for Γ_2. Furthermore, by rule Γ-μ, no new transitions are introduced via recursive binders, hence φ-μ must also hold for Γ_2. Lastly, by the coinductive hypothesis, safe holds for contexts containing the continuation types of elements in Γ_1 and Γ_2—these contexts are uniquely contained within the context reduction relation; therefore φ-$\to$ holds for Γ_2. □

3. Since $\Gamma_2 \to$, then by rule Γ-Com:

$$\Gamma_2 = \Gamma_2'', s[\boldsymbol{p}] : S^{\oplus\prime}, s[\boldsymbol{q}] : S^{\&\prime}$$
$$S^{\oplus\prime} = \oplus_{i \in I'} \boldsymbol{r_i}:\mathsf{m}_i''(\vec{T}_i'').S_i''$$
$$S^{\&\prime} = \&_{j \in J'} \boldsymbol{p}:\mathsf{m}_j'''(\vec{T}_j''').S_j'''$$
$$\exists K' \subseteq I' \text{ such that } \forall i \in K' : \boldsymbol{r_i} = \boldsymbol{q}$$
$$K' \subseteq J' \text{ and } \forall i \in K' : \vec{T}_i'' \leqslant \vec{T}_i'''$$

Via subtyping, the shape of Γ_1 can be inferred (since $\Gamma_1 \leqslant \Gamma_2$).

$$\Gamma_1 = \Gamma_1'', s[\boldsymbol{p}] : S^{\oplus}, s[\boldsymbol{q}] : S^{\&}$$
$$S^{\oplus} = \oplus_{i \in I} \boldsymbol{r_i}{:}\mathsf{m}_i(\vec{T}_i).S_i$$
$$S^{\&} = \&_{j \in J} \boldsymbol{p}{:}\mathsf{m}'_j(\vec{T}'_j).S'_j$$
$$\exists K \subseteq I \text{ such that } \forall i \in K : \boldsymbol{r_i} = \boldsymbol{q}$$
$$K \subseteq J \text{ and } \forall i \in K : \vec{T}_i \leqslant \vec{T}'_i$$

By the definition of $\leqslant$, it follows that $K \supseteq K'$ and therefore $K' \subseteq J$. Hence, since $\mathsf{safe}(\Gamma_1)$, $\forall i \in K' : \vec{T}''_i \leqslant \vec{T}'_i$; therefore Γ_1 can reduce, at the very least, by the transitions enabled on the supertype. That is, $\exists \Gamma'_1 : \Gamma_1 \rightarrow \Gamma'_1$. Furthermore, from Definition 3.3, type continuations preserve subtyping. Therefore $\Gamma'_1 \leqslant \Gamma'_2$. □

Lemma 3.14 identifies propositions on safe contexts that are key to the metatheory. Essentially, the propositions state that:

1. context splits (parallel composition in processes) preserve safety;
2. supertyping preserves safety; and
3. given safe contexts, reduction commutes over subtyping.

The first main result of the core calculus can now be formalised as the subject reduction theorem below. [1]

Theorem 3.15 (Subject Reduction). If $\Gamma \vdash_D P$ with $\mathsf{safe}(\Gamma)$ and $P \rightarrow_D P'$, then $\exists \Gamma'$ s.t. $\Gamma \rightarrow^* \Gamma'$ and $\Gamma' \vdash_D P'$ with $\mathsf{safe}(\Gamma')$.

Proof. (Sketch.) By induction on the derivation of $P \rightarrow_D P'$. By case analysis of the process reduction rules, the shapes of P and P' can be determined. Given $\Gamma \vdash_D P$, the shape of Γ can be inferred by inversion of the typing rules. Then, by $\mathsf{safe}(\Gamma)$, Γ is observed to mimic any communication actions performed by the process; otherwise the same context can be shown to type the reduced process (for cases such as R-+). □

Subject reduction guarantees that well-typed processes remain well-typed after reduction. Furthermore, a process being well-typed itself guarantees (via the safety condition in rule T-ν and φ-S) that no unexpected messages will ever be manifested at runtime. Unexpected messages are formalised as erroneous processes in Definition 3.16, and subsequently shown to be absent from any well-typed program.

[1] The next chapter proves subject reduction and session fidelity for a conservative extension of the core calculus. For this reason, a full proof is not presented here, as it will be subsumed.

Definition 3.16 (Erroneous process). It is said that P is *erroneous* iff:

1. $P \equiv \widetilde{(\nu s)}\, s[\mathbf{p}][\mathbf{q}] \oplus \mathsf{m}_k\langle \vec{V} \rangle \,.\, Q \mid s[\mathbf{q}][\mathbf{p}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, Q'_i \mid Q''$; and
2. either $k \notin I$ or $|\vec{V}| \neq |\vec{b}_k|$.

Corollary 3.17 (Type safety). Given a well-typed program $\vdash \mathscr{P} = (P, D)$, if $P \to_D^* P'$ then P' is *not erroneous*.

Proof. From rule T-$\mathscr{P}$, it follows that $\emptyset \vdash_D P$ and $\emptyset$ is safe. Using Theorem 3.15 and induction over $\to_D^*$, $\exists \Gamma'$ s.t. $\Gamma' \vdash_D P'$ and $\mathsf{safe}(\Gamma')$. Continuing the proof by contradiction, assume P' is erroneous. Since $\Gamma' \vdash_D P'$ and by Definition 3.16, $\Gamma' \xrightarrow{s:\mathbf{p}\oplus\mathbf{q}:m_k(\vec{T}_k)}$, $\Gamma'(s[\mathbf{q}]) \leqslant \&_{i\in I}\, \mathbf{p}:\mathsf{m}_i(\vec{T}'_i).S_i$, and either $k \notin I$ or $|\vec{T}_k| \neq |\vec{T}'_k|$. But both of these latter statements imply that $\Gamma' \xrightarrow{s:\mathbf{p},\mathbf{q}:\mathsf{m}_k} \not\longrightarrow$ (by rule Γ-Com). This violates φ-S—contradiction. □

Proposition 3.18 (Deadlocked Processes). Via the contrapositive of subject reduction (Theorem 3.15), it is possible to statically determine processes as *deadlocked*. For instance, recall Example 3.11, the final context

$$s[\mathbf{srv}] : \mathbf{end},\quad s[\mathbf{c}] : \mathbf{end},\quad s[\mathbf{w}_1] : \mathbf{end},\quad s[\mathbf{w}_2] : \&\mathbf{srv}{:}\mathsf{fw}(\mathtt{Int}).\oplus \mathbf{c}{:}\mathsf{ans}(\mathtt{String}).\mathbf{end}$$

implies that the process it types is deadlocked since: 1. the context types a non-end process P (by inversion of typing rules); and 2. by contrapositive of subject reduction, $P \not\to P'$.

3.3.3 Session Fidelity

Informally, *session fidelity* states that if a context can take a step, then there should exist at least one transition on the context that can be matched by the processes. In other words, the process is faithful to the session type, and is guaranteed to follow one of the protocol's paths at runtime. This is a strong property—as is standard in the literature [103, 118, 51, 48], assumptions on programs are necessary in order to guarantee session fidelity in the core calculus. The assumptions are that processes *(i)* are productive (i.e., cannot infinitely reduce without communicating), *(ii)* do not incorrectly interleave multiple sessions, and *(iii)* do not play the wrong roles at the wrong times. Each of these issues may result in a program being deadlocked in spite of sessions being deadlock free. The following formally defines restrictions on the calculus that guarantee these assumptions. Session fidelity is then presented and proved, dependent on these restrictions.

$$\begin{array}{ll} \textit{Processes} & P,Q ::= P|Q \mid \sum_{i\in I} P_i^{\oplus} \mid P^{\&} \mid \mathrm{X}\langle \vec{V} \rangle \mid \mathbf{0} \\ \textit{Programs} & \mathscr{P} ::= ((\nu s : \Psi)\, P, D) \end{array}$$

Figure 3.7: Session Fidelity Restrictions on Core Calculus Syntax

Definition 3.19 (Guarded definitions). Process definitions D are said to be *guarded* iff for all mappings of the form $\mathrm{X} \mapsto (\vec{x}, \vec{T}, P)$ in D, if $\mathrm{Y}\langle \vec{V} \rangle$ is a subterm of P, then: $\exists i \in 1..|\vec{x}|$ s.t. $T_i \leqslant S \not\leqslant \mathbf{end}$ and $x_i \in \vec{V}$ implies $\mathrm{Y}\langle \vec{V} \rangle$ is guarded by a communication action on x_i in P.

To ensure processes do not infinitely reduce without communicating, Definition 3.19 requires all recursive calls involving channels as arguments to be guarded by at least one action for every channel. This guarantees processes will be productive.

Definition 3.20 (Plays one role). Assume programs to be constructed via the grammar in Figure 3.7. Then, given the judgement $\Gamma \vdash_D P$, it is said that P *only plays role* $\boldsymbol{p}$ *by* Γ iff: *(i)* D is guarded; *(ii)* $\mathsf{fv}(P) = \emptyset$; *(iii)* $\Gamma = \Gamma', s[\boldsymbol{p}] : S$ with $S \not\leqslant \mathbf{end}$ and $\mathsf{end}(\Gamma')$.

Furthermore, it is said that:

1. P *only plays role* $\boldsymbol{p}$ iff $\exists \Gamma, D$ s.t. $\Gamma \vdash_D P$ and the above conditions hold; and
2. $\mathsf{one\text{-}role}(P)$ iff $P \equiv \Big|_{\boldsymbol{p}\in I} P_{\boldsymbol{p}}$ where each $P_{\boldsymbol{p}}$ is either $\mathbf{0}$ (up-to $\equiv$), or only plays role $\boldsymbol{p}$.

The next assumption is that processes cannot deadlock (for reasons other than a deadlocked protocol). To ensure this, the core calculus is first restricted to the grammar in Figure 3.7, requiring programs to only contain a single multiparty session. This prevents programs from getting stuck due to incorrect interleaving of multiple sessions. Furthermore, using the restricted grammar, processes could still deadlock by playing the wrong role at the wrong time. To circumvent this issue, Definition 3.20 describes processes that can only play a single role within the program's single session. The key point to the definition is item *(iii)*, stating that the process must be typed under a context containing a singular (non-end) session type. Then, $\mathsf{one\text{-}role}(P)$ is used to describe an ensemble of such processes in parallel, each playing their own distinct role in the session. These restrictions suffice for session fidelity.

It is key to observe that, as is standard, these restrictions prevent *delegation* within the calculus. Allowing for delegation (without any restrictions on its uses) could deadlock a program by incorrectly using the wrong session at the wrong time, even if the individual session protocols are deadlock free. This metatheory instead focuses on providing a framework for verifying properties on ensembles of processes each playing a unique role within a single session.

Different from the standard formalism [103], but similar to work more based in practice [51], session fidelity (Theorem 3.22) requires a property stronger than safety. This is because the

presented calculus makes use of undirected selection. (An example context which is safe but violates session fidelity is given in Example 3.23.) Instead, the theorem is proved parametric on a *fidelity* property. Definition 3.21 describes the largest fidelity property, capturing all fidelitous contexts. A brief point of novelty to the core calculus is the identification of this fidelity property; generalised MPST systems typically tend to use directed choice (in which case a safety property suffices), or use undirected selection but require a stronger property than that presented in Definition 3.21. For instance, Harvey et al. [51] require a property akin to lock-freedom.

Definition 3.21 (Fidelity). φ is a fidelity property on type environment Γ iff:

(i) $\varphi(\Gamma)$ implies Γ is *fidelitous*;

(ii) $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu \mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu \mathrm{t}.S/\mathrm{t}\})$;

(iii) $\varphi(\Gamma)$ and $\Gamma \to \Gamma'$ implies $\varphi(\Gamma')$.

$\mathsf{fid}(\Gamma)$ is written iff $\varphi(\Gamma)$ and φ is a fidelity property.

A fidelity property is similar to safety, with the difference of requiring contexts to be *fidelitous*—i.e., contexts must be safe *and* if a branch on a selection can be communicated then all branches should be (eventually) capable of communication. This prevents processes from implementing paths which get stuck when there are non-deadlocked paths still in the type context.

Theorem 3.22 (Session Fidelity). Assume $\Gamma \vdash_D P$ with $\mathsf{fid}(\Gamma)$ and $\mathsf{one\text{-}role}(P)$. Then, $\Gamma \to$ implies $\exists \Gamma', P'$ s.t. *(i)* $\Gamma \to \Gamma'$; *(ii)* $P \to_D^* P'$; *(iii)* $\Gamma' \vdash_D P'$; and *(iv)* $\mathsf{one\text{-}role}(P')$.

Proof. (Sketch.) Since a communication action is enabled on the context, the shape of the types can be inferred. From this, the process must be able to perform a communication action, either immediately or after some further reductions (e.g. via R-+, R-X). The proof then proceeds by a case analysis for each possible shape of the process, showing that the process can always mimic at least one of the reductions available to the type. □

Example 3.23 (Safety vs. Fidelity). Note the following *safe* but not *fidelitous* protocol.

$$\Psi = \left\{ \boldsymbol{p} : \oplus \left\{ \begin{array}{l} \boldsymbol{q}:\mathsf{m}.\mathbf{end} \\ \boldsymbol{r}:\mathsf{m}'.\mathbf{end} \end{array} \right. , \quad \boldsymbol{q} : \&\boldsymbol{p}:\mathsf{m}.\mathbf{end}, \quad \boldsymbol{r} : \mathbf{end} \right\}$$

Protocol Ψ is safe since there are no *unexpected messages*—role $\boldsymbol{q}$ expects to receive a message m from role $\boldsymbol{p}$, which does in fact offer this message. The protocol, however, is not fidelitous since only one out of the two choices are eventually capable of communication (the selection towards role $\boldsymbol{r}$ is deadlocked); i.e., there exists a sequence of actions

for which communication towards $\boldsymbol{r}$ is not possible (the empty sequence), violating the universal quantification in the definition of *fidelitous* (Table 3.1). This protocol can type a process violating session fidelity.

$$\mathtt{assoc}_s(\Psi) \vdash_\emptyset \quad s[\boldsymbol{p}][\boldsymbol{r}] \oplus \mathsf{m}' \,.\, \mathbf{0} \quad | \quad s[\boldsymbol{q}][\boldsymbol{p}] \& \mathsf{m} \,.\, \mathbf{0}$$

By rule T-$\oplus$, channel $s[\boldsymbol{p}]$ can implement one of the two choices and remain well-typed. If the implemented choice is the message towards role $\boldsymbol{r}$, then the resulting process is deadlocked even though the type context has an available communication action; violating session fidelity. Hence, Theorem 3.22 relies on a *fidelity property* rather than *safety*. Note that, by the definition of a fidelitous context, both of the following protocols are captured by the largest fidelity property.

$$\Psi' = \left\{ \boldsymbol{p} : \oplus \left\{ \begin{array}{l} \boldsymbol{q}:\mathsf{m}.\mathbf{end} \\ \boldsymbol{r}:\mathsf{m}'.\mathbf{end} \end{array} , \begin{array}{l} \boldsymbol{q} : \&\boldsymbol{p}:\mathsf{m}.\mathbf{end}, \\ \boldsymbol{r} : \&\boldsymbol{p}:\mathsf{m}'.\mathbf{end} \end{array} \right\} \right. \quad \Psi'' = \left\{ \boldsymbol{p} : \oplus \left\{ \begin{array}{l} \boldsymbol{q}:\mathsf{m}.\mathbf{end} \\ \boldsymbol{r}:\mathsf{m}'.\mathbf{end} \end{array} , \begin{array}{l} \boldsymbol{q} : \mathbf{end}, \\ \boldsymbol{r} : \mathbf{end} \end{array} \right\} \right.$$

This demonstrates the interesting qualities of the novel fidelity property, as it is strictly weaker than lock-freedom (and therefore liveness and termination), and can capture both deadlocked and deadlock-free protocols; all whilst being strictly stronger than safety.

3.3.4 Behavioural Properties

The aim of the generalised theory developed thus far has been to provide a framework for verifying runtime properties of programs. The benefit of the type system is that verifying these properties directly on processes is *undecidable*; whereas verifying the properties in table 3.1 on type contexts is *decidable* since the LTS of the type semantics produces a finite state machine [103].

Definition 3.24 (Process properties). A process P is said to be:

deadlock-free (df) iff $P \to_D^* P' \not\to_D$ implies $P' \equiv \mathbf{0}$;

terminating (term) iff $P \to_D^* P'$ implies $\exists n$ finite s.t. $P' \to_D^n P'' \equiv \mathbf{0}$;

never-terminating (n-term) iff $P \to_D^* P'$ implies $P' \to_D$;

lock-free (lf) iff $P \to_D^* P'$ implies:

(1) if $P' \equiv c[\boldsymbol{q}] \oplus \mathsf{m}\langle \vec{V} \rangle \,.\, Q \mid Q'$, then $P' \to_D^* P'' \equiv Q \mid Q''$; and
(2) if $P' \equiv c[\boldsymbol{q}] \,\&_{i \in I}\, \mathsf{m}_i(\vec{b_i}) \,.\, Q_i \mid Q'$, then $\exists k \in I, \vec{V} : P' \to_D^* Q_k\{\vec{V}/b_k\} \mid Q''$;

live (live) iff $P \to_D^* P'$ implies $\exists n$ finite s.t.:

(1) if $P' \equiv c[\boldsymbol{q}] \oplus \mathsf{m}\langle \vec{V} \rangle \,.\, Q \mid Q'$ and $P' \to_D^n P'_n$, then $\exists j \leq n : P'_j \to_D P'' \equiv Q \mid Q''$; and

(2) if $P' \equiv c[\boldsymbol{q}] \,\&_{i \in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, Q_i \mid Q'$ and $P' \to_D^n P'_n$, then $\exists j \leq n, k \in I, \vec{V} : P'_j \to_D P'' \equiv Q_k\{\vec{V}/b_k\} \mid Q''$.

Runtime properties on processes are given in Definition 3.24, and can be informally described as follows.

deadlock-free A process only stops reducing when it reaches the empty process.
terminating A process will reach **0** within a finite number of reductions.
never-terminating A process can always reduce, infinitely.
lock-free A process *can* always reduce to the continuations of outputs/inputs.
live A process *will* always reduce to the continuations of outputs/inputs (within a finite number of steps).

The final result of the core calculus is Theorem 3.25, stating that given a well-typed process obeying the session fidelity assumptions, and a type context captured by the largest fidelity property, then properties on the type context imply properties on processes.

Theorem 3.25 (Property verification). If $\Gamma \vdash_D P$ with $\mathsf{fid}(\Gamma)$ and $\mathsf{one\text{-}role}(P)$, then $\phi(\Gamma) \implies \phi(P)$ for $\phi \in \{\textit{deadlock-free, terminating, never-terminating, lock-free, live}\}$.

Proof. The result holds by inductively applying Theorem 3.22 based on the definition of the property on the type context; then observing that the process typed under the resulting reduced context must be of the shape described by Definition 3.24. □

3.4 Conclusion

This chapter presented a synchronous multiparty session π-calculus with a generalised MPST theory, serving as a demonstration of the bottom-up approach to MPST. The material—whilst not novel in technique—is unique w.r.t. its design and intersection of language features, as the calculus is intended to be seamlessly extended in the coming chapters. There is however novelty in the *fidelity* property required for session fidelity; it is the most general of properties presented in generalised MPST theories with undirected selection. The next chapter extends the core calculus with constructs aimed at promoting the design of modular distributed client-server protocols.

Chapter 4

Idealism: The Client-Server Paradigm

4.1 Introduction

A common design pattern in distributed computing is that of client-server systems [104, 112], where a server is designed to be infinitely available to respond to client requests on demand. In such a paradigm, servers are agnostic of the number of clients in the system, and respond to requests as they are made; i.e., without a bias on ordering. This results in highly modular distributed systems, since the implementations of clients and servers are decoupled. Hence, systems become easier to develop and maintain, and can be easily integrated (servers can be clients to other servers). The formalism of the core calculus presented in the previous chapter fails to model this practical setting.

Example 4.1 (Deadlock-free load balancer). Recall the load balancer example from the previous chapter. In Example 3.11 it was shown that the protocol for the load balancer is *deadlocked*. A deadlock-free version of the system could possibly be described as:

$$\Psi_{lb} = \{srv : S_{srv},\ w_1 : S_w,\ w_2 : S_w,\ c : S_c\}$$

$$S_{srv} = \&c : \mathsf{req}(\mathtt{Int}).\oplus \begin{cases} w_1 : \mathsf{fw}(\mathtt{Int}).\oplus w_2 : \mathsf{stop}().\oplus c : \mathsf{w1}().\mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}).\oplus w_1 : \mathsf{stop}().\oplus c : \mathsf{w2}().\mathbf{end} \end{cases}$$

$$S_w = \&srv : \begin{cases} \mathsf{fw}(\mathtt{Int}).\oplus c : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \mathsf{stop}().\mathbf{end} \end{cases}$$

$$S_c = \oplus srv : \mathsf{req}(\mathtt{Int}).\&srv : \begin{cases} \mathsf{w1}().\&w_1 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \mathsf{w2}().\&w_2 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \end{cases}$$

The difference from the previous load balancer is the use of a control message stop to inform the worker that was not chosen to terminate. Whilst Example 4.1 fixes the deadlock in the load balancer, a number of issues on *practicality* are exposed. Firstly, server- and client-side protocols are tightly coupled. For instance, adding a new worker to the system would require changes to

both the S_{srv} and S_c types; and generalising the server-side to handle multiple clients is non-trivial—handling n clients where n is known requires the server to keep track of how many requests it services so as to only send stop labels when all clients are serviced. Handling an *unknown* number of clients and requests is not possible using standard MPST formalisms since: *(i)* MPSTs assume a statically known set of roles; *(ii)* the *subjects* of actions (e.g., the sender or recipient of a message) are *hard coded* (because role names are *constants*); and *(iii)* servers would not know when it is safe to terminate, and thus may result in deadlocks.

The design of distributed systems in practice relies on the modularisation of server-side and client-side components, and on making servers *reactive*—i.e., observed as infinitely available components that handle requests when prompted. This chapter aims to model this setting by extending the core calculus with *replication* and *first-class roles*.

Example 4.2 (Modular Load Balancer). By extending the core calculus with *replication* and *first-class roles*, Example 4.1 can be generalised to support *any number* of clients.

$$S_{srv} = !\&\alpha : \mathsf{req}(\mathtt{Int}) \,.\, \oplus \begin{cases} w_1 : \mathsf{fw}(\mathtt{Int}, \alpha) \,.\, \oplus\alpha : \mathsf{wrk}(w_1) \,.\, \mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}, \alpha) \,.\, \oplus\alpha : \mathsf{wrk}(w_2) \,.\, \mathbf{end} \end{cases}$$

$$S_w = !\&srv : \mathsf{fw}(\mathtt{Int}, \beta) \,.\, \oplus\beta : \mathsf{ans}(\mathtt{String}) \,.\, \mathbf{end}$$

The first difference is the use of the *bang* (!) operator, denoting a *replicated* receive, i.e., one which may occur *any number* of times. This makes the server agnostic to the *number* of requests. Second, a server now waits for requests—not from a specific client—but from *any* participant, binding the name of the sender to role variable α. This makes the server agnostic to the *source* of requests.

After receiving a request, the server makes a *choice* to forward it to one of two workers, notably whilst passing the name of the client as one of the payloads. Finally, the server informs the client of the choice it made by sending the name of the worker as a *payload* rather than a label.

The type of a worker is also updated to be replicated, as it is dependent on the number of requests forwarded by the server. Notably, it receives the name of the client in the forward message, binding it to β, and uses it to send the final answer. It is key to observe that the use of a control signal is no longer needed, since workers are modelling *infinitely available servers*, and thus are not expected to terminate.

A client may now be defined as:

$$S_c = \oplus srv : \mathsf{req}(\mathtt{Int}) \,.\, \&srv : \mathsf{wrk}(\gamma) \,.\, \&\gamma : \mathsf{ans}(\mathtt{String}) \,.\, \mathbf{end}$$

As a result of the *replicated types* and *first-class roles*, *any number of clients* may now be

> instantiated with type S_c—without needing to make changes to the protocol on the server-side. Furthermore, clients may also be updated to make different numbers of requests, and the server-side protocol could still remain the same. Conversely, if the number of workers in the load balancer were to change, the client type remains safe.

In fact, as will be seen in Section 4.4, the addition of replication—especially when it is used in tandem with recursion—has several surprising consequences, in particular allowing for encoding of *context-free* data structures as well as describing protocols that rely on *races* and *mutual exclusion*.

Contributions

The overarching contribution of this chapter is the first integration of replication and first-class roles into a generalised MPST calculus, and an exploration of the impacts of these extensions on expressiveness and decidability. The specific contributions are as follows:

1. Section 4.2 presents MPST!, the first multiparty session-typed language with *replication* and *first-class roles*, and proves its *metatheory* in Section 4.3.
2. Several expressiveness results are shown through a series of representative examples (Section 4.4.1). In particular, replication lifts the expressive power of types, thus MPST! serves as the first account of *context-free* MPST. Furthermore, in MPST, replication and recursion are mutually non-inclusive—a type system combining both constructs allows for modelling *races* and *mutual exclusion*. These results are demonstrated through non-trivial examples including *binary tree serialisation*, the *dining philosophers problem*, and an *auction service*.
3. In Section 4.4.2, it is shown that the decidability of typechecking is contingent on the decidability of a given safety property; and sublanguages for which typechecking is decidable are identified. (Importantly, these sublanguages are still expressive enough to capture the examples demonstrated in the chapter.)

Section 4.5 gives an account of related work and Section 4.6 concludes the chapter.

4.2 Multiparty Session Types with a Bang!

This section introduces MPST!, a conservative extension of the core calculus presented in Chapter 3, with support for *replication* and *first-class roles*.

4.2.1 Language

Figure 4.1 shows the syntax of MPST!.

$$
\begin{array}{rl}
\textit{Concretes} & d ::= s[\boldsymbol{q}] \mid v \mid \boldsymbol{q} \\
\textit{Channels} & c ::= x \mid s[\boldsymbol{q}] \\
\textit{Binders} & b ::= x \mid \boldsymbol{\alpha} \\
\textit{Names} & a ::= c \mid \boldsymbol{\alpha} \\
\textit{Values} & V ::= c \mid v \mid \boldsymbol{\rho} \\
\textit{Roles} & \boldsymbol{\rho} ::= \boldsymbol{q} \mid \boldsymbol{\alpha} \\
 & \\
\textit{Processes} & P,Q ::= (\nu s)P \mid P|Q \mid \textstyle\sum_{i\in I} P_i^{\oplus} \mid P^{\&} \mid !P^{\&} \mid \mathrm{X}\langle\vec{V}\rangle \mid \mathbf{0} \\
\textit{Send Process} & P^{\oplus} ::= c[\boldsymbol{\rho}] \oplus \mathsf{m}\langle\vec{V}\rangle \,.\, P \\
\textit{Receive Process} & P^{\&} ::= c[\boldsymbol{\rho}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, P_i \\
 & \\
\textit{Programs} & \mathscr{P} ::= (P,D) \\
\textit{Definitions} & D ::= \emptyset \mid D, \mathrm{X} \mapsto (\vec{b},\vec{T},P) \\
\textit{Protocols} & \Psi ::= \{\boldsymbol{p} : S_{\boldsymbol{p}}\}_{\boldsymbol{p}\in I}
\end{array}
$$

Figure 4.1: Syntax of MPST!

Names, values, and binders. A *session name*, ranged over by $s, s', \ldots$, represents a collection of interconnected participants. A *role* is a participant in a multiparty communication protocol, and each *communication endpoint* $s[\boldsymbol{q}]$ is obtained by indexing a session name with a role. In contrast to existing MPST calculi, MPST! supports *first-class* roles, meaning that a role may be communicated as part of a message. To this end, a role $\boldsymbol{\rho}$ may either be a concrete role $\boldsymbol{q}$ (e.g., $\boldsymbol{srv}$, $\boldsymbol{w_1}$, $\boldsymbol{w_2}$, in the load balancer Example 4.2) or a *role variable* $\boldsymbol{\alpha}$.

A name a is either an endpoint, a variable, or a role variable, whereas a value V is either a channel, a basic value, or a role. Binders b are used when receiving a message and can either be a variable binder or a role variable binder. Concrete values d are used when sending a message (at runtime) and are either an endpoint, a basic value, or a concrete role.

Processes. Processes are ranged over by $P, Q, R, \ldots$: *session restriction* $(\nu s)P$ binds session name s in process P; and process $P|Q$ denotes P and Q running in parallel.

As in the core calculus, MPST! supports *choice-guarded output* $\sum_{i\in I} c_i[\boldsymbol{\rho_i}] \oplus \mathsf{m}_i\langle\vec{V}_i\rangle \,.\, P_i$, allowing a nondeterministic send along any c_i to role $\boldsymbol{\rho_i}$ with label m_i and payload $\vec{V}_i$, with the process continuing as P_i. (It is still assumed that all channels c_i are the same.) *Branching receive* $c[\boldsymbol{\rho}]\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, P_i$ denotes a process waiting on channel c for one of a set of messages from role $\boldsymbol{\rho}$ with label m_i, binding the received data to variables $\vec{x}_i$ before continuing according to P_i. It is key to note that the subject of a communication action is indicated via $\boldsymbol{\rho}$ (for both sending and receiving), which can either be a hard-coded value, or a role variable.

Process reduction $\boxed{P_1 \to P_2}$

$$\text{R-C}\quad s[\mathbf{p}][\mathbf{q}] \oplus \mathsf{m}_k\langle \vec{d} \rangle \,.\, P \mid s[\mathbf{q}][\mathbf{p}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, Q_i \;\to_D\; P \mid Q_k\{\vec{d}/\vec{b}_k\} \quad \text{if } k \in I$$

$$\text{R-!C}_1\quad \frac{R = !s[\mathbf{q}][\mathbf{p}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, Q_i}{s[\mathbf{p}][\mathbf{q}] \oplus \mathsf{m}_k\langle \vec{d} \rangle \,.\, P \mid R \;\to\; P \mid R \mid Q_k\{\vec{d}/\vec{b}_k\}} \text{ if } k \in I \qquad \text{R-+}\quad \frac{j \in I}{\sum_{i\in I} P_i^{\oplus} \;\to_D\; P_j^{\oplus}}$$

$$\text{R-!C}_2\quad \frac{R = !s[\mathbf{q}][\boldsymbol{\alpha}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, Q_i}{s[\mathbf{p}][\mathbf{q}] \oplus \mathsf{m}_k\langle \vec{d} \rangle \,.\, P \mid R \;\to\; P \mid R \mid Q_k\{\vec{d}/\vec{b}_k\}\{\mathbf{p}/\boldsymbol{\alpha}\}} \text{ if } k \in I \qquad \text{R-X}\quad \frac{D(\mathrm{X}) = (\vec{x}, \vec{T}, P)}{\mathrm{X}\langle \vec{d} \rangle \;\to_D\; P\{\vec{d}/\vec{x}\}}$$

$$\text{R-|}\quad \frac{P \to_D P'}{P \mid Q \to_D P' \mid Q} \qquad \text{R-}\nu\quad \frac{P \to_D P'}{(\nu s)P \;\to_D\; (\nu s)P'} \qquad \text{R-}\equiv\quad \frac{P \equiv P' \qquad P \;\to_D\; Q \qquad Q \equiv Q'}{P' \;\to_D\; Q'}$$

Figure 4.2: Operational semantics for MPST!.

The second key difference is that receiving processes $P^{\&}$ may be optionally prefixed with the *bang* ! operator, identifying it as *replicated*—i.e., it may be used 0 or more times, modelling *infinitely available servers*. Lastly, *process call* $\mathrm{X}\langle \vec{V} \rangle$ offers recursion and $\mathbf{0}$ denotes the *empty process*.

Programs, Definitions, and Protocols. A *program* is what is written by an end-user of the language; it is a tuple (P, D) of a "main" process P and process *definitions* D. Definitions map process names (ranged over by X,Y,...) to process bodies $(\vec{b}, \vec{T}, P)$ to be used for recursion, where $\vec{b}$ is a list of accepted arguments, $\vec{T}$ is a list of types for said arguments, and P is the process in which the arguments are used. Finally, *protocols* are *session type* specifications of roles in a given session. Formally, Ψ is a set of role-session type pairs, $\{\mathbf{p} : S_{\mathbf{p}}\}_{\mathbf{p}\in I}$ for a non-empty I, defining the communication patterns that each participant of a session should follow.

4.2.2 Process Semantics

Assuming the notion of *structural congruence* $\equiv$ presented in the core calculus (Figure 3.5 top), reduction for MPST! is now formalised.

Definition 4.3 (Process reduction). A process P reduces parametric on some process definitions D, written $P \to_D P'$, via the operational semantics defined in Figure 4.2. Multistep reductions are expressed using the reflexive and transitive closure, i.e., $P \to_D^* P'$.

$$\mathrm{Srv}(r : S_{srv}) \mapsto !r[\alpha]\ \&\ \mathsf{req}(x)\ .\ \left(\begin{array}{c} r[w_1] \oplus \mathsf{fw}\langle x, \alpha\rangle\ .\ r[\alpha] \oplus \mathsf{wrk}\langle w_1\rangle\ .\ \mathbf{0} \\ + \\ r[w_2] \oplus \mathsf{fw}\langle x, \alpha\rangle\ .\ r[\alpha] \oplus \mathsf{wrk}\langle w_2\rangle\ .\ \mathbf{0} \end{array}\right)$$

$$\mathrm{Wrk}(w : S_w) \mapsto !w[srv]\ \&\ \mathsf{fw}(y, \beta)\ .\ w[\beta] \oplus \mathsf{ans}\langle f(y)\rangle\ .\ \mathbf{0}$$

$$\mathrm{Clnt}(c : S_c, n : \mathtt{Int}) \mapsto c[srv] \oplus \mathsf{req}\langle n\rangle\ .\ c[srv]\ \&\ \mathsf{wrk}(\gamma)\ .\ c[\gamma]\ \&\ \mathsf{ans}(z)\ .\ \mathbf{0}$$

Figure 4.3: Process definitions for load balancer example

Rule R-C shows synchronous communication between two processes in session s. The first process, playing role p, is sending a message with label m_k and payloads $\vec{d}$. The second process, playing role q, offers role p a choice of message labels and has associated process continuations. The processes reduce to the selected continuation with the transmitted payloads substituted for the binders in the selected branch.

Rules R-!C$_1$ and R-!C$_2$ describe communication with a *replicated* process R. Rule R-!C$_1$ is similar to R-C but the replicated process remains unchanged and the continuation Q_k is evaluated in parallel. Rule R-!C$_2$ handles the case where the replicated process does not need to receive from a specific role, but instead allows communication with an *arbitrary* role: the rule binds the sending role to α in the replicated continuation. This is referred to as *universal receive*.

Rule R-+ evaluates a branching output by nondeterministically evaluating to one of the sending branches; and Rule R-X handles a recursive call. Rules R-| and R-ν allow reduction under parallel composition and session restriction respectively. Finally, Rule R-$\equiv$ allows for reduction modulo structural congruence.

Example 4.4 (Load Balancer: Process Reduction). This example presents processes, and their reductions, for the protocol given in Example 4.2. The program in question uses the process definitions given in Figure 4.3. Consider a single client in parallel with three server-side processes:

$$\begin{array}{lllll} \mathrm{Clnt}\langle s[c], 42\rangle & \mid \mathrm{Srv}\langle s[srv]\rangle & \mid \mathrm{Wrk}\langle s[w_1]\rangle & \mid \mathrm{Wrk}\langle s[w_2]\rangle \\ \to^* s[c][srv] \oplus \mathsf{req}\langle 42\rangle\ .\ P_c & \mid\ !s[srv][\alpha]\ \&\ \mathsf{req}(x)\ .\ P_{srv} & \mid \mathrm{Wrk}\langle s[w_1]\rangle & \mid \mathrm{Wrk}\langle s[w_2]\rangle \end{array}$$

Using Rule R-!C$_2$, the client and server reduce. The reduction advances the client to its continuation P_c, and pulls out a copy of the server's continuation as a new process. It is key to note that α is acting as a binder in the server process, therefore, in the continuation *role variable substitution* is observed:

$$\rightarrow P_c \mid !s[srv][\alpha] \,\&\, \mathsf{req}(x) \,.\, P_{srv} \mid P_{srv}\{42/x\}\{c/\alpha\} \mid \mathrm{Wrk}\langle s[w_1]\rangle \mid \mathrm{Wrk}\langle s[w_2]\rangle$$

$$\text{where } P_{srv}\{42/x\}\{c/\alpha\} = \left(\sum_{i=1}^{2} s[srv][w_i] \oplus \mathsf{fw}\langle x, \alpha\rangle \,.\, {P_{srv_i}}'\right) \{42/x\}\{c/\alpha\}$$
$$= \sum_{i=1}^{2} s[srv][w_i] \oplus \mathsf{fw}\langle 42, c\rangle \,.\, \left({P_{srv_i}}' \{42/x\}\{c/\alpha\}\right)$$

The spawned server process will then non-deterministically choose a worker to send to via Rule R-+; suppose w_1 is picked.

$$\rightarrow^* \quad P_c \mid !s[srv][\alpha] \,\&\, \mathsf{req}(x) \,.\, P_{srv} \mid s[srv][w_1] \oplus \mathsf{fw}\langle 42, c\rangle \,.\, {P_{srv_1}}''$$
$$\mid !s[w_1][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_1}}' \mid !s[w_2][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_2}}'$$

Communication is now possible between the spawned server process and worker w_1, using Rule R-!C$_1$. As before, this communication advances the sender process and pulls out a copy of the worker's continuation:

$$\rightarrow \quad P_c \mid !s[srv][\alpha] \,\&\, \mathsf{req}(x) \,.\, P_{srv} \mid s[srv][c] \oplus \mathsf{wrk}\langle w_1\rangle \,.\, \mathbf{0}$$
$$\mid !s[w_1][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_1}}' \mid s[w_1][\beta] \oplus \mathsf{ans}\langle f(y)\rangle \,.\, \mathbf{0}\ \{42/y\}\{c/\beta\}$$
$$\mid !s[w_2][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_2}}'$$

$$= \quad P_c \mid !s[srv][\alpha] \,\&\, \mathsf{req}(x) \,.\, P_{srv} \mid s[srv][c] \oplus \mathsf{wrk}\langle w_1\rangle \,.\, \mathbf{0}$$
$$\mid !s[w_1][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_1}}' \mid s[w_1][c] \oplus \mathsf{ans}\langle f(42)\rangle \,.\, \mathbf{0}$$
$$\mid !s[w_2][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_2}}'$$

The client can now learn which worker was chosen by the server, terminating the spawned server process, then the answer is exchanged between the worker and client:

$$\rightarrow \quad s[c][w_1] \,\&\, \mathsf{ans}(z) \,.\, \mathbf{0} \mid !s[srv][\alpha] \,\&\, \mathsf{req}(x) \,.\, P_{srv} \mid \mathbf{0}$$
$$\mid !s[w_1][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_1}}' \mid s[w_1][c] \oplus \mathsf{ans}\langle f(42)\rangle \,.\, \mathbf{0}$$
$$\mid !s[w_2][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_2}}'$$

$$\rightarrow\ !s[srv][\alpha] \,\&\, \mathsf{req}(x) \,.\, P_{srv} \mid !s[w_1][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_1}}' \mid !s[w_2][srv] \,\&\, \mathsf{fw}(y, \beta) \,.\, {P_{w_2}}'$$

$$
\begin{array}{rl}
\textit{Session types} & S ::= S^{\&} \mid !(S^{\&}) \mid S^{\oplus} \mid \mu \mathrm{t}.S \mid \mathrm{t} \mid \mathbf{end} \\
\textit{Branching/Selection types} & S^{\&} ::= \&_{i \in I} \rho : \mathsf{m}_i(\vec{T}_i).S_i \qquad S^{\oplus} ::= \oplus_{i \in I} \rho_i : \mathsf{m}_i(\vec{T}_i).S_i \\
\textit{Role singletons} & \rho ::= q \mid \alpha \\
\textit{Static types} & T ::= S \mid B \mid \rho \\
\textit{Basic types} & B ::= \mathtt{Nat} \mid \mathtt{Real} \mid \mathtt{String} \mid \cdots \\
\textit{Runtime types} & U ::= S \mid (U_1 | U_2)
\end{array}
$$

Figure 4.4: Syntax of Types

4.2.3 Types

Figure 4.4 shows the syntax of MPST! types.

Syntax of types. Session types S type communication endpoints. They consist of branching types $S^{\&}$, *replicated* branching types $!(S^{\&})$, selection types $S^{\oplus}$, recursive types $\mu \mathrm{t}.S$ and variables t, and the **end** type.[1]

Branching session types have the form $\&_{i \in I} \rho : \mathsf{m}_i(\vec{T}_i).S_i$ indicating that role ρ offers a choice of message labels m_i with payload types $\vec{T}_i$ and continuations S_i. Similarly, selection session types $\oplus_{i \in I} \rho_i : \mathsf{m}_i(\vec{T}_i).S_i$ indicate an internal choice towards one of a set of roles ρ_i, with a message label, given payload types and continues as the corresponding continuation type. Branching and selection types are assumed to include a non-empty set of messages with pairwise distinct message names m_i (per role for $\oplus$).

A *replicated* branching type $!\&_{i \in I} \rho : \mathsf{m}_i(\vec{T}_i).S_i$ types a replicated channel. Role ρ can be a concrete role, or a role variable in *binding* position: as will be seen later, the type semantics (cf. Figure 4.7) are responsible for pulling out copies of the type continuation and performing type-level substitution on roles.

The type system supports *singleton* types for roles: specifically a role ρ has singleton type ρ, and these are used to require that a specific role is passed as a payload. *Static* types T are those written as part of a protocol specification, whereas *runtime* types U are introduced by the type semantics to include a notion of *parallel composition* at the level of types: a type $U_1 | U_2$ allows an endpoint to be associated with multiple "active" session types.

Definition 4.5 (Subtyping). The **subtyping relation** $\leqslant$ is co-inductively defined on types

[1] For selection and (replicated) branching types with a single path, the shorthand notations $\rho \oplus \mathsf{m}(\vec{T}).S$, $\rho \& \mathsf{m}(\vec{T}).S$ and $!\rho \& \mathsf{m}(\vec{T}).S$ are sometimes used.

by the following inference rules:

$$\frac{(\vec{T}_i \leqslant \vec{T}_i')_{i\in I} \qquad (S_i \leqslant S_i')_{i\in I}}{\&_{i\in I}\boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T}_i).S_i \leqslant \&_{i\in I\cup J}\boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T}_i').S_i'} \qquad \frac{(\vec{T}_i \geqslant \vec{T}_i')_{i\in I} \qquad (S_i \leqslant S_i')_{i\in I}}{\oplus_{i\in I\cup J}\boldsymbol{\rho_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \leqslant \oplus_{i\in I}\boldsymbol{\rho_i}{:}\mathsf{m}_i(\vec{T}_i').S_i'}$$

$$\frac{S_1^{\&} \leqslant S_2^{\&}}{!S_1^{\&} \leqslant !S_2^{\&}} \qquad \frac{U \leqslant U' \qquad S \leqslant S'}{U \,|\, S \leqslant U' \,|\, S'} \qquad \frac{S\{\mu \mathrm{t}.S/\mathrm{t}\} \leqslant S'}{\mu \mathrm{t}.S \leqslant S'} \qquad \frac{S \leqslant S'\{\mu \mathrm{t}.S'/\mathrm{t}\}}{S \leqslant \mu \mathrm{t}.S'} \qquad \frac{}{T \leqslant T}$$

The definition of subtyping (Definition 4.5) is mostly standard. The adopted approach is the convention of smaller types being ones with less external choice and more internal choice (*à la* Gay and Hole [45]). Subtyping of replication is based on regular branching; and parallel types are related iff their session types are subtypes. Subtypes are related up-to their recursive unfolding, and subtyping is *reflexive*.

Definition 4.6 (Type Congruence). *Type congruence* treats parallel runtime types as commutative and associative with identity element **end**.

$$U_1 \,|\, U_2 \equiv U_2 \,|\, U_1 \qquad (U_1 \,|\, U_2) \,|\, U_3 \equiv U_1 \,|\, (U_2 \,|\, U_3) \qquad U \,|\, \mathbf{end} \equiv U$$

Figure 4.5 shows the definition of the typing context Γ and its operations. Context Γ maps channels to runtime types, variable names to static types, and role variable names to their associated singleton type. Context composition Γ, Γ' is defined only if both Γ and Γ' have disjoint domains. Subtyping and type congruence are lifted to contexts in the expected way.

As inspired by (e.g.) [114], linearity is enforced through the use of a *context split* operation $\Gamma = \Gamma_1 \cdot \Gamma_2$ that splits a context Γ into two environments Γ_1 and Γ_2. These environments may share variables with non-linear types (basic types and role singletons). Additionally, a channel c with runtime type $U_1|U_2$ may be split such that Γ_1 contains $c : U_1$ and Γ_2 contains $c : U_2$; this allows for typing of processes pulled out in parallel at runtime via rules R-!C$_1$ and R-!C$_2$.

The (left) inverse operation is *context addition* $\Gamma_1 + \Gamma_2 = \Gamma$ combining Γ_1 and Γ_2 into an environment Γ; again roles and unrestricted variables may be shared across environments. In the case of $\Gamma_1, c : U_1 + \Gamma_2, c : U_2$ (i.e., adding two session types associated with the same endpoint), context addition results in the channel c having the combined runtime type $U_1|U_2$.

The *role insertion* operation $\Gamma \hookleftarrow \boldsymbol{\rho}$ is used when typing replicated receives and extends a context with a mapping $\boldsymbol{\alpha} : \boldsymbol{\alpha}$ in the case that $\boldsymbol{\rho}$ is a role variable, and leaves Γ unchanged otherwise.

Typing contexts

$$\Gamma ::= \emptyset \mid \Gamma, s[p] : U \mid \Gamma, x : T \mid \Gamma, \alpha : \alpha$$

Context splitting $\boxed{\Gamma = \Gamma_1 \cdot \Gamma_2}$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = \Gamma_1, c : U \cdot \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = \Gamma_1 \cdot \Gamma_2, c : U} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, x : B = \Gamma_1, x : B \cdot \Gamma_2, x : B}$$

$$\frac{}{\emptyset = \emptyset \cdot \emptyset} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, \alpha : \alpha = \Gamma_1, \alpha : \alpha \cdot \Gamma_2, \alpha : \alpha} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U_1 | U_2 = \Gamma_1, c : U_1 \cdot \Gamma_2, c : U_2}$$

Context addition $\boxed{\Gamma_1 + \Gamma_2 = \Gamma}$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad a \notin \mathsf{dom}(\Gamma_2)}{\Gamma_1, a : T + \Gamma_2 = \Gamma, a : T} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \qquad a \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + \Gamma_2, a : T = \Gamma, a : T} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, x : B + \Gamma_2, x : B = \Gamma, x : B}$$

$$\frac{}{\Gamma + \emptyset = \Gamma} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, \alpha : \alpha + \Gamma_2, \alpha : \alpha = \Gamma, \alpha : \alpha} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, c : U_1 + \Gamma_2, c : U_2 = \Gamma, c : U_1 | U_2}$$

Context role insertion $\boxed{\Gamma \hookleftarrow \rho}$

$$\Gamma \hookleftarrow q = \Gamma \qquad\qquad \Gamma \hookleftarrow \alpha = \Gamma + \alpha : \alpha$$

Figure 4.5: Typing contexts and context operations.

Type Checking

Using the updated definition for an *end-typed* environment (Definition 4.7), the typing rules for MPST! are given in Figure 4.6.

Definition 4.7 (End-typed environment). A context is *end-typed*, written $\mathsf{end}(\Gamma)$, iff its channels have type **end** (up-to type congruence $\equiv$).

$$\frac{}{\mathsf{end}(\emptyset)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(c : \mathbf{end}, \Gamma)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(x : B, \Gamma)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(\alpha : \alpha, \Gamma)} \qquad \frac{\Gamma \equiv \Gamma' \quad \mathsf{end}(\Gamma)}{\mathsf{end}(\Gamma')}$$

Typing rules use the same three typing judgements from the core calculus: *(i)* the *value judgement* $\Gamma \vdash V : T$ assigns type T to value V under context Γ; *(ii)* the *program judgement* $\vdash \mathscr{P}$ checks whether a program is self-enclosed and *type safe*; and *(iii)* the *process judgement* $\Gamma \vdash_D P$ checks whether a process P is well-typed under a context Γ and process definitions D.

As in the core calculus, Rule T-Sub types variables and endpoints, allowing for subtyping;

Typing Rules for Values $\boxed{\Gamma \vdash V : T}$

$$\text{T-Sub}\ \frac{T \leqslant T'}{c : T \vdash c : T'} \qquad \text{T-Wkn}\ \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad \Gamma_1 \vdash V : T \quad \mathsf{end}(\Gamma_2)}{\Gamma \vdash V : T} \qquad \text{T-B}\ \frac{v \in B}{\emptyset \vdash v : B} \qquad \text{T-}\boldsymbol{q}\ \frac{}{\emptyset \vdash \boldsymbol{q} : \boldsymbol{q}} \qquad \text{T-}\boldsymbol{\alpha}\ \frac{}{\boldsymbol{\alpha} : \boldsymbol{\alpha} \vdash \boldsymbol{\alpha} : \boldsymbol{\alpha}}$$

Typing Rules for Programs and Processes $\boxed{\vdash \mathscr{P}}$ $\boxed{\Gamma \vdash_D P}$

$$\text{T-}\mathscr{P}\ \frac{\emptyset \vdash_D P \quad \forall i \in I : \{\overrightarrow{b_i : T_i}\} \vdash_D Q_i}{\vdash (P, D = \{\mathrm{X}_i \mapsto (\vec{b}_i, \vec{T}_i, Q_i)\}_{i \in I})} \qquad \text{T-}\nu\ \frac{\varphi(\mathtt{assoc}_s(\Psi)) \quad \varphi \text{ is a } !\text{-}\textit{safety} \text{ property} \quad s \notin \Gamma \quad \Gamma + \mathtt{assoc}_s(\Psi) \vdash_D P}{\Gamma \vdash_D (\nu s : \Psi)\, P} \qquad \text{T-}+\ \frac{(\Gamma \vdash_D P_i^{\oplus})_{i \in I}}{\Gamma \vdash_D \sum_{i \in I} P_i^{\oplus}}$$

$$\text{T-X}\ \frac{D(\mathrm{X}) = (\vec{b}, \vec{T}, P) \quad \forall i \in 1..n : \Gamma_i \vdash V_i : T_i \quad \mathsf{end}(\Gamma_0)}{\Gamma_0 (\cdot \Gamma_i)_{i \in 1..n} \vdash_D \mathrm{X}\langle (V_i)_{i \in 1..n} \rangle} \qquad \text{T-}!\ \frac{\Gamma_! \vdash c : !\&_{i \in I} \boldsymbol{\rho} : \mathsf{m}_i(\vec{T}_i).S_i \quad \mathsf{end}(\Gamma) \quad (\Gamma + c : S_i + \overrightarrow{b_i : T_i} \leftrightarrow \boldsymbol{\rho} \vdash_D P_i)_{i \in I}}{\Gamma \cdot \Gamma_! \vdash_D !c[\boldsymbol{\rho}]\ \&_{i \in I}\, \mathsf{m}_i(\vec{b}_i)\ .\ P_i} \qquad \text{T-}|\ \frac{\Gamma_1 \vdash_D P_1 \quad \Gamma_2 \vdash_D P_2}{\Gamma_1 \cdot \Gamma_2 \vdash_D P_1 | P_2}$$

$$\text{T-}\&\ \frac{\Gamma_\& \vdash c : \&_{i \in I} \boldsymbol{\rho} : \mathsf{m}_i(\vec{T}_i).S_i \quad \Gamma_r \vdash \boldsymbol{\rho} : \boldsymbol{\rho} \quad (\Gamma + c : S_i + \overrightarrow{b_i : T_i} \vdash_D P_i)_{i \in I}}{\Gamma \cdot \Gamma_\& \cdot \Gamma_r \vdash_D c[\boldsymbol{\rho}]\ \&_{i \in I}\, \mathsf{m}_i(\vec{b}_i)\ .\ P_i} \qquad \text{T-}\oplus\ \frac{\Gamma_\oplus \vdash c : \oplus \boldsymbol{\rho} : \mathsf{m}(\vec{T}).S \quad \Gamma_r \vdash \boldsymbol{\rho} : \boldsymbol{\rho} \quad (\Gamma_i \vdash V_i : T_i)_{i \in 1..n} \quad \Gamma + c : S \vdash_D P}{\Gamma \cdot \Gamma_\oplus \cdot \Gamma_r (\cdot \Gamma_i)_{i \in 1..n} \vdash_D c[\boldsymbol{\rho}] \oplus \mathsf{m}\langle (V_i)_{i \in 1..n} \rangle\, .\, P} \qquad \text{T-}\mathbf{0}\ \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash_D \mathbf{0}}$$

Figure 4.6: Typing rules.

Rule T-Wkn allows for weakening; and Rule T-B types a basic value v to type B iff v belongs to B. New to MPST! are Rule T-$\boldsymbol{q}$, which types concrete roles as singleton types under the empty environment; and Rule T-$\boldsymbol{\alpha}$, which types a role variable provided that it is contained within the type environment.

The program judgement is only used by rule T-$\mathscr{P}$ to type a program (P, D). This judgement does not use a type environment, and ensures that programs are *closed*. This is achieved by requiring the main process P to be well-typed under an empty context. Furthermore, process definitions are also required to be closed by typing them under a context containing only the arguments defined for that process name.

For the process judgement, the following rules have not changed. Rule T-$\mathbf{0}$ types the inactive process under an end-typed environment. Rule T-$+$ types an output-directed choice; since this operation uses branching control flow, each choice must be typable under the same environment. Rule T-X types recursive processes by checking that types of values used in process calls match the types specified in their definitions. Parallel composition Rule T-$|$ types two parallel processes under a split environment; it is key to note that even though this rule is unchanged, the updated definition of context splitting (Figure 4.5) will cater for processes pulled out of replicated receives at runtime.

Context Reduction $\boxed{\Gamma \xrightarrow{A} \Gamma'}$

$$\Gamma\text{-}\& \quad \frac{S = \&_{i\in I}\boldsymbol{q}{:}\mathsf{m}_i(\vec{T}_i).S'_i \qquad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S'_k} \qquad \Gamma\text{-}\oplus \quad \frac{S = \oplus_{i\in I}\boldsymbol{q}_i{:}\mathsf{m}_i(\vec{T}_i).S'_i \qquad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}_k:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S'_k}$$

$$\Gamma\text{-}\mu \quad \frac{\Gamma \cdot c : S\{\mu \mathrm{t}.S/\mathrm{t}\} \xrightarrow{A} \Gamma'}{\Gamma \cdot c : \mu \mathrm{t}.S \xrightarrow{A} \Gamma'}$$

$$\Gamma\text{-}! \quad \frac{R = !\&_{i\in I}\boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T}_i).S_i \qquad k \in I}{s[\boldsymbol{q}] : R \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{\rho}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{q}] : R \,|\, S_k}$$

$$\Gamma\text{-}\textsc{Cong}_1 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma, \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''} \qquad \Gamma\text{-}\textsc{Cong}_2 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma \cdot \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''}$$

$$\Gamma\text{-}\textsc{Com}_1 \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \vec{T}'_1 \leqslant \vec{T}'_2 \qquad \vec{T}_1 = (\vec{\boldsymbol{r}}, \vec{T}'_1) \qquad \vec{T}_2 = (\vec{\boldsymbol{\alpha}}, \vec{T}'_2) \qquad \Gamma_1 \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T}_1)} \Gamma'_1 \qquad \Gamma_2 \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}(\vec{T}_2)} \Gamma'_2}{\Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \Gamma'_1 + \Gamma'_2\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}}\}}$$

$$\Gamma\text{-}\textsc{Com}_2 \quad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2 \qquad \vec{T}'_1 \leqslant \vec{T}'_2 \qquad \vec{T}_1 = (\vec{\boldsymbol{r}}, \vec{T}'_1) \qquad \vec{T}_2 = (\vec{\boldsymbol{\alpha}}, \vec{T}'_2) \qquad \Gamma_1 \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T}_1)} \Gamma'_1 \qquad \Gamma_2 \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{\alpha}':\mathsf{m}(\vec{T}_2)} \Gamma'_2}{\Gamma \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \Gamma'_1 + \Gamma'_2\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}}\}\{\boldsymbol{p}/\boldsymbol{\alpha}'\}}$$

Figure 4.7: Type semantics.

Rule T-! and Rule T-& type replicated and non-replicated receives respectively. In both cases the rules check that the channel has a receiving session type. Both rules check that each branch is typable by extending a common typing context with the variables bound by the receives, along with the continuation type for the session channel. In the case of a replicated receive there are two differences: first, the context used to type each branch must be end-typed (in order to avoid duplicating linear resources). Second, since the role $\boldsymbol{\rho}$ may be a *binding* occurrence of a role variable $\boldsymbol{\alpha}$, the context used to type each branch must be extended with the role variable (if applicable) using the insertion operator.

Rule T-$\oplus$ types an output by checking that the sending channel can be mapped to a selection type with payload types that match the values being sent. The rule ensures that role $\boldsymbol{\rho}$ is either a value, or it exists in the type context—i.e., messages cannot be sent to unbound role variables. The selection type continuation should be used, along with the common context, to type the continuation process.

Rule T-ν types a session name restriction $(\nu s : \Psi)P$ using an updated safety property, catering for appropriate use of replication and free role names. The specifics of this property are discussed in Section 4.3.

4.2.4 Type Semantics

Definition 4.8 (Context reduction). Labels for the type LTS are updated to allow for role variables. The new definition for actions A, is given below.

$$\text{Actions} \quad A ::= s{:}\boldsymbol{q}\oplus\boldsymbol{r}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{q}\&\boldsymbol{\rho}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{q},\boldsymbol{r}{:}\mathsf{m}$$

The type LTS is defined by the transitions listed in Figure 4.7. Context *reduction*, $\Gamma\to\Gamma'$, is defined iff $\Gamma\xrightarrow{A}\Gamma'$ where $A = s{:}\boldsymbol{q},\boldsymbol{r}{:}\mathsf{m}$ for any $s,\boldsymbol{q},\boldsymbol{r},\mathsf{m}$—i.e., a context reduces iff it can transition via a communication action. The transitive and reflexive closure of context reduction is written as $\to^*$; and $\Gamma\to$ represents the existence of some transition from Γ.

Transitions Γ-& and Γ-$\oplus$ are standard: a context can fire an input label (resp. output label) matching any of the branches in the top-level branch type (resp. selection type). Doing so transitions the context entry to the continuation type of the selected branch.

Transition Γ-! models the receipt of a message by a replicated input. The two main differences to the linear receive are: *(i)* the role $\boldsymbol{\rho}$ used in the transition label is allowed to be a role variable name; and *(ii)* firing an input does not advance the type, but instead pulls out a copy of the continuation and places it in parallel. Role $\boldsymbol{\rho}$ is considered bound in R and its continuations, but is *free* in pulled out copies of the continuations composed in parallel (S_k).

Transition rules Γ-Cong$_1$, Γ-Cong$_2$, Γ-μ allow contexts to reduce under a larger context, or when types are guarded by recursive binders.

Transitions Γ-Com$_1$ and Γ-Com$_2$ model type-level communication; for simplicity and without loss of generality, assume a convention wherein role-typed payloads are at the head. Both rules state that if a context can be split such that one part fires an output label, and the other fires an input with matching roles, message label and payloads, then the entire context can transition via a *communication* action. Sender payloads must be subtypes of the receiver payloads, provided they are not role types, for which *role substitution* occurs after communication. Rule Γ-Com$_2$ additionally caters for universal receives, where the input label consists of a role variable in binding position—this is reflected in the role substitution.

Role substitution, and the type semantics in general, are demonstrated in Example 4.9.

Example 4.9 (Load Balancer: Context Reduction). Recalling the load balancer protocol from Example 4.2, context reduction is now demonstrated for a system with one server, two workers, and one client. Initially, the only communication action possible is between

the client and server, via Rule Γ-Com$_2$.

$$
\begin{aligned}
& s[c] : S_c,\ s[srv] : S_{srv},\ s[w_1] : S_w,\ s[w_2] : S_w \\
=\ & (s[c] : S_c \cdot s[srv] : S_{srv}) \cdot (s[w_1] : S_w,\ s[w_2] : S_w) \\
\rightarrow\ & s[c] : srv\&\mathsf{wrk}(\gamma).\gamma\&\mathsf{ans}(\mathtt{String}).\mathbf{end} \\
& + \left(s[srv] : S_{srv} \;\middle|\; \oplus \left\{ \begin{array}{l} w_1{:}\mathsf{fw}(\mathtt{Int},\alpha).\alpha\oplus\mathsf{wrk}(w_1).\mathbf{end} \\ w_2{:}\mathsf{fw}(\mathtt{Int},\alpha).\alpha\oplus\mathsf{wrk}(w_2).\mathbf{end} \end{array} \right. \right)\{c/\alpha\} \\
& + s[w_1] : S_w,\ s[w_2] : S_w \\
=\ & s[c] : srv\&\mathsf{wrk}(\gamma).\gamma\&\mathsf{ans}(\mathtt{String}).\mathbf{end}, \\
& s[srv] : S_{srv} \;\middle|\; \oplus \left\{ \begin{array}{l} w_1{:}\mathsf{fw}(\mathtt{Int},c).c\oplus\mathsf{wrk}(w_1).\mathbf{end} \\ w_2{:}\mathsf{fw}(\mathtt{Int},c).c\oplus\mathsf{wrk}(w_2).\mathbf{end} \end{array} \right. , s[w_1] : S_{w_1},\ s[w_2] : S_{w_2}
\end{aligned}
$$

It is key to note the role substitution for the type of $s[srv]$ above; specifically, how the substitution affects the parallel type extracted through communication, but does not affect the replicated type S_{srv}. This is because only the role variables in the pulled out continuation are considered free. From here, there are multiple reduction paths. Consider the path in which srv communicates with w_1.

$$
\begin{aligned}
=\ & s[c] : srv\&\mathsf{wrk}(\gamma).\gamma\&\mathsf{ans}(\mathtt{String}).\mathbf{end},\ s[w_2] : S_w,\ s[srv] : S_{srv} \\
& \cdot \left(s[srv] : \oplus \left\{ \begin{array}{l} w_1{:}\mathsf{fw}(\mathtt{Int},c).c\oplus\mathsf{wrk}(w_1).\mathbf{end} \\ w_2{:}\mathsf{fw}(\mathtt{Int},c).c\oplus\mathsf{wrk}(w_2).\mathbf{end} \end{array} \right. \cdot s[w_1] : S_w \right) \\
\rightarrow\ & s[c] : srv\&\mathsf{wrk}(\gamma).\gamma\&\mathsf{ans}(\mathtt{String}).\mathbf{end},\ s[w_2] : S_w,\ s[srv] : S_{srv} \\
& + (s[srv] : c\oplus\mathsf{wrk}(w_1).\mathbf{end} + (s[w_1] : S_w \,|\, \gamma\oplus\mathsf{ans}(\mathtt{String}).\mathbf{end})\{c/\gamma\}) \\
=\ & s[c] : srv\&\mathsf{wrk}(\gamma).\gamma\&\mathsf{ans}(\mathtt{String}).\mathbf{end},\ s[w_2] : S_w, \\
& s[srv] : S_{srv} \,|\, c\oplus\mathsf{wrk}(w_1).\mathbf{end},\ s[w_1] : S_w \,|\, c\oplus\mathsf{ans}(\mathtt{String}).\mathbf{end}
\end{aligned}
$$

Note how reduction is possible because the context split allows the server's parallel type to be extracted into its own context. Then, reduction occurs via Rule Γ-Cong$_1$ and Rule Γ-Com$_1$. From here, the context reduces in a similar fashion: first the server communicates a role with the client; followed by the final communication between the client

Free role variables in types $\boxed{\mathsf{frv}(T) = \{\alpha_i\}_{i\in I}}$

$$\begin{aligned}
\mathsf{frv}(B) = \mathsf{frv}(q) = \mathsf{frv}(\mathrm{t}) = \mathsf{frv}(\mathbf{end}) &= \emptyset \\
\mathsf{frv}(\alpha) &= \{\alpha\} \\
\mathsf{frv}(\mu \mathrm{t}.S) &= \mathsf{frv}(S) \\
\mathsf{frv}(\oplus_{i\in I}\rho_i{:}m_i(\vec{T}_i).S_i) &= \bigcup_{i\in I} \mathsf{frv}(\rho_i) \cup \mathsf{frv}(\vec{T}_i) \cup \mathsf{frv}(S_i) \\
\mathsf{frv}(\&_{i\in I}\rho{:}m_i(\vec{T}_i).S_i) &= \mathsf{frv}(\rho) \bigcup_{i\in I} (\mathsf{frv}(S_i) \setminus \mathsf{frv}(\vec{T}_i)) \\
\mathsf{frv}(!\&_{i\in I}\rho{:}m_i(\vec{T}_i).S_i) &= \bigcup_{i\in I} \Big(\mathsf{frv}(S_i) \setminus (\mathsf{frv}(\rho) \cup \mathsf{frv}(\vec{T}_i))\Big)
\end{aligned}$$

Substitution in contexts $\boxed{\Gamma\{q/\alpha\} = \Gamma}$

$$\frac{}{(c : T)\{q/\alpha\} = c : (T\{q/\alpha\})} \qquad \frac{\mathsf{end}(\Gamma)}{\Gamma\{q/\alpha\} = \Gamma}$$

$$\frac{}{(\Gamma_1 \cdot \Gamma_2)\{q/\alpha\} = (\Gamma_1\{q/\alpha\}) \cdot (\Gamma_2\{q/\alpha\})} \qquad \frac{}{(\Gamma_1, \Gamma_2)\{q/\alpha\} = (\Gamma_1\{q/\alpha\}), (\Gamma_2\{q/\alpha\})}$$

Figure 4.8: Free role variables and substitution in contexts

and worker.

$$\begin{aligned}
&\to \\
&\quad s[c] : w_1\&\mathsf{ans}(\mathtt{String}).\mathbf{end},\ s[srv] : S_{srv} \,|\, \mathbf{end}, \\
&\quad s[w_1] : S_w \,|\, c\oplus\mathsf{ans}(\mathtt{String}).\mathbf{end},\ s[w_2] : S_w \\
&\to \\
&\quad s[c] : \mathbf{end},\ s[srv] : S_{srv} \,|\, \mathbf{end},\ s[w_1] : S_w \,|\, \mathbf{end},\ s[w_2] : S_w \\
&\equiv \\
&\quad s[c] : \mathbf{end},\ s[srv] : S_{srv},\ s[w_1] : S_w,\ s[w_2] : S_w
\end{aligned}$$

A formal definition of free role variables (frv) is given in Figure 4.8 (top). In selection types, the free role variables are the ones in the subject position and in payloads. For branching, payloads are in binding position, thus only a role variable in the subject is free. Lastly, both the subject role and payloads are in binding position for replicated receives.

Substitution on types $T\{q/\alpha\}$ then refers to replacing all free occurrences of α in T with q. The operation is lifted to type contexts in Figure 4.8 (bottom). Notably, substitution should only affect session-typed entities in contexts, thus it ignores end-typed environments and is propagated across context composition and context splits (to handle parallel types).

4.3 Metatheory

Following the methodology of the generalised approach to MPST outlined in Section 3.3, this section now proves the metatheory for MPST!. First, given the calculus has been extended to include replication and first-class roles, some definitions of properties need to be updated—these are presented in Section 4.3.1. Section 4.3.2 presents the necessary lemmata and propositions on which the main results are dependent upon. *Subject reduction* is presented and proved in Section 4.3.3, and *session fidelity* in Section 4.3.4. Lastly, Section 4.3.5 updates process properties to allow for the infinite behaviour of replicated processes, and proves that the updated properties can be inferred from type contexts. Simplified proofs are presented in the main text; full proofs can be found in appendix A.

4.3.1 Context Properties

The updated definition of a context path is given in Definition 4.11—notably, only the *live* path definition has changed. The update relies on an assumption that is made for all MPST! types henceforth, i.e., message labels used in linear communication are unique w.r.t. those used by replicated receives.

Definition 4.10 (Unique Label Sets). Message labels are assumed to be taken from one of two unique and infinitely countable sets L and R, such that $\mathsf{L} \cap \mathsf{R} = \emptyset$. Sending and receiving types are updated to be constructed using the following:

Linear Branch	$S^{\&} ::= \&_{i \in I} \boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T}_i).S_i$	where $\forall i \in I : \mathsf{m}_i \in \mathsf{L}$
Replicated Branch	$!S^{\&} ::= !\&_{i \in I} \boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T}_i).S_i$	where $\forall i \in I : \mathsf{m}_i \in \mathsf{R}$
Selection	$S^{\oplus} ::= \oplus_{i \in I} \boldsymbol{\rho_i}{:}\mathsf{m}_i(\vec{T}_i).S_i$	where $\forall i \in I : \mathsf{m}_i \in \mathsf{L} \cup \mathsf{R}$

Definition 4.11 (Path). A type context *path* is a possibly infinite sequence of type contexts, $\vec{\Gamma}$, s.t. $\forall \Gamma_i \in \vec{\Gamma} : \Gamma_i \rightarrow \Gamma_{i+1}$. A path $\vec{\Gamma}$ is said to be:

1. *terminating* iff $\vec{\Gamma}$ is finite;
2. *fair* iff $\forall \Gamma_i \in \vec{\Gamma} : \Gamma_i \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \implies \exists \Gamma_k \in \vec{\Gamma}, \boldsymbol{r}, \mathsf{m}' : k \geq i \wedge \Gamma_k \xrightarrow{s:\boldsymbol{p},\boldsymbol{r}:\mathsf{m}'} \Gamma_{k+1}$
3. *live* iff $\forall \Gamma_i \in \vec{\Gamma}$:
   (a) $\Gamma_i \xrightarrow{s:\boldsymbol{p} \oplus \boldsymbol{q}:\mathsf{m}(\vec{T})} \implies \exists \Gamma_k \in \vec{\Gamma}, \boldsymbol{r}, \mathsf{m}' : k \geq i \wedge \Gamma_k \xrightarrow{s:\boldsymbol{p},\boldsymbol{r}:\mathsf{m}'} \Gamma_{k+1}$; and
   (b) $\Gamma_i \xrightarrow{s:\boldsymbol{p} \& \boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge\ \mathsf{m} \in \mathsf{L} \implies \exists \Gamma_k \in \vec{\Gamma}, \mathsf{m}' \in \mathsf{L} : k \geq i \wedge \Gamma_k \xrightarrow{s:\boldsymbol{q},\boldsymbol{p}:\mathsf{m}'} \Gamma_{k+1}$.

Given the semantic approach to replication at the type level, the previous definition of a live path (Definition 3.12) becomes too strict. Using that version, protocols involving finite

clients communicating with infinitely available servers would be incapable of being live, since the context would eventually reach a state where clients have type **end** and servers are capable of receiving (through a replicated branch). Therefore, the liveness condition is updated to only apply for *linear* branching types, and does not apply for replicated types (since they are *always* capable of firing an input action).

Definition 4.12 (All Enabled). A context with transition $\Gamma \xrightarrow{s:p,q:\mathsf{m}}$ is said to be enabled for *all* labels m iff a context can perform the transition for every path labelled by m. Formally:

$$\mathsf{all}\left(\Gamma \xrightarrow{s:p,q:\mathsf{m}}\right) ::= \Gamma \xrightarrow{s:p,q:\mathsf{m}} \wedge\ \Gamma = \Gamma_0(\cdot \Gamma_i^{\&})_{i\in I} \wedge\ \Gamma_0 \xrightarrow{s:q\&p:m(\vec{T'})} \not\wedge$$
$$\forall i \in I : \Gamma_i^{\&} = \Gamma_i^{e}, c_i : S_i \wedge \mathsf{end}(\Gamma_i^{e}) \wedge \Gamma_i^{\&} \xrightarrow{s:q\&p:m(\vec{T_i})} \implies (\Gamma_0 + \Gamma_i^{\&}) \xrightarrow{s:p,q:\mathsf{m}}$$

Example 4.13. Consider the following contexts, Γ and Γ'.

$$\Gamma = \left\{ \begin{array}{l} s[p] : !S \mid q\&\mathsf{m}(\mathtt{Int}) \mid q\&\mathsf{m}(\mathtt{String}) \\ s[q] : p\oplus\mathsf{m}(\mathtt{Int}).S' \end{array} \right\}$$

$$\Gamma' = \left\{ \begin{array}{l} s[p] : !S \mid q\&\mathsf{m}(\mathtt{Int}) \mid q\&\mathsf{m}(\mathtt{Int}) \\ s[q] : p\oplus\mathsf{m}(\mathtt{Int}).S' \end{array} \right\}$$

Only context Γ' is *all*-enabled for transition $\xrightarrow{s:p,q:\mathsf{m}}$. This is because Γ can be split in a way such that some splits can communicate via m, where others cannot, due to mismatching payloads. Specifically:

$$\Gamma = \{s[p] : !S, s[q] : p\oplus\mathsf{m}(\mathtt{Int}).S'\} \cdot \{s[p] : q\&\mathsf{m}(\mathtt{Int})\} \cdot \{s[p] : q\&\mathsf{m}(\mathtt{String})\}$$

Observe that:

$$\{s[p] : !S, s[q] : p\oplus\mathsf{m}(\mathtt{Int}).S'\} + \{s[p] : q\&\mathsf{m}(\mathtt{String})\} \xrightarrow{s:p,q:\mathsf{m}} \not\wedge$$

Using the updated definitions for paths, message labels and the new *all* predicate, context properties for MPST! types are presented in Table 4.1. Changes have been made to the definitions of *safe* and *deadlock-free* contexts. The former makes the same safety requirement as in Table 3.1, but now splits the condition to hold separately for linear and replicated communication. In other words, *(i)* if a linear message can be sent and received between two participants, then communication should be possible; and *(ii)* if a replicated message can be sent and received

safe	$\left(\Gamma \xrightarrow{s:p\oplus q:\mathsf{m}(\vec{T})} \wedge\ \Gamma \xrightarrow{s:q\&p:\mathsf{m}'(\vec{T}')} \wedge\ \mathsf{m},\mathsf{m}' \in \mathsf{L} \implies \mathsf{all}\left(\Gamma \xrightarrow{s:p,q:\mathsf{m}}\right)\right)$ and $\Gamma \xrightarrow{s:p\oplus q:\mathsf{m}(\vec{T})} \wedge\ \Gamma \xrightarrow{s:q\&\rho:\mathsf{m}'(\vec{T}')} \wedge\ \mathsf{m},\mathsf{m}' \in \mathsf{R} \wedge \rho \in \{p,\alpha\} \implies \mathsf{all}\left(\Gamma \xrightarrow{s:p,q:\mathsf{m}}\right)$
fidelitous	$\Gamma \xrightarrow{s:p\oplus q:\mathsf{m}(\vec{T})} \wedge\ \Gamma \xrightarrow{s:p,r:\mathsf{m}'} \implies \forall \vec{A} : \Gamma \xrightarrow{\vec{A}}{}^{*}\, \Gamma' \xrightarrow{s:p,q:\mathsf{m}} \wedge\ p \notin \mathtt{roles}(\vec{A})$
deadlock-free	$\Gamma \to^{*} \Gamma' \not\to \implies \mathsf{end!}(\Gamma')$
terminating	Γ is deadlock free and all paths from Γ are terminating
never-term.	$\Gamma \to^{*} \Gamma' \implies \Gamma' \to$
lock-free	$\Gamma \to^{*} \Gamma' \implies$ all fair paths from Γ' are also live
live	$\Gamma \to^{*} \Gamma' \implies$ all paths from Γ' are live

Table 4.1: Properties of a type context Γ, in MPST!.

between two participants, then communication should be possible. In both cases, communication must be enabled for all paths prefixed with the label chosen by the sender. (Otherwise, context splits would not preserve safety, cf. Lemma A.13(1).) Splitting the condition prevents protocols being flagged as unsafe when a linear message can be sent, a replicated receive is possible, but the corresponding linear receive is hidden behind some action.

Example 4.14 (Updated Safe Context). Context Γ' below is safe w.r.t. the definition provided in Table 4.1, but is not safe using the property defined for the core calculus (Table 3.1).

$$\Gamma = \left\{ \begin{array}{l} s[p] : !q\&\mathsf{m}.r\&\mathsf{m}'.q\&\mathsf{m}' \\ s[q] : p \oplus \mathsf{m}.p \oplus \mathsf{m}' \\ s[r] : p \oplus \mathsf{m}' \end{array} \right\} \to \Gamma' = \left\{ \begin{array}{l} s[p] : !q\&\mathsf{m}.r\&\mathsf{m}'.q\&\mathsf{m}' \mid r\&\mathsf{m}'.q\&\mathsf{m}' \\ s[q] : p \oplus \mathsf{m}' \\ s[r] : p \oplus \mathsf{m}' \end{array} \right\}$$

Once again, due to situations in which terminating clients communicate with replicated servers, the definition for *deadlock-free* contexts is also updated. Essentially, the end predicate used for typechecking is too strict for verifying deadlock-freedom. Hence, the property now relies on a new end! predicate presented in Definition 4.15, allowing for replicated types.

Definition 4.15 (End!-typed environment). A context is *end!-typed*, written $\mathsf{end!}(\Gamma)$, iff all of its channels are replicated (up-to $\equiv$) or have the type **end**: i.e., $\forall c \in \mathsf{dom}(\Gamma) : \Gamma(c) \equiv !S^{\&} \vee \Gamma(c) \leqslant \mathbf{end}$.

The definitions of other properties have remained unchanged.

4.3.2 Role Substitution

The following presents non-standard lemmata for the novel role substitution used in MPST!. Their intention is to provide further insight into the internal workings of first-class roles in the designed calculus, and they are integral to the larger proofs of main results presented later. An exhaustive list of all the lemmata, along with their detailed proofs, can be found in appendix A.1.

Lemma 4.16 states that role substitution is preserved through addition, whilst lemma 4.17 states that it preserves subtyping. These are then used to present the main role substitution lemmas, for the value judgement (lemma 4.18) and the process judgement (lemma 4.19).

Lemma 4.16. If $\Gamma_1 + \Gamma_2 = \Gamma$, then $\Gamma_1\{q/\alpha\} + \Gamma_2\{q/\alpha\} = \Gamma\{q/\alpha\}$.

Proof. By induction on the derivation of $\Gamma_1 + \Gamma_2 = \Gamma$. A sample case is given below.

Case 1: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma \qquad a \notin \mathsf{dom}(\Gamma_2)}{\Gamma_1, a : T + \Gamma_2 = \Gamma, a : T}$

First, observe that by Figure 4.8, substitution does not affect context domains.

$(\Gamma_1, a : T)\{q/\alpha\} = (\Gamma_1\{q/\alpha\}), (a : T\{q/\alpha\})$	(by Figure 4.8)	(1)
$\Gamma_1\{q/\alpha\} + \Gamma_2\{q/\alpha\} = \Gamma\{q/\alpha\}$	(ind. hyp.)	(2)
$a \notin \mathsf{dom}(\Gamma_2\{q/\alpha\})$	(since $a \notin \mathsf{dom}(\Gamma_2)$)	(3)
$(\Gamma_1, a : T)\{q/\alpha\} + \Gamma_2\{q/\alpha\} = (\Gamma\{q/\alpha\}), (a : T\{q/\alpha\})$	(by (1), (2), (3), +)	(4)
Case 1 holds.	(by (4), Figure 4.8)	(5)

Other cases are either similar or follow directly from the inductive hypothesis. □

Lemma 4.17. $T \leqslant T'$ implies $T\{q/\alpha\} \leqslant T'\{q/\alpha\}$.

Proof. By coinduction on $T \leqslant T'$ (Definition 4.5).

Case 1: $\dfrac{}{T \leqslant T}$

The reflexive case immediately holds since the substitution occurs on the same type on both sides.

Case 2: $\dfrac{(\vec{T}_i \leqslant \vec{T}'_i)_{i \in I} \qquad (S_i \leqslant S'_i)_{i \in I}}{\&_{i \in I} \rho : \mathsf{m}_i(\vec{T}_i).S_i \leqslant \&_{i \in I \cup J} \rho : \mathsf{m}_i(\vec{T}'_i).S'_i}$

From the coinductive hypothesis, it follows that $\forall i \in I : \vec{T}_i\{q/\alpha\} \leqslant \vec{T}'_i\{q/\alpha\}$ and $S_i\{q/\alpha\} \leqslant S'_i\{q/\alpha\}$. Furthermore, if $\rho = \alpha$, then the substitution occurs on both sides.

Other cases follow the same reasoning and hold directly from the coinductive hypothesis and since any role substitution made on the type subject affects both sides equally. □

Lemma 4.18. If $\Gamma \vdash V : T$, then $\Gamma\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$.

Proof. By rule induction on $\Gamma \vdash V : T$. Two sample cases are given below.

Case T-Sub: $\dfrac{T \leqslant T'}{\Gamma = c : T \vdash c : T'}$. RTP: $(c : T)\{q/\alpha\} \vdash c\{q/\alpha\} : T'\{q/\alpha\}$

Note that if c is a variable and T is *not* a session type, then role substitution has no effect and thus the case would hold trivially. Consider now $T = S$ and $T' = S'$.

$S \leqslant S'$	(by assumption)	(1)
$S\{q/\alpha\} \leqslant S'\{q/\alpha\}$	(by (1) and lemma 4.17)	(2)
$(c : S)\{q/\alpha\} \vdash c : S'\{q/\alpha\}$	(by (2) and rule T-Sub)	(3)
$c\{q/\alpha\} = c$	(since $c = x$ or $c = s[p]$, for any x, s, p)	(4)
Case T-Sub holds.	(by (3), (4), and since $S = T$ and $S' = T'$)	(5)

Case T-Wkn: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma \qquad \Gamma_1 \vdash V : T \qquad \mathsf{end}(\Gamma_2)}{\Gamma \vdash V : T}$. RTP: $\Gamma\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$

$\Gamma_1\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$	(ind. hyp.)	(1)
$\mathsf{end}(\Gamma_2)$	(by assumption)	(2)
$\Gamma_2 = \Gamma_2\{q/\alpha\}$	(by Figure 4.8)	(3)
$\Gamma_1\{q/\alpha\} + \Gamma_2 = \Gamma\{q/\alpha\}$	(by hyp., (3), and lemma 4.16)	(4)
$\Gamma\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$	(by (1), (2), (4))	(5)
Case T-Wkn holds.	(by (5))	(6)

□

Lemma 4.19. If $\Gamma \vdash_D P$, then $\Gamma\{q/\alpha\} \vdash_D P\{q/\alpha\}$.

Proof. By rule induction on $\Gamma \vdash_D P$.

Case T-0: $\dfrac{\mathsf{end}(\Gamma)}{\Gamma \vdash_D \mathbf{0}}$. Since $\mathsf{end}(\Gamma)$, then $\Gamma\{q/\alpha\} = \Gamma$ (by Figure 4.8).

Case T-⊕: $$\frac{\Gamma_\oplus \vdash c : \oplus \rho : \mathsf{m}(\vec{T}).S \qquad \Gamma_r \vdash \rho : \rho \qquad (\Gamma_i \vdash V_i : T_i)_{i \in 1..n} \qquad \Gamma_0 + c : S \vdash_D P'}{\Gamma = \Gamma_0 \cdot \Gamma_\oplus \cdot \Gamma_r (\cdot \Gamma_i)_{i \in 1..n} \vdash_D P = c[\rho] \oplus \mathsf{m}\langle (V_i)_{i \in 1..n} \rangle . P'}$$

First observe that by the definition of role substitution on contexts (Figure 4.8), substitution distributes over context splits. Then observe that by the ind. hyp.:

$$(\Gamma_0 + c : S)\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash_D P'\{\boldsymbol{q}/\boldsymbol{\alpha}\}$$

Furthermore, by lemma 4.18:

$$\begin{gathered}\Gamma_{\oplus}\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash (c\{\boldsymbol{q}/\boldsymbol{\alpha}\}) : (\oplus \boldsymbol{\rho}:\mathsf{m}(\vec{T}).S\{\boldsymbol{q}/\boldsymbol{\alpha}\}) \\ \Gamma_r\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash (\boldsymbol{\rho}\{\boldsymbol{q}/\boldsymbol{\alpha}\}) : (\boldsymbol{\rho}\{\boldsymbol{q}/\boldsymbol{\alpha}\}) \\ \forall i \in 1..n : \Gamma_i\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash (V_i\{\boldsymbol{q}/\boldsymbol{\alpha}\}) : (T_i\{\boldsymbol{q}/\boldsymbol{\alpha}\})\end{gathered}$$

Then, using T-$\oplus$, the case holds.
All other cases follow the same steps, relying on the inductive hypothesis and lemma 4.18.
□

4.3.3 Subject Reduction

Using the lemmata from the previous section, the main results of the language are presented, starting with *subject reduction*. To reiterate, subject reduction (Theorem 4.21) informally states that, if a well-typed process can reduce, then its reductums remain well-typed. The property is proven parametric on a largest safety property, which is updated for MPST! in Definition 4.20.

Definition 4.20 (Safety). φ is a !-*safety* property on type environment Γ iff:
φ-**S** $\varphi(\Gamma)$ implies Γ is *safe*;
φ-$\boldsymbol{\rho}$ $\varphi(\Gamma)$ implies $\mathsf{frv}(\Gamma) = \emptyset$;
φ-μ $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu \mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu \mathrm{t}.S/\mathrm{t}\})$;
φ-$\rightarrow$ $\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$.
$\mathsf{safe}_!(\Gamma)$ is written iff $\varphi(\Gamma)$ and φ is a !-safety property.

As in the core calculus, the property requires contexts (along with their reductions and unfoldings) to be safe (as defined in table 4.1). Furthermore, the updated property now requires that contexts are void of unbound role variables. Otherwise, role variable singletons in contexts may end up being shadowed by role variables defined in different sessions.

Theorem 4.21 (Subject Reduction). If $\Gamma \vdash_D P$ with $\mathsf{safe}_!(\Gamma)$ and $P \rightarrow_D P'$, then $\exists \Gamma'$ s.t. $\Gamma \rightarrow^* \Gamma'$ and $\Gamma' \vdash_D P'$ with $\mathsf{safe}_!(\Gamma')$.

Proof. By induction on the derivation of $P \rightarrow_D P'$. A sample case is given below.

Additional cases are given in Theorem A.14.

The assumptions are: **(A1)** $\Gamma \vdash_D P$; **(A2)** $\mathsf{safe}_!(\Gamma)$; and **(A3)** $P \to_D P'$.

Case R-!C$_2$:

Begin by inferring the shape of processes P and P' from the reduction rule R-!C$_2$. Then by assumptions (A1) and inversion of typing rules, the shape of context Γ can be inferred.

$$\text{Let } R = !s[\boldsymbol{q}][\boldsymbol{\delta}]\ \&_{i\in I}\ \mathsf{m}_i(\vec{b}_i)\ .\ Q'_i \qquad \text{(definition)} \quad (1)$$
$$\text{Let } P_\oplus = s[\boldsymbol{p}][\boldsymbol{q}] \oplus \mathsf{m}_k\langle\vec{d}\rangle\ .\ Q \qquad \text{(definition)} \quad (2)$$
$$\text{Let } n = |\vec{d}| \qquad \text{(definition)} \quad (3)$$
$$P = P_\oplus \mid R \qquad \text{(by R-!C}_2\text{ and (A3))} \quad (4)$$
$$P' = Q \mid R \mid Q'_k\{\vec{d}/\vec{b}_k\}\{\boldsymbol{p}/\boldsymbol{\delta}\} \qquad \text{(by R-!C}_2\text{ and (A3))} \quad (5)$$

Without loss of generality, consider $\vec{d} = \vec{\boldsymbol{r}}, \vec{d}'$ where $\vec{d}'$ does not contain role values.

$$\Gamma = \Gamma^\oplus \cdot \Gamma^! \vdash_D P \qquad \text{(by (4), (A1), and T-|)} \quad (6)$$

$$\Gamma^\oplus = \Gamma^\oplus_0 \cdot \Gamma^\oplus_c \cdot \Gamma^\oplus_r \cdot (\cdot \Gamma^\oplus_{d_i})_{i\in 1..n} \vdash_D P_\oplus \qquad \text{(by (6), (4), T-}\oplus\text{)} \quad (7)$$
$$\Gamma^\oplus_c \vdash s[\boldsymbol{p}] : \oplus \boldsymbol{q}{:}\mathsf{m}_k(\vec{\boldsymbol{r}}, \vec{T}).S \qquad \text{(by (7), T-}\oplus\text{)} \quad (8)$$
$$\Gamma^\oplus_r \vdash \boldsymbol{q} : \boldsymbol{q} \qquad \text{(by (7), T-}\oplus\text{)} \quad (9)$$
$$\forall i \in 1..|\vec{\boldsymbol{r}}| : \Gamma^\oplus_{d_i} \vdash d_i : \boldsymbol{r_i} \text{ and } \forall j \in |\vec{\boldsymbol{r}}|..|\vec{T}| : \Gamma^\oplus_{d_j} \vdash d_j : T_j \qquad \text{(by (7), T-}\oplus\text{)} \quad (10)$$
$$\Gamma^\oplus_0 + s[\boldsymbol{p}] : S \vdash_D Q \qquad \text{(by (7), T-}\oplus\text{)} \quad (11)$$

$$\Gamma^! = \Gamma^!_0 \cdot \Gamma^!_c \vdash_D R \qquad \text{(by (6), (4), T-!)} \quad (12)$$
$$\Gamma^!_c \vdash s[\boldsymbol{q}] : !\&_{i\in I}\boldsymbol{\delta}{:}\mathsf{m}_i(\vec{\boldsymbol{\alpha}}_i, \vec{T'_i}).S'_i \qquad \text{(by (12), T-!)} \quad (13)$$
$$\mathsf{end}(\Gamma^!_0) \qquad \text{(by (12), T-!)} \quad (14)$$
$$\forall i \in I : \Gamma^!_0 + s[\boldsymbol{q}] : S'_i + \overrightarrow{\boldsymbol{\alpha}_i : \boldsymbol{\alpha}_i} + \overrightarrow{x_i : T'_i} \hookleftarrow \boldsymbol{\delta} \vdash_D Q'_i \qquad \text{(by (12), T-!)} \quad (15)$$

Since Γ models sending and receiving actions, and since Γ is safe (table 4.1) by assumption (A2), then the transitions enabled for the context can be inferred. Lemma A.13 (used below) states that if a context is safe, then its splits are also safe in isolation.

$$\Gamma^\oplus_c \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}_k(\vec{\boldsymbol{r}},\vec{T})} \Gamma^{\oplus\prime}_c \qquad \text{(by (8), T-Wkn, T-Sub, Definition 4.5)} \quad (16)$$
$$\Gamma^!_c \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{\delta}:\mathsf{m}'_j(\vec{\boldsymbol{\alpha}}_j,\vec{T'_j})} \Gamma^{!\prime}_{c_j} \text{ for } j \in I \qquad \text{(by (13), T-Wkn, T-Sub, Definition 4.5)} \quad (17)$$
$$\Gamma^\oplus_c \cdot \Gamma^!_c \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}_k} \Gamma^{\oplus\prime}_c + \Gamma^{!\prime}_{c_k}\{\vec{\boldsymbol{r}}, \boldsymbol{p}/\vec{\boldsymbol{\alpha}}, \boldsymbol{\delta}\} \quad \text{(by (16), (17), (A2), lemma A.13, } \Gamma\text{-Com}_1\text{)} \quad (18)$$

$$\Gamma^{\oplus\prime}_c \vdash s[\boldsymbol{p}] : S \qquad \text{(by (8), (16), } \Gamma\text{-}\oplus\text{, T-}\oplus\text{)} \quad (19)$$
$$\Gamma^{!\prime}_{c_k} = \Gamma^!_c \cdot \Gamma_k \text{ such that:} \qquad \text{(by (13), (17), } \Gamma\text{-!)} \quad (20)$$
$$\Gamma_k \vdash s[\boldsymbol{q}] : S'_k \qquad \text{(by (20), (13), (17), } \Gamma\text{-!, T-!)} \quad (21)$$
$$\Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}}_k\}\{\boldsymbol{p}/\boldsymbol{\delta}\} \vdash s[\boldsymbol{q}] : S'_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}}_k\}\{\boldsymbol{p}/\boldsymbol{\delta}\} \qquad \text{(by (21), lemma 4.18)} \quad (22)$$
$$\vec{T} \leqslant \vec{T'_k} \qquad \text{(by (16), (17), (18), } \Gamma\text{-Com}_1\text{)} \quad (23)$$

Finally, contexts under which the reduced processes are typed can be identified from

the typing rules, and it is subsequently shown that a valid context is obtained from the previously observed context reduction. Lemma A.7 states that singleton types can be removed from a context if they do not appear free in the process, and lemma A.8 is the standard substitution lemma.

$\Gamma_0^{\oplus} + \Gamma_c^{\oplus\prime} \vdash_D Q$ (by (11), (19)) (24)

$\Gamma_0^{!} + \Gamma_c^{!} \vdash_D R$ (by (12)) (25)

$\Gamma_0^{!} + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\} + \overrightarrow{\alpha_i : \alpha_i} + \overrightarrow{x_i : T_i'} \vdash_D Q_k'\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}$
(by (15), (22), lemma 4.19) (26)

$\Gamma_0^{!} + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\} + \overrightarrow{x_i : T_i'} \vdash_D Q_k'\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}$ (by (26), lemma A.7) (27)

$\Gamma_0^{!} + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}(+\Gamma_{d_i}^{\oplus})_{i\in 1..n} \vdash_D Q_k'\{\vec{d}/\vec{b_k}\}$ ((10), (15), (23), lemma A.8) (28)

$\Gamma_0^{\oplus} + \Gamma_c^{\oplus\prime} + \Gamma_0^{!} + \Gamma_c^{!} + \Gamma_0^{!} + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}(+\Gamma_{d_i}^{\oplus})_{i\in 1..n} \vdash_D P'$
(by (24), (25), (28), T-|) (29)

By (12) and (14), $\Gamma_0^{!}$ is end-typed. Assume without loss of generality that $\Gamma_0^{!}$ does not contain any session typed entities, since any end-typed channels could be grouped with a different split of the context instead. Then, it holds that $\Gamma_0^{!} = \Gamma_0^{!} \cdot \Gamma_0^{!}$. Lemma A.1 states that addition is the left-inverse of context splitting.

$\Gamma \to \Gamma' = (\Gamma_c^{\oplus\prime} + (\Gamma_c^{!} \cdot \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\})) + (\Gamma_0^{!} \cdot \Gamma_0^{!} \cdot \Gamma_0^{\oplus} \cdot \Gamma_r^{\oplus}(\cdot \Gamma_{d_i}^{\oplus})_{i\in 1..n})$
(by (18), Γ-Cong$_1$) (30)

$= \Gamma_0^{\oplus} + \Gamma_c^{\oplus\prime} + \Gamma_0^{!} + \Gamma_c^{!} + \Gamma_0^{!} + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}(+\Gamma_{d_i}^{\oplus})_{i\in 1..n} + \Gamma_r^{\oplus}$
(by (30), lemma A.1) (31)

$\mathsf{end}(\Gamma_r^{\oplus})$ (by (9), T-Wkn, T-q) (32)

Also assume, without loss of generality, that $\Gamma_r^{\oplus}$ is void of end-typed channels. Then, the proof can be concluded using lemma A.2, stating that a context void of channel mappings can be added to a type environment with no effect on the typing judgement.

$\Gamma' \vdash_D P'$ (by (29), (31), (32), lemma A.2) (33)

$\mathsf{safe}_!(\Gamma')$ (by (A2), (30), φ-$\to$) (34)

Case R-!C$_2$ holds. (by (30), (33), (34)) (35)

□

4.3.4 Session Fidelity

Session fidelity in MPST! (Theorem 4.27) is similar to the theorem defined for the core calculus, but this time uses an updated fidelity property. The theorem, as is standard, states that if a context can reduce and it types a process under some certain constraints, then the process should be able to mimic at least one reduction modelled by the context.

$$\begin{array}{ll} \textit{Processes} & P,Q ::= \sum_{i\in I} P_i^{\oplus} \mid P^{\&} \mid !P^{\&} \mid \mathrm{X}\langle\vec{V}\rangle \mid \mathbf{0} \\ \textit{Programs} & \mathscr{P} ::= ((\nu s:\Psi)P_1 \mid \cdots \mid P_n, D) \end{array}$$

Figure 4.9: Session Fidelity Restrictions on MPST!

Definition 4.22 (Fidelity). φ is a !-*fidelity* property on type environment Γ iff:

(i) $\varphi(\Gamma)$ implies Γ is *fidelitous*;

(ii) $\varphi(\Gamma)$ implies $\mathsf{frv}(\Gamma) = \emptyset$;

(iii) $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu \mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu \mathrm{t}.S/\mathrm{t}\})$;

(iv) $\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$.

$\mathsf{fid}_!(\Gamma)$ is written iff $\varphi(\Gamma)$ and φ is a !-fidelity property.

The updated fidelity property (Definition 4.22) requires all role variables in a context to be bound (just as in Definition 4.20), and contexts should be *fidelitous* (as defined in table 4.1). These conditions should hold after type unfolding and context reduction.

The constraints on processes required for session fidelity are similar to those identified in Section 3.3.3, redefined in the following for convenience. Definition 4.23 requires that session typed arguments in process definitions must be used for communication at least once before passed as arguments in a process call. This ensures that process definitions are productive, and thus deadlocks cannot occur by vacuously reducing using rule R-X.

Definition 4.23 (Guarded definitions). Process definitions D are said to be *guarded* iff for all mappings of the form $\mathrm{X} \mapsto (\vec{x}, \vec{T}, P)$ in D, if $\mathrm{Y}\langle\vec{V}\rangle$ is a subterm of P, then: $\exists i \in 1..|\vec{x}|$ s.t. $T_i \leqslant S \not\leqslant \mathbf{end}$ and $x_i \in \vec{V}$ implies $\mathrm{Y}\langle\vec{V}\rangle$ is guarded by a communication action on x_i in P.

Programs are also restricted to be defined by the sub-grammar of MPST! presented in Figure 4.9, allowing for an ensemble of parallel processes communicating via a single multiparty channel. Lastly, parallel processes are restricted to only play a single role, as defined by Definition 4.24. Different to the previous chapter, multiple processes may possibly play the same role at runtime, provided that this only results from communication with replicated branches.

Definition 4.24 (Plays one role). Assume programs to be constructed via the grammar in Figure 4.9. Then, given the judgement $\Gamma \vdash_D P$, it is said that P *only plays role* $\boldsymbol{p}$ *by* Γ iff: *(i)* D is guarded; *(ii)* $\mathsf{fv}(P) = \emptyset$; *(iii)* $\Gamma = \Gamma', s[\boldsymbol{p}] : U$ with $U \not\leqslant \mathbf{end}$ and $\mathsf{end}(\Gamma')$.

Furthermore, it is said that:

1. P *only plays role* $\boldsymbol{p}$ iff $\exists \Gamma, D$ s.t. $\Gamma \vdash_D P$ and the above conditions hold; and

2. $\mathsf{one\text{-}role}_!(P)$ iff $P \equiv \Big|_{p \in I} P_p$ where each P_p is either $\mathbf{0}$ (up-to $\equiv$), or only plays role p.

The following lemmas are used together to infer information about processes from a type context given the session fidelity assumptions. Lemma 4.25 infers the shape of a type environment for an ensemble of processes that follow the session fidelity assumptions; whilst lemma 4.26 (known as session inversion) determines the possible shapes of a process typed under contexts identified from the initial lemma.

Lemma 4.25. Assume $\Gamma \vdash_D P$ with $\mathsf{one\text{-}role}_!(P)$. Then $P \equiv \Big|_{p \in I} P_p$ and $\Gamma = \Gamma_0(\cdot \Gamma'_p, s[p] : U_p)_{p \in I'}$, where:

1. $\forall p \in I' : \Gamma'_p, s[p] : U_p \vdash_D P_p$ and $c : U \notin \Gamma'_p$;
2. $\forall q \in I \setminus I' : P_q \equiv \mathbf{0}$;
3. $c : U \notin \Gamma_0$.

Proof. First observe that by Definition 4.24(2), $P \equiv \Big|_{p \in I} P_p$ where $\forall p \in I : P_p \equiv \mathbf{0}$ or P_p only plays role p. Therefore, by rule T-$|$, $\Gamma = (\cdot \Gamma_p)_{p \in I}$ where $\forall p \in I : \Gamma_p \vdash_D P_p$. For each split of the context, there are two cases to consider.

If $P_p \equiv \mathbf{0}$, then by rule T-$\mathbf{0}$, $\mathsf{end}(\Gamma_p)$.

Otherwise, P_p only plays role p. Since $\Gamma_p \vdash_D P_p$, then Γ_p serves as witness for Definition 4.24(1). Hence, it can be concluded that $\Gamma_p = \Gamma''_p, s[p] : U_p$ with $U_p \not\leqslant \mathbf{end}$ and $\mathsf{end}(\Gamma''_p)$.

Therefore, $\Gamma = (\cdot \Gamma''_p, s[p] : U_p)_{p \in I''} (\cdot \Gamma_q)_{q \in I \setminus I''}$ where $\forall p \in I'' : \Gamma''_p, s[p] : U_p \vdash_D P_p$ with $\mathsf{end}(\Gamma''_p)$ and $\forall q \in I \setminus I'' : \Gamma_q \vdash_D P_q \equiv \mathbf{0}$ with $\mathsf{end}(\Gamma_q)$.

Without loss of generality, assume that all end-typed channels are grouped in $I \setminus I''$. Hence, $\forall p \in I'' : \Gamma''_p, s[p] : U_p \vdash_D P_p$ with $c : U \notin \Gamma''_p$ and $\forall q \in I \setminus I'' : \Gamma_q \vdash_D P_q \equiv \mathbf{0}$ with $\mathsf{end}(\Gamma_q)$ and either $\Gamma_q = \Gamma''_q, s[q] : \mathbf{end}$ or $c : U \notin \Gamma''_q$.

Thus, $(\cdot \Gamma_q)_{q \in I \setminus I''} = \Gamma_0 (\cdot \Gamma_q)_{q \in I'''}$ where $c : U \notin \Gamma_0$ and $I''' \subseteq I \setminus I''$.

Let $I' = (I \setminus I'') \cup I'''$, then $\Gamma = \Gamma_0(\cdot \Gamma'_p, s[p] : U_p)_{p \in I'}$. □

The session inversion lemma (lemma 4.26) can be used to infer further information about the contexts obtained from lemma 4.25. Essentially, processes either directly mimic the action in the type, or they are a process call to a recursive definition that will then perform the action identified by the type. Different to a standard session inversion lemma, it is possible for the type to model a parallel composition of session types. This is shown to type a parallel composition of processes which obey the assumptions of the lemma, thus allowing the proposition to be inductively applied on the smaller processes.

Lemma 4.26. Assume $\Gamma_0, s[\mathbf{p}] : U_{\mathbf{p}} \vdash_D P_{\mathbf{p}}$ with $c : U \notin \Gamma_0$ and either $P_{\mathbf{p}}$ only plays role $\mathbf{p}$ or $P_{\mathbf{p}} \equiv \mathbf{0}$. Then:

1. if $S_1 | \cdots | S_n \leqslant U_{\mathbf{p}}$, then $P_{\mathbf{p}} \equiv P_1 | \cdots | P_n$ and $\forall i \in 1..n : \Gamma_0, s[\mathbf{p}] : S_i \vdash_D P_i$ and P_i only plays role $\mathbf{p}$;
2. if $!\&_{i\in I}\mathbf{p}{:}\mathsf{m}_i(\vec{T}_i).S_i \leqslant U_{\mathbf{p}}$, then either:
   (a) $P_{\mathbf{p}} \equiv !s[\mathbf{p}][\mathbf{p}]\&_{i\in I'}\mathsf{m}_i(\vec{b}_i).P'_i$ and $I \subseteq I'$; or
   (b) $P_{\mathbf{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv !s[\mathbf{p}][\mathbf{p}]\&_{i\in I'}\mathsf{m}_i(\vec{b}_i).P'_i$ and $I \subseteq I'$;
3. if $\&_{i\in I}\mathbf{q}{:}\mathsf{m}_i(\vec{T}_i).S_i \leqslant U_{\mathbf{p}}$, then either:
   (a) $P_{\mathbf{p}} \equiv s[\mathbf{p}][\mathbf{q}]\&_{i\in I'}\mathsf{m}_i(\vec{b}_i).P'_i$ and $I \subseteq I'$; or
   (b) $P_{\mathbf{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv s[\mathbf{p}][\mathbf{q}]\&_{i\in I'}\mathsf{m}_i(\vec{b}_i).P'_i$ and $I \subseteq I'$;
4. if $\oplus_{i\in I}\mathbf{q}_i{:}\mathsf{m}_i(\vec{T}_i).S_i \leqslant U_{\mathbf{p}}$, then either:
   (a) $P_{\mathbf{p}} \equiv \sum_{i\in I'} s[\mathbf{p}][\mathbf{q}_i] \oplus \mathsf{m}_i\langle\vec{V}_i\rangle.P'_i$ and $I \supseteq I'$; or
   (b) $P_{\mathbf{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv \sum_{i\in I'} s[\mathbf{p}][\mathbf{q}_i] \oplus \mathsf{m}_i\langle\vec{V}_i\rangle.P'_i$ and $I \supseteq I'$;
5. if $\mathbf{end} \leqslant U_{\mathbf{p}}$, then $P_{\mathbf{p}} \equiv \mathbf{0}$.

Proof. By case analysis on the 5 possible implications.

Case 1:

Follows from the definition of context splits (Figure 4.5), observing that since $c : U \notin \Gamma_0$ then $\Gamma_0, s[\mathbf{p}] : S_1 | \cdots | S_n = \Gamma_0, s[\mathbf{p}] : S_1 \cdot \ldots \cdot \Gamma_0, s[\mathbf{p}] : S_n$.

Furthermore, by rule T-|, $\forall i \in 1..n : \Gamma_0, s[\mathbf{p}] : S_i \vdash_D P_i$.

Lastly, from the hypothesis it follows that D is guarded, $\mathsf{fv}(P) = \emptyset$ and $U_{\mathbf{p}} \not\leqslant \mathbf{end}$, thus $\forall i \in 1..n : \mathsf{fv}(P_i) = \emptyset$ and $S_i \not\leqslant \mathbf{end}$. (Notice that any $\mathbf{end}$ in the parallel type can be removed via type congruence, and also reflected in the process.) Therefore, by Definition 4.24, $\forall i \in 1..n : P_i$ only plays role $\mathbf{p}$, by $\Gamma_0, s[\mathbf{p}] : S_i$.

Case 2:

From the hypothesis, and by subtyping and narrowing, $\Gamma_0, s[\mathbf{p}] : !\&_{i\in I}\mathbf{p}{:}m_i(\vec{T}_i).S_i \vdash_D P_{\mathbf{p}}$. By inversion of typing rules, there are two possible rules that could be applied.

Rule T-!:

$\Gamma_0, s[\boldsymbol{p}] : !\&_{i\in I}\boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T_i}).S_i \vdash_D !s[\boldsymbol{p}][\boldsymbol{\rho}]\&_{i\in I}\mathsf{m}_i(\vec{b_i}).P_i'$ (by T-!) (1)

$\Gamma_0, s[\boldsymbol{p}] : U_{\boldsymbol{p}} \vdash_D !s[\boldsymbol{p}][\boldsymbol{\rho}]\&_{i\in I'}\mathsf{m}_i(\vec{b_i}).P_i'$ where $I \subseteq I'$ (by (1) and Definition 4.5) (2)

$P_{\boldsymbol{p}} \equiv !s[\boldsymbol{p}][\boldsymbol{\rho}]\&_{i\in I'}\mathsf{m}_i(\vec{b_i}).P_i'$ (by (2) and subject congruence (lemma A.12)) (3)

Rule T-X:

By inversion of T-X, $P_{\boldsymbol{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $\exists k \in 1..|\vec{V}| : V_k = s[\boldsymbol{p}]$ and $s[\boldsymbol{p}] : U_{\boldsymbol{p}} \vdash V_k : T_k'$. Without loss of generality, assume $k = 1$, then $D(\mathrm{X}) = ((x, \vec{b}), (T_k, \vec{T'}), Q)$.

Assuming that $\Gamma_0, s[\boldsymbol{p}] : U_{\boldsymbol{p}} \vdash_D P_{\boldsymbol{p}}$ is derived from a well-typed program, then it follows that $x : T_k', \{\overrightarrow{b : T'}\} \vdash_D Q$ by rule T-$\mathscr{P}$. Furthermore, by Definition 4.23, it follows that Q cannot be typed by a further application of rule T-X, and thus must instead be typed by rule T-!. Following the same reasoning as outlined in the previous subcase, it must be that $Q \equiv !x[\boldsymbol{\rho'}]\&_{i\in I'}\mathsf{m}_i(\vec{b_i}).P_i''$ and $I \subseteq I'$. Lastly, it follows that either $\boldsymbol{\rho'} = \boldsymbol{\rho}$, or $\exists k' \in 1..|\vec{V}| : V_{k'} = \boldsymbol{\rho}$ and $b_{k'} = \boldsymbol{\rho'}$. Either way, it results that $Q\{\vec{V}/\vec{b'}\} \equiv !s[\boldsymbol{p}][\boldsymbol{\rho}]\&_{i\in I'}\mathsf{m}_i(\vec{b_i}).P_i'$.

Case 3, 4: Similar to Case 2.

Case 5:

Since $\mathbf{end} \leqslant U_{\boldsymbol{p}}$ and $c : U \notin \Gamma_0$, then $\mathsf{end}(\Gamma_0, s[\boldsymbol{p}] : U_{\boldsymbol{p}})$. By inversion of the typing rules, the only possible shape for $P_{\boldsymbol{p}}$ is $P_{\boldsymbol{p}} \equiv \mathbf{0}$ via rule T-**0**. □

Theorem 4.27 (Session Fidelity). Assume $\Gamma \vdash_D P$ with $\mathsf{fid}_!(\Gamma)$ and $\mathsf{one\text{-}role}_!(P)$. Then, $\Gamma\!\rightarrow$ implies $\exists\Gamma', P'$ s.t.:

(i) $\Gamma\!\rightarrow\Gamma'$;

(ii) $P \rightarrow_D^* P'$;

(iii) $\Gamma' \vdash_D P'$ with $\mathsf{fid}_!(\Gamma')$; and

(iv) $\mathsf{one\text{-}role}(P')$.

Proof. By induction on the derivation of $\Gamma\!\rightarrow$. There are two possible cases by which Γ can be enabled with a reduction: by rule Γ-Com$_1$ or by rule Γ-Com$_2$. The following elaborates on the latter case (the other case is similar).

The assumptions are: **(A1)** $\Gamma \vdash_D P$; **(A2)** $\mathsf{fid}_!(\Gamma)$; **(A3)** $\mathsf{one\text{-}role}_!(P)$; and **(A4)** $\Gamma\!\rightarrow$.

The proof begins by inferring some structure on both the types and processes, utilizing lemma 4.25.

$P \equiv \Big|_{p\in I} P_p$ and $\Gamma = \Gamma_0(\cdot\, \Gamma'_p, s[p] : U_p)_{p\in I'}$ (by (A1), (A3), and lemma 4.25) (1)

$\forall p \in I' : \Gamma'_p, s[p] : U_p \vdash_D P_p$ and $c : U \notin \Gamma'_p$ (by (A1), (A3), and lemma 4.25) (2)

$\forall q \in I \setminus I' : P_q \equiv \mathbf{0}$ (by (A1), (A3), and lemma 4.25) (3)

$c : U \notin \Gamma_0$ (by (A1), (A3), and lemma 4.25) (4)

Since the context is known to be enabled with a reduction (by assumption), the actions emitted by the context can be inferred from rule Γ-Com$_2$. Specifically, $\exists p, q \in I'$ such that:

$\Gamma'_p, s[p] : U_p \xrightarrow{s:p\oplus q:\mathsf{m}(\vec{r},\vec{T}_1)} \Gamma'_p, s[p] : U'_p$ (by (1), (A4) and Γ-Com$_2$) (5)

$\Gamma'_q, s[q] : U_q \xrightarrow{s:q\&\alpha':\mathsf{m}(\vec{\alpha},\vec{T}_2)} \Gamma'_q, s[q] : U'_q$ (by (1), (A4) and Γ-Com$_2$) (6)

$\vec{T}_1 \leqslant \vec{T}_2$ (by (1), (A4) and Γ-Com$_2$) (7)

$(\Gamma'_p, s[p] : U_p) \cdot (\Gamma'_q, s[q] : U_q) \xrightarrow{s:p,q:\mathsf{m}} (\Gamma'_p, s[p] : U'_p) + (\Gamma'_q, s[q] : U'_q\{\vec{r}/\vec{\alpha}\}\{p/\alpha'\})$
(by (5), (6), (7), and Γ-Com$_2$) (8)

From these actions, the shapes of the types can be outlined, following from the type semantics in Figure 4.7.

$\oplus_{j\in J} q_j{:}\mathsf{m}_j(\vec{T}_j).S_j \leqslant U_p \leqslant q \oplus \mathsf{m}(\vec{r},\vec{T}_1).S$ (by (5), Γ-$\oplus$) (9)

$\exists l \in J : q_l = q \wedge \mathsf{m}_l = \mathsf{m} \wedge \vec{T}_l \geqslant \vec{T}_1$ (by (9), Definition 4.5) (10)

$!\&_{k\in K} \alpha'{:}\mathsf{m}'_k(\vec{\delta}_k,\vec{T}'_k).S'_k \leqslant U_q$ and $\exists l \in K : \mathsf{m}'_l = \mathsf{m}$ (by (6), Γ-!) (11)

$\vec{T}'_l \leqslant \vec{T}_2$ (by (11), (6), Definition 4.5) (12)

The shapes of processes can now be inferred from the types, using (2), (9), lemma 4.26 for P_p, and (2), (11), lemma 4.26 for P_q. Note that by the session inversion lemma, there are two possible shapes for each process; thus a total of four possible permutations to consider, resulting in 4 subcases.

Case $\oplus$!: Consider:

$P_p \equiv \sum_{j\in J'} s[p][q_j] \oplus \mathsf{m}_j\langle\vec{V}_j\rangle.P'_j$ and $J \supseteq J'$ (by case assumption) (13)

$P_q \equiv !s[q][\alpha']\&_{k\in K'}\mathsf{m}'_k(\vec{b}_k).P''_k$ and $K \subseteq K'$ (by case assumption) (14)

This is where the novel fidelitous property comes into play. As it stands, there is no guarantee that $\exists x \in J' : q_x = q$, as the implementation is a subset of the type and could hence implement a part of the selection that does not include the role considered for communication in (8). However, by (8), it is guaranteed that at least one (type-level) communication is possible. Then by (A2) and table 4.1, it follows that communication should be eventually possible for every path in J, and thus also J' (since $J' \subseteq J$ by (13)). Therefore, it can be assumed without loss of generality that the nondeterministic choice in P_p reduces to a path x in J' such that $q_x = q$.

$P_p \to P'_p \equiv s[p][q] \oplus \mathsf{m}_x\langle \vec{V}_x \rangle.P'_x$ and $x \in J'$ (explanation above) (15)

$\exists y \in K : \mathsf{m}_y = \mathsf{m}_x$ (by (15), (A2), Definition 4.22) (16)

$y \in K'$ (by (16) and (14)) (17)

$P'_p \mid P_q \to P'_x \mid P_q \mid P''_y\{p/\alpha'\}\{\vec{V}_x/\vec{b}_y\}$ (by (14), (15), (17), R-!C$_2$) (18)

$\Gamma'_p, s[p] : U_p \xrightarrow{s:p\oplus q:\mathsf{m}_x(\vec{r'},\vec{T}_x)} \Gamma'_p, s[p] : U'_p$ (by (2), (15), Γ-$\oplus$) (19)

$(\Gamma'_p, s[p] : U_p) \cdot (\Gamma'_q, s[q] : U_q) \xrightarrow{s:p,q:\mathsf{m}_x} (\Gamma'_p, s[p] : U'_p) + (\Gamma'_q, s[q] : U'_q\{\vec{r'}/\vec{\delta}\}\{p/\alpha'\})$
(by (19), (8), (A2)) (20)

$\Gamma'_q, s[q] : U_q \xrightarrow{s:q\oplus\alpha':\mathsf{m}_x(\vec{\delta},\vec{T}'_x)} \Gamma'_q, s[p] : U'_q$ (by (20), (19), Γ-Com$_2$) (21)

$\vec{T}_x \leqslant \vec{T}'_x$ (by (21), (20), (19), Γ-Com$_2$) (22)

The subcase is concluded by observing that the reduced contexts type the reduced processes, and that the reduced processes still adhere to the one-role$_!$ requirements.

$\Gamma'_p, s[p] : U'_p \vdash_D P'_x$ (by (2), (19), T-$\oplus$) (23)

$\Gamma'_q, s[p] : U'_q\{p/\alpha'\}\{\vec{r'}/\vec{\delta}\} \vdash_D P''_y\{p/\alpha'\}\{\vec{V}_x/\vec{b}_y\}$
(by (2), (21), (22), (23), T-!, and both substitution lemmas) (24)

Note that for the above, all the types for payloads are present in Γ'_q since by the session fidelity assumptions, payloads can only contain non-session-typed entities; these are copied through context splits (Figure 4.5).

$P \to^* P' \mid P'_x \mid P_q \mid P''_y\{p/\alpha'\}\{\vec{V}_x/\vec{b}_y\}$ with one-role$_!(P')$ (by (A3), (15), (18)) (25)

one-role$_!(P'_x \mid P_q \mid P''_y\{p/\alpha'\}\{\vec{V}_x/\vec{b}_y\})$ (by (1), (23), (24), Definition 4.24) (26)

$\Gamma \to \Gamma'$ (by (20)) (27)

$P \to^* P' \mid P'_x \mid P_q \mid P''_y\{p/\alpha'\}\{\vec{V}_x/\vec{b}_y\}$ (by (25)) (28)

$\Gamma' \vdash_D P' \mid P'_x \mid P_q \mid P''_y\{p/\alpha'\}\{\vec{V}_x/\vec{b}_y\}$ (by (2), (20), (23), (24), T-|) (29)

fid$_!(\Gamma')$ (by (A2), (27), φ-$\to$) (30)

one-role$_!(P' \mid P'_x \mid P_q \mid P''_y\{p/\alpha'\}\{\vec{V}_x/\vec{b}_y\})$ (by (25), (26)) (31)

Case $\oplus!$ holds. (by (25), (27)–(31)) (32)

The remaining subcases all follow similar reasoning, except with the additional step of allowing for process reduction via rule R-X before communication becomes available.

□

4.3.5 Behavioural Properties

A benefit of integrating replication and first-class roles into a generalised MPST calculus is to verify behavioural properties on client-server systems where finite clients may communicate with infinitely available servers. The definitions of certain properties for such systems need to be relaxed from those specified in Definition 3.24. For instance, under to the previous definitions, any program with a replicated server is considered deadlocked, since it can never reduce to $\mathbf{0}$. Thus, Definition 4.28 redefines behavioural properties of MPST! processes, allowing for the infinite nature of replication. Using these definitions, the property verification theorem for MPST! is given in Theorem 4.29.

Definition 4.28 (Process properties). A process P is said to be:

deadlock-free (df) iff $P \to_D^* P' \not\to_D$ implies either

(1) $P' \equiv \mathbf{0}$; or
(2) $P' \equiv \big|_{i\in I} !P'_i$;

terminating (term) iff $P \to_D^* P'$ implies $\exists n$ finite s.t. $P' \to_D^n P''$ and either

(1) $P'' \equiv \mathbf{0}$; or
(2) $P'' \equiv \big|_{i\in I} !P''_i$;

never-terminating (n-term) iff $P \to_D^* P'$ implies $P' \to_D$;

lock-free (lf) iff $P \to_D^* P'$ implies:

(1) if $P' \equiv c[\boldsymbol{q}] \oplus \mathsf{m}\langle\vec{V}\rangle . Q \mid Q'$, then $P' \to_D^* P'' \equiv Q \mid Q''$; and
(2) if $P' \equiv c[\boldsymbol{q}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) . Q_i \mid Q'$, then $\exists k \in I, \vec{V} : P' \to_D^* Q_k\{\vec{V}/b_k\} \mid Q''$;

live (live) iff $P \to_D^* P'$ implies $\exists n$ finite s.t.:

(1) if $P' \equiv c[\boldsymbol{q}] \oplus \mathsf{m}\langle\vec{V}\rangle . Q \mid Q'$ and $P' \to_D^n P'_n$, then $\exists j \leq n : P'_j \to_D P'' \equiv Q \mid Q''$; and
(2) if $P' \equiv c[\boldsymbol{q}] \,\&_{i\in I}\, \mathsf{m}_i(\vec{b}_i) . Q_i \mid Q'$ and $P' \to_D^n P'_n$, then $\exists j \leq n, k \in I, \vec{V} : P'_j \to_D P'' \equiv Q_k\{\vec{V}/b_k\} \mid Q''$.

The updates in Definition 4.28 are to deadlock-free and terminating processes. A deadlock-free process is now one which only stops reducing when it reaches $\mathbf{0}$, *or* when it only contains replicated servers (since they are designed to remain infinitely available). Terminating processes reflect the same changes. Importantly, lock-free and live processes are unchanged. It would not make sense to impose the eventual communication requirement on replicated servers, as it is intentional that they may expect messages that may possibly never arrive (e.g. after servicing all client requests).

Theorem 4.29 (Property verification). If $\Gamma \vdash_D P$ with $\mathsf{fid}_!(\Gamma)$ and $\mathsf{one\text{-}role}_!(P)$, then $\phi(\Gamma) \implies \phi(P)$ for $\phi \in \{\textit{deadlock-free, terminating, never-terminating, lock-free, live}\}$.

Proof. Observe that by Theorem 4.27, Γ simulates the *at most* behaviour of process P. Therefore, the trace observed at runtime for a program is guaranteed to be simulated by a path on the type context. Hence, if the sequences of actions for all paths of a type context adhere to some property, then that property holds for processes typed under that context. Thus, to prove this theorem, what is required is to show that the definitions for type-level properties soundly capture the process-level equivalents. This is demonstrated for deadlock-free processes below (other properties follow similar reasoning).

Case df: RTP (under the theorem's assumptions) $\mathsf{df}(\Gamma)$ implies $\mathsf{df}(P)$.

Note that if $\Gamma\rightarrow$, then $P \rightarrow$, hence P would be trivially deadlock-free. Thus, consider $\Gamma\not\rightarrow$. In this case, by table 4.1, it can be concluded that $\mathsf{end!}(\Gamma)$. Then, by Definition 4.15 and session inversion (lemma 4.26), either $P \equiv \mathbf{0}$, or $P \equiv \big|_{i\in I}\, !P_i'$, or $P \rightarrow^* P'' \equiv \big|_{i\in I}\, !P_i''$ via a finite series of applications of R-X. (It is guaranteed that the number of reductions via R-X is finite by Definition 4.23 and since $\mathsf{one\text{-}role}_!(P)$ by assumption.)

□

4.4 Expressivity and Decidability

This section discusses, and shows by example, the benefits and limitations of MPST!. Section 4.4.1 focuses on demonstrating the exressivity gained by using replication and first-class roles in MPST, whilst Section 4.4.2 discusses decidability.

4.4.1 Expressivity

To begin, the following demonstrates a common design pattern used for describing protocols, referred to in this thesis as *services*. A number of services are built to showcase language features, and to describe protocols which were previously untypable in any MPST theory. Specifically, using the increased expressiveness of replication and first-class roles, protocols are defined for communicating serialised *binary trees*, the *dining philosophers problem*, and an *auction*. Importantly, all examples shown adhere to the decidability requirements discussed later (cf. Section 4.4.2).

Services.

A *service* is a building-block of a protocol, involving some universal receive, with the aim of *offering* a specific interaction. A *client* interacts with a service to achieve the communication pattern it offers. Importantly, services may be clients of other services, promoting a modular

design of protocols in MPST! and adhering to the client-server paradigm [104] mentioned in Section 4.1.

Example 4.30 (Ping). The *ping service* simply responds to a ping with a pong.

$$P : !\alpha\&\mathsf{ping}.\alpha\oplus\mathsf{pong}.\mathbf{end}$$

A basic yet useful service is given in Example 4.30, where role P offers a *ping* service. As a convention, capitals are used for naming services. It is key to note the use of both replication and free role names in types to be able to design modular components of a protocol—both are integral to designing a sub-protocol agnostic of the larger scope in which it is used. The following showcases how a service can be used as a building-block for larger protocols, and also serves as witness for the increased expressiveness gained in MPST!.

Context-Free MPST.

Context-free session types [108, 3, 92] are a formalism that replace prefix-style session types with individual actions, along with a sequencing operator ; with neutral element skip. The goal of this line of work is to express communication protocols that are not possible under tail-recursive session types, given their restriction to regular languages. The classic example is that of communicating a serialised binary tree.

Example 4.31 (Binary tree in standard context-free STs). Consider a binary tree data type described by the following context-free grammar.

$$\mathsf{tree} ::= (\mathsf{node}, \mathsf{tree}, \mathsf{tree}) \mid \mathsf{leaf}$$

An attempt at representing this data type in a session protocol follows:

$$\mu \mathrm{t}.q \oplus \{\mathsf{leaf}.\mathbf{end},\ \mathsf{node}.\mathrm{t}\}$$

However, this type is not precise enough—it does not guarantee that the correct structure of a binary tree is maintained. Work on context-free session types solves this by proposing type systems allowing for specifications similar to:

$$\mu \mathrm{t}.q \oplus \{\mathsf{leaf}.\mathsf{skip},\ \mathsf{node}.\mathsf{skip}\,;\mathrm{t}\,;\mathrm{t}\}$$

Selecting the node label instead guarantees that the type requires the program to send *two* sub-trees.

The parallel types presented in Section 4.2.3, although not exposed directly to users, lift expressiveness of types in MPST!. In fact, since replicated branches are *permanently available* (by composing continuation types in parallel), the sequencing operator **;** can be simulated using type-level parallel composition.

Example 4.32 (Binary Tree Service). Recall the ping service P from Example 4.30. A *binary tree service* T is built using P as shown below:

$$T : !\beta\&\mathsf{tree}.P\oplus\mathsf{ping}.!P\&\mathsf{pong}.\beta\&\{\mathsf{leaf}.\mathbf{end},\ \mathsf{node}.P\oplus\mathsf{ping}.P\oplus\mathsf{ping}.\mathbf{end}\}$$

The service begins by receiving a request for a tree from a client. It then sends a ping to the ping service, exposing a replicated branch waiting to receive the pong reply. The client is now free to build the binary tree. It is key to note that any node sent to the service will subsequently forward *two* ping requests to P. In turn, this communication will pull out two copies of the type continuation $\beta\&\{\mathsf{leaf}.\mathbf{end},\ \mathsf{node}.P\oplus\mathsf{ping}.P\oplus\mathsf{ping}.\mathbf{end}\}$, forcing the client to maintain the appropriate binary tree structure. For example, if a client t wishes to build a tree consisting of one root node and two leaf nodes, its type would be defined as:

$$t : T\oplus\mathsf{tree}.T\oplus\mathsf{node}.T\oplus\mathsf{leaf}.T\oplus\mathsf{leaf}.\mathbf{end}$$

The metatheoretic framework can be used to determine that any protocol failing to abide by the binary tree structure will result in a *deadlock*; whilst any fidelitous protocol that obeys the correct structure is *terminating*, e.g. the protocol $\{t, T, P\}$.

An obscure limitation of the tree service in Example 4.32 is that it can only be used by a single client. Consider, for example, two separate clients sending a node message to T. Since both tree service types communicate with P to unroll the replicated branch $!P\&\mathsf{pong}$, the protocol becomes non-deterministic in a non-confluent manner and can result in deadlocked behaviour. To resolve this issue, the tree service is amended to accept a payload role acting as a personal ping service for the client; this guarantees that the tree type is only unrolled by the client that made the initial request.

Example 4.33 (Multi-Client Binary Tree). The binary tree service is now redesigned, this time capable of concurrently building multiple trees for different clients. The key difference being that the new service, M, accepts a role as a payload on the initial request

to which it will issue its pings.

$$M : !\alpha\&\mathsf{tree}(\beta).\beta\oplus\mathsf{ping}.!\beta\&\mathsf{pong}.\alpha\&\{\mathsf{leaf}.\mathbf{end},\mathsf{node}.\beta\oplus\mathsf{ping}.\beta\oplus\mathsf{ping}.\mathbf{end}\}$$

$$S_p = !M\&\mathsf{ping}.M\oplus\mathsf{pong}.\mathbf{end}$$

Multiple clients can now issue concurrent requests to the tree service whilst maintaining safety and fidelity. A sample (terminating) protocol is that of $\{t_1, t_2, p_1, p_2, M\}$, where $p_1, p_2 : S_p$, and the types for t_1, t_2 are given by:

$$t_1 : M\oplus\mathsf{tree}(p_1).M\oplus\mathsf{node}.M\oplus\mathsf{leaf}.M\oplus\mathsf{leaf}.\mathbf{end}$$
$$t_2 : M\oplus\mathsf{tree}(p_2).M\oplus\mathsf{node}.M\oplus\mathsf{leaf}.M\oplus\mathsf{node}.M\oplus\mathsf{leaf}.M\oplus\mathsf{leaf}.\mathbf{end}$$

Replication vs. Recursion.

As demonstrated, replication and parallel composition increase the expressive power of MPST beyond that of tail-recursion. Naturally, one might ask, "*is recursion still needed?*" Interestingly, this study finds replication and recursion in MPST to be mutually non-inclusive—i.e., both can produce protocols which *cannot* be typed under the other construct. This has already been demonstrated in one direction with the binary tree examples; the following showcases how recursion cannot be replaced by replication.

Example 4.34 (Lock Service). The lock service provides clients with a *mutex lock*.

$$L : !\theta\&\mathsf{lock}.\mu\mathsf{t}.\theta\&\{\mathsf{acquire}.\theta\&\mathsf{release}.\mathsf{t},\ \mathsf{done}.\mathbf{end}\}$$

When a client requests a lock from L, a copy of the recursive continuation is exposed. The recursive definition allows sequences of acquire and release messages to be received. It is key to note that, whilst replication maintains a top-level branch that is permanently available to receive a message, the top-level action in a recursive definition is *not* fixed.

Copies of the continuation type of a replicated receive are executed concurrently. Example 4.34 provides a service for roles to enter race-sensitive portions of a protocol, as if it were an atomic action. This can be used to type the dining philosophers problem.

Example 4.35 (Dining Philosophers). A number of philosophers gather to eat on a round table. Each plate is separated by a single chopstick, and a philosopher needs two chopsticks to eat. The dining philosophers problem requires the philosophers to employ

an algorithm to ensure the table does not get deadlocked. It is possible to view *chopsticks as services* and *philosophers as clients*. The types for n chopsticks are given by:

$$(C_i)_{i\in 1..n} : L\oplus\mathsf{lock}\,.\,!\eta\,\&\left\{\begin{array}{l}\mathsf{take?}\,.\,L\oplus\mathsf{acquire}\,.\,\eta\oplus\mathsf{ok}\,.\,\eta\,\&\mathsf{give}\,.\,L\oplus\mathsf{release}\,.\,\mathbf{end}\\ \mathsf{done}\,.\,L\oplus\mathsf{done}\,.\,\mathbf{end}\end{array}\right\}$$

Before offering its service, a chopstick requests a lock from L. This ensures that every chopstick has its own lock that it may acquire and release. The lock is used to guarantee that a chopstick is only ever taken by a single philosopher at a time. A chopstick then waits for a take? request from a philosopher; receiving one will result in it attempting to acquire the lock. This acquisition is only successful if the same role has not already requested it in some other parallel composition. If the lock was already acquired, then the $L\oplus\mathsf{acquire}$ will block until the lock is released. Acquiring the lock sends an ok back to the philosopher, symbolising that they have successfully obtained the chopstick. When done from eating, the philosopher may then send back a give message, which in turn releases the lock, as the chopstick is now available to be taken by a different philosopher. The naïve algorithm for n philosophers may be described as follows:

$$\begin{aligned}(p_i)_{i\in 1..n} &: C_i\oplus\mathsf{take?}\,.\,C_{i+1}\oplus\mathsf{take?}\,.\,C_i\,\&\mathsf{ok}\,.\,C_{i+1}\,\&\mathsf{ok}\,.\,C_i\oplus\mathsf{give}\,.\,C_{i+1}\oplus\mathsf{give}\,.\,q\oplus\mathsf{fin}\,.\,\mathbf{end}\\ q &: p_1\,\&\mathsf{fin}\,.\,\cdots\,.\,p_n\,\&\mathsf{fin}\,.\,C_1\oplus\mathsf{done}\,.\,\cdots\,.\,C_n\oplus\mathsf{done}\,.\,\mathbf{end}\end{aligned}$$

Every philosopher p_i has a similar type. They begin by requesting to take the chopsticks to their left and right—note that this results in every chopstick receiving two take? requests. Receiving both ok messages means the philosopher can eat, and subsequently give back the chopsticks. Finally, when finished, philosophers send a fin to role q, acting as a clean-up for the protocol. The protocol $\{p_i, q, C_i, L\}_{i\in 1..n}$ is fidelitous, but fails typechecking for $\varphi = \mathit{terminating}$. In fact, the naïve protocol allows for the scenarios in which all philosophers take a single chopstick, resulting in a deadlock. This problem has many solutions; the following presents the simplest, in which philosophers take turns to eat. (Key changes are underlined.)

$$\begin{aligned}S_1 &= C_1\oplus\mathsf{take?}\,.\,C_2\oplus\mathsf{take?}\,.\,C_1\,\&\mathsf{ok}\,.\,C_2\,\&\mathsf{ok}\,.\,C_1\oplus\mathsf{give}\,.\,C_2\oplus\mathsf{give}\,.\,\underline{p_2\oplus\mathsf{fin}}\,.\,\mathbf{end}\\ S_2 &= \underline{p_{i-1}\,\&\mathsf{fin}}\,.\,C_i\oplus\mathsf{take?}\,.\,C_{i+1}\oplus\mathsf{take?}\,.\,C_i\,\&\mathsf{ok}\,.\,C_{i+1}\,\&\mathsf{ok}\,.\,C_i\oplus\mathsf{give}\,.\,C_{i+1}\oplus\mathsf{give}\,.\,\underline{p_{i+1}\oplus\mathsf{fin}}\,.\,\mathbf{end}\\ S_3 &= \underline{p_{n-1}\,\&\mathsf{fin}}\,.\,C_n\oplus\mathsf{take?}\,.\,C_1\oplus\mathsf{take?}\,.\,C_n\,\&\mathsf{ok}\,.\,C_1\,\&\mathsf{ok}\,.\,C_n\oplus\mathsf{give}\,.\,C_1\oplus\mathsf{give}\,.\,q\oplus\mathsf{fin}\,.\,\mathbf{end}\\ q' &: p_n\,\&\mathsf{fin}\,.\,C_1\oplus\mathsf{done}\,.\,\cdots\,.\,C_n\oplus\mathsf{done}\,.\,\mathbf{end}\end{aligned}$$

Here, all philosophers other than the first must wait for the previous to finish eating before they can request to take their chopsticks. The updated protocol $\{p_1 : S_1, p_i : S_2, p_n :$

$S_3, q', C_j, L\}_{j\in 1..n}^{i\in 2..n-1}$ now typechecks for $\varphi = \textit{terminating}$.

The previous examples demonstrate how recursion hidden by a universal receive can be used to mimic changes in state. The final example does the inverse, i.e., a universal receive hidden by a recursive binder can be used to model resources which eventually reach some permanent state. In addition, the example demonstrates how universal receives model *fair races*, since they do not impose an order on how communication is handled.

Example 4.36 (Auction). A merchant m sets up an auction A to accept bids from some buyers b. A merchant can employ different mechanisms for choosing who to sell to (e.g., first come first served, highest bid, biased selling, etc.); but must always respond to buyers with either a yes, no, or not-avail message.

$$A : !\alpha\&\mathsf{bid}(\mathtt{Int}).m\oplus\mathsf{bid}(\mathtt{Int},\alpha).\mathbf{end}$$

$$m : \mu t.A\&\mathsf{bid}(\mathtt{Int},\beta).\beta\oplus\left\{\begin{array}{l}\mathsf{yes}.!A\&\mathsf{bid}(\mathtt{Int},\kappa).\kappa\oplus\mathsf{not\text{-}avail}.\mathbf{end}\\ \mathsf{no}.t\end{array}\right\}$$

$$(b_i)_{i\in 1..n} : \mu t.A\oplus\mathsf{bid}(\mathtt{Int}).m\&\{\mathsf{yes}.\mathbf{end},\ \mathsf{no}.t,\ \mathsf{not\text{-}avail}.\mathbf{end}\}$$

Buyers b_i race to send bids to the auction service. In turn, the auction forwards bids and buyer role identifiers to the merchant, who processes bids sequentially (but still in an arbitrary order). If the merchant declines a bid, then the client is offered another chance; if the merchant accepts a bid, then it exposes a replicated receive which informs any further buyers that the product is no longer available. It is key to note that, unlike in Example 4.35 where locks are used to avoid race conditions, races here are not only allowed but are integral to the protocol. Additionally, by uncovering a replicated receive, the merchant enters a *permanent* state. These two characteristics guarantee that, no matter the selling algorithm employed by the merchant: *(i)* bids always arrive in a fair arbitrary order; and *(ii)* the product can only be sold once.

Discussion.

Replication and *recursion* have been shown to be mutually non-inclusive in MPST, and that the studied extension increases the expressiveness of the type system. It is important to understand the dependencies between added features and the expressiveness gained. Since MPST! is a *conservative* extension, it is guaranteed that the increase of expressiveness derives from the two extensions: *1.* the addition of *replication*; and *2.* the addition of *first-class roles*.

Replication alone is enough to increase the expressiveness of MPSTs w.r.t. the Chomsky hierarchy. Note that, e.g., Example 4.32 could be easily re-written without the universal receive, especially since it should not be used by multiple clients to uphold deadlock-freedom—thus, *replication in MPSTs increases their expressiveness to that of context-free languages*.

"First-class roles" in the formalism refers to: *(i) universal receives* acting as binders on role variables; and *(ii)* the ability to *pass roles* as payloads in messages. Universal receives allow *protocols to be designed agnostic of the client pool* (consider Examples 4.2, 4.30, 4.33 and 4.34); and also act as a *fair way of introducing races*—e.g. Example 4.2 describes a load balancer that responds to requests in *no particular order*. Role passing allows for *safe distributed choice*. In a load balancer (in general), it is *impossible* for a client to know which worker will service its request. In Example 4.2, role passing allows the server to inform clients of its choice, and also informs the worker of the identity of the client. Role passing increases expressiveness by introducing dependencies into a protocol. For instance, Example 4.36 uses role passing to ensure the merchant correctly services the right buyers; without it, a merchant could not respond to requests without bias.

As a final note, first-class roles are different to, e.g., *delegation* or *multiple sessions* (both supported in MPST!) since they act *inside* a session. Therefore these constructs can be used whilst abiding by the session fidelity assumptions, and thus the generalised framework can still be used to verify runtime properties. This not possible with interleaved sessions without other mechanisms such as an interaction typing system [14] or priorities [31].

4.4.2 Decidability

As may be expected, the added expressiveness of replication and first-class roles into types does not come without a cost. Unlike the core calculus, the type semantics for MPST! does not simulate a finite state machine, and so decidability results are not obvious. The remainder of this section discusses decidability of typechecking in detail. Typechecking is shown to be *only as decidable as the safety property*; examples of problematic types are given; decidable sublanguages of MPST! are identified; and strategies for restricting protocols to these decidable sublanguages are given.

Theorem 4.37 (Decidablility of type checking). If φ is decidable, then typechecking is decidable.

Proof. Since typing rules in Figure 4.6 can be deterministically applied based on the structure of a process P, and a typing context need only be split a finite number of times to separate all linear types, there are a finite number of contexts that can be tried for each rule that requires a context split. Lastly, subtyping is decidable [45] (decidability of

subtyping replicated types is equivalent to regular branching types, and of parallel types is equivalent to checking multiple session types); and φ is decidable by assumption. □

Theorem 4.37 states that decidability of type checking is only as decidable as property φ. Naturally, the next step is thus to determine the decidability of φ. Example 4.40, will demonstrate why φ may not necessarily be decidable in general for the type semantics presented in Figure 4.7. To do this, the following first defines *behavioural sets* of type contexts (as in [102, appendix K]).

Definition 4.38 (Behavioural set). The behavioural set of a type context, written $\mathsf{beh}(\Gamma)$, is given by $\mathsf{beh}(\Gamma) = \mathsf{unf}^*(\{\Gamma' | \Gamma \rightarrow^* \Gamma'\})$; where unf^* is the closure of unf—a function that unfolds all top-level recursive binders in a set of contexts.

Informally, the behavioural set of a context Γ is the set of *(i)* its reductions; and *(ii)* its reductions' unfoldings. The benefit of beh is that it mechanically abides by conditions φ-μ and φ-$\rightarrow$ from Definition 4.20. Condition φ-$\boldsymbol{\rho}$ can be checked by traversing through the syntax of the (finite) type context, as per Figure 4.8. Therefore, to determine whether $\mathsf{beh}(\Gamma)$ is a safety property, all that remains is to exhaustively check the contexts that inhabit $\mathsf{beh}(\Gamma)$ against the remaining safety conditions.

Example 4.39. Consider a context $\Gamma = \{s[\boldsymbol{p}] : \mu \mathrm{t}.\boldsymbol{q} \oplus \mathrm{m}.\mathrm{t}, s[\boldsymbol{q}] : \mu \mathrm{t}'.\boldsymbol{p} \& \mathrm{m}.\mathrm{t}'\}$. The behavioural set of Γ is given by:

$$\mathsf{beh}(\Gamma) = \left\{ \left\{ \begin{array}{l} s[\boldsymbol{p}] : \mu \mathrm{t}.\boldsymbol{q} \oplus \mathrm{m}.\mathrm{t}, \\ s[\boldsymbol{q}] : \mu \mathrm{t}'.\boldsymbol{p} \& \mathrm{m}.\mathrm{t}' \end{array} \right\}, \left\{ \begin{array}{l} s[\boldsymbol{p}] : \boldsymbol{q} \oplus \mathrm{m}.\mu \mathrm{t}.\boldsymbol{q} \oplus \mathrm{m}.\mathrm{t}, \\ s[\boldsymbol{q}] : \boldsymbol{p} \& \mathrm{m}.\mu \mathrm{t}'.\boldsymbol{p} \& \mathrm{m}.\mathrm{t}' \end{array} \right\} \right\}$$

The left element is the original context after 0 reduction steps, whereas the right element is the unfolding of Γ. Moreover, any further reductions only yield contexts (and unfoldings) already captured by these two elements.

The next example context, however, is problematic for typechecking.

Example 4.40. Consider a context $\Gamma = \{s[\boldsymbol{p}] : \mu \mathrm{t}.\boldsymbol{q} \oplus \mathrm{m}.\mathrm{t}, s[\boldsymbol{q}] : !\boldsymbol{p} \& \mathrm{m}.\boldsymbol{r} \oplus \mathrm{m}\}$. The be-

havioural set of Γ is given by:

$$\mathsf{beh}(\Gamma) = \left\{ \left\{ \begin{array}{l} s[p] : \mu \mathrm{t}.q \oplus \mathrm{m}.\mathrm{t}, \\ s[q] : !p \& \mathrm{m}.r \oplus \mathrm{m} \end{array} \right\}, \left\{ \begin{array}{l} s[p] : q \oplus \mathrm{m}.\mu \mathrm{t}.q \oplus \mathrm{m}.\mathrm{t}, \\ s[q] : !p \& \mathrm{m}.r \oplus \mathrm{m} \end{array} \right\}, \right.$$
$$\left. \left\{ \begin{array}{l} s[p] : \mu \mathrm{t}.q \oplus \mathrm{m}.\mathrm{t}, \\ s[q] : !p \& \mathrm{m}.r \oplus \mathrm{m} \mid r \oplus \mathrm{m} \end{array} \right\}, \left\{ \begin{array}{l} s[p] : q \oplus \mathrm{m}.\mu \mathrm{t}.q \oplus \mathrm{m}.\mathrm{t}, \\ s[q] : !p \& \mathrm{m}.r \oplus \mathrm{m} \mid r \oplus \mathrm{m} \end{array} \right\}, \right.$$
$$\left. \left\{ \begin{array}{l} s[p] : \mu \mathrm{t}.q \oplus \mathrm{m}.\mathrm{t}, \\ s[q] : !p \& \mathrm{m}.r \oplus \mathrm{m} \mid r \oplus \mathrm{m} \mid r \oplus \mathrm{m} \end{array} \right\}, \ldots \right\}$$

Indeed, $\mathsf{beh}(\Gamma)$ is *infinite*. This is a result of how replication and parallel composition are modelled in Figure 4.7. In fact, the type semantics for replicated communication allows for context reduction to yield *larger* types. Note how in this example, the contexts that inhabit $\mathsf{beh}(\Gamma)$ get infinitely larger by pulling out infinitely many copies of type $r \oplus \mathrm{m}$.

Furthermore, infinite behavioural sets are not only a result of recursive communication with replicated branches. Consider, e.g.:

$$\Gamma' = \{s[p] : !\alpha \& \mathrm{m}.\alpha \oplus \mathrm{m}'.r \oplus \mathrm{m}, s[q] : p \oplus \mathrm{m}.!\beta \& \mathrm{m}'.\beta \oplus \mathrm{m}\}$$

Such a context will also pull out infinitely many copies of type $r \oplus \mathrm{m}$, because the replicated communication forms an infinite loop. Lastly, it is key to note that $\mathsf{beh}(\Gamma'')$ is finite for any Γ'' that does not contain replicated branches, since there is no other way for a context reduction to yield a larger type.

Knowing whether $\mathsf{beh}(\Gamma)$ is (in-)finite is key for the main decidability result.

Theorem 4.41 (Decidability of beh). Let $\varphi = \mathsf{beh}(\Gamma)$. If $\mathsf{beh}(\Gamma)$ is finite, then φ is decidable.

Proof. Since $\mathsf{beh}(\Gamma)$ contains all possible reductums and unfoldings of Γ, then conditions φ-$\rightarrow$ and φ-μ are satisfied immediately. Therefore, to determine whether $\mathsf{beh}(\Gamma)$ is a safety property, all inhabitants of $\mathsf{beh}(\Gamma)$ may be exhaustively checked against conditions φ-S and φ-ρ, which is decidable since $\mathsf{beh}(\Gamma)$ is finite (by assumption); and since subtyping and frv are decidable. □

Theorem 4.41 states that φ is *decidable* for any $\varphi = \mathsf{beh}(\Gamma)$ where $\mathsf{beh}(\Gamma)$ is a finite set. In other words, if a protocol can be shown to have a finite behavioural set, then typechecking for that protocol is *decidable*. This could be done manually for each protocol; however, to further increase the practicality of the type system, two strategies for restricting protocols into a subset of MPST! with finite behavioural sets are presented.

Decidability Strategies.

By Theorems 4.37 and 4.41, typechecking is decidable for the subset of MPST! typed under a context Γ for which $\mathsf{beh}(\Gamma)$ is finite. Hence, the strategies presented determine sound (but not complete) properties guaranteed to restrict types to this decidable subset.

Definition 4.42 prevents types like those in Example 4.40 using a naïve approach; $\mathsf{tf}(\Gamma)$ captures protocols where all clients of a replicated server are non-recursive and non-replicated.

Definition 4.42 (Trivially finite). A context Γ is trivially finite, $\mathsf{tf}(\Gamma)$, iff:

1. no type in the body of a recursive binder sends to a replicated branch; and
2. continuations of replicated branches do not send to other replicated branches.

Example 4.43. The protocols modelling the dining philosophers problem in Example 4.35 are *trivially finite*. Note how the chopstick services make the initial request to the lock service *before* they offer their replicated branch.

The trivially finite property is restrictive. Indeed it nullifies the benefit of the using the client-server paradigm to build modular systems in which servers can act as clients to other servers. For such protocols a more nuanced strategy is required. Definition 4.44 formalises "loops" in a protocol which may result from replicated servers infinitely bouncing messages amongst each other (such as Γ' in Example 4.40).

Definition 4.44 (Loop free). Given a protocol Ψ, and a context Γ derived from Ψ (possibly after a number of reductions), a **cycle** in the LTS of Γ is defined as the series of transitions s.t., for $\Gamma = \Gamma' \cdot s[\boldsymbol{p}] : !\boldsymbol{\rho}\&_{i\in I}\mathsf{m}_i(\vec{T}_i).S_i$

$$\Gamma \xrightarrow{s:\boldsymbol{q},\boldsymbol{p}:\mathsf{m}_k} \left(\xrightarrow{s:\boldsymbol{p}'_j,\boldsymbol{q}'_j:\mathsf{m}'_j} \right)_{j\in 1..n} \xrightarrow{s:\boldsymbol{q},\boldsymbol{p}:\mathsf{m}_k} \Gamma''$$

where $k \in I$, and for any $\boldsymbol{p}', \boldsymbol{q}', \mathsf{m}', n, \Gamma''$. A cyclic replicated communication path (**CRCP**) is defined as a cycle with these added conditions:

1. $\Gamma(s[\boldsymbol{q}]) = S_{\boldsymbol{q}}$ s.t. either $S_{\boldsymbol{q}} \equiv !S^{\&}|U$ or $S_{\boldsymbol{q}}$ appears after a recursive binder in $\Psi(\boldsymbol{q})$, for any $S^{\&}, U$; and
2. $\forall x \in 1..n : \Gamma \xrightarrow{s:\boldsymbol{q},\boldsymbol{p}:\mathsf{m}_k} \left(\xrightarrow{s:\boldsymbol{p}'_l,\boldsymbol{q}'_l:\mathsf{m}'_l} \right)_{l\in 1..x-1} \Gamma''' \xrightarrow{s:\boldsymbol{p}'_x,\boldsymbol{q}'_x:\mathsf{m}'_x}$, it must be that:
   $\Gamma'''(s[\boldsymbol{q}'_x]) = S_{\boldsymbol{q}'_x}$ s.t. either $S_{\boldsymbol{q}'_x} \equiv !S^{\&}|U$ or $S_{\boldsymbol{q}'_x}$ appears after a recursive binder in

$\Psi(q'_x)$.

Γ is said to be **loop free**, written $\mathsf{lf}(\Gamma)$, iff the LTS of Γ does not contain a CRCP.

A *cycle* in the LTS of a context Γ: *(i)* starts with an incoming communication action into a replicated type; *(ii)* performs some intermediary transitions; and *(iii)* ends with the transition that began the cycle. A CRCP is a special case of a cycle, where all intermediary transitions must also be between roles that have a replicated type; or form part of the body of a recursive type. Finally, a context is *loop free* iff its LTS does not produce any CRCPs.

Example 4.45. Contexts Γ and Γ' from Example 4.40 contain CRCPs: Γ contains a CRCP at q with 0 intermediary transitions; and Γ' contains two CRCPs, at q and p, both with 1 intermediary transition forming part of a replicated type.

Example 4.46. The protocols in Examples 4.2, 4.32 and 4.33 are *loop free*: Example 4.2 because there are no cycles; Examples 4.32 and 4.33 because the cycle between the pong branch on T (resp. M) and the ping branch on P (resp. p_1, p_2) includes communication with the (non-replicated/-recursive) client; breaking the CRCP.

Proposition 4.47. If $\mathsf{beh}(\Gamma)$ is infinite, then Γ contains a CRCP.

Proof. From the type semantics (Figure 4.7), the only reductions that can yield *larger* types are communications with replicated branches. Therefore, it follows that for $\mathsf{beh}(\Gamma)$ to be infinite, there must be some reoccurring transitions in the LTS of Γ that repeatedly communicates with a replicated branch. That is, there is a sequence of transitions, repeated infinitely, that results in communication with a replicated type. This is the definition of a CRCP (Definition 4.44). □

Theorem 4.48 (Strategy soundness). Given a context Γ, $\Phi(\Gamma)$ implies $\mathsf{beh}(\Gamma)$ is finite, for $\Phi \in \{\mathsf{tf}, \mathsf{lf}\}$.

Proof. *Case* tf. By contradiction: assume $\mathsf{beh}(\Gamma)$ is infinite, then, by Proposition 4.47, Γ contains a CRCP; but, by Definition 4.44, a CRCP will violate at least one of the conditions for tf in Definition 4.42—contradiction.

Case lf. By contradiction: assume $\mathsf{beh}(\Gamma)$ is infinite, then, by Proposition 4.47, Γ contains a CRCP—contradiction; therefore $\mathsf{beh}(\Gamma)$ is *finite*. □

Approximations. Properties tf and lf are both *decidable* for all protocols *without* first-class roles: tf can be determined via a linear traversal of a type context; and lf can be checked by constructing a directed graph of visited replicated branches in a context, then checking that the graph is acyclic (which is decidable). An approximation is only required for protocols using role variables, since their values can only be known at runtime. One possible approximation is to treat any role variable in a selection type to have the capability of reaching *any other role*.

Example 4.49 (Approximation false negative). Consider the following protocol:

$$p : !\alpha\&\mathsf{m}.\alpha\oplus\mathsf{m}' \qquad q : p\oplus\mathsf{m}.p\&\mathsf{m}'.r\oplus\mathsf{m}'.r\&\mathsf{m} \qquad r : !\beta\&\mathsf{m}'.\beta\oplus\mathsf{m}$$

Although the above is *trivially finite* (p and r do not communicate), it would be *falsely* flagged as *not* tf because α is over-approximated to include r.

It is key to note that false negatives of the approximation are avoidable by *requiring unique branching labels on replicated types*. Furthermore, even with this approximation, all examples presented in this chapter (except Example 4.36) are captured by either lf or tf. With respect to Example 4.36, the presented protocol still yields a finite behavioural set, and thus by Theorem 4.41 and Theorem 4.37, it lives in the subset of MPST! for which typechecking is decidable.

4.5 Related Work

Toninho and Yoshida [110] assess the relative expressiveness of a multiparty session calculus and a process calculus inspired by classical linear logic, showing that MPST calculi allow strictly more expressive process networks (i.e., those that can include cycles). As part of this investigation they explore a limited form of type-level replication that permits a liveness property. However, their system does not consider first-class roles and pre-dates generalised MPST so is guided primarily by global types, and is therefore less expressive than MPST!.

Replicated session types have been used to a limited extent in a wide variety of works on binary session types (e.g., [21, 31, 19, 117]), primarily in works pertaining to Curry-Howard interpretations of propositions as session types, where the exponential modality from linear logic $!A$ is typically linked to replication from the π-calculus. Several further lines of work investigate client-server communication following this correspondence.

Kokke et al. [62] investigate an extension of the logically-inspired HCP calculus [61] with two dual modalities $!_nA$ and $?_nA$ to type a pool of n clients and a replicated server that can

service n requests respectively, and show that their calculus allows nondeterministic behaviour while still preventing deadlocks and ensuring termination. Qian et al. [97] develop CSLL (client-server linear logic) that uses the dual *coexponential* modalities ¡A and ¿A to type servers and client pools respectively, along with rules to merge client pools. The subtle difference between the ¡A modality and the exponential !A being that the former serves type A *only as many times as required according to client requests*. This is similar to how replication in the MPST! type system operates, given that replicated receives only pull out copies of continuations upon communication. Multiple requests induce non-determinism into further reductions; in MPST! this is observed through parallel types, which in the work of Qian et al. [97] is observed through hyperenvironments [61, 37].

Unlike all the aforementioned works, the calculus developed in this chapter focuses on *multiparty* session types, where interacting with a replicated channel spawns a process that *remains in the same session*. This results in the key novelty of the type system, i.e., the semantic interpretation of type-level replication through the use of parallel types.

Marshall and Orchard [76] investigate the effects of adding a *semiring graded necessity* modality (a generalisation of the ! modality) to a session-typed λ-calculus, showing interesting consequences such as replicated servers and multicast communication. However, the type system was developed to target a different class of protocols than to those of MPST!. Specifically, their type system allows for describing protocols that rely on multicasts; a construct that cannot be straightforwardly encoded in MPST!. Meanwhile, it is difficult to see how their approach would scale to the examples described in Section 4.4.

Rocha and Caires [98] introduce CLASS, a process calculus with a correspondence to Differential Linear Logic [36]. CLASS integrates session-typed communication, reference cells with mutual exclusion, and replication. Their calculus guarantees preservation and progress, the proof of the latter property requiring a logical relation. CLASS can encode the dining philosophers problem, making essential use of shared state; in contrast, MPST! does not rely on sharing memory, but rather leverages the interplay between replication and recursion to model locks.

Deniélou et al. [33] introduce parameterised MPST as a means of designing protocols for parallel algorithms. Their formalism allows for parameterisation of participants in the form of $\boldsymbol{c}[i]$, representing the i^{th} client from some group of n clients, for a bound n. The key difference between their formalism and MPST! is that the latter preserves, and allows for the fair handling of, *races*. Parameterised MPST enforce a predetermined prioritisation on the order of communication (thus, Example 4.36 cannot be expressed in that system).

4.6 Conclusion

This chapter presented MPST!, a conservative extension of the core calculus which introduced for the first time *replication* and *first-class roles* into MPST, and proved its metatheory. The

language demonstrates that the interplay between replication and recursion in MPST allows for typing interesting and previously inexpressible protocols, such as those that rely on races and mutual exclusion, as well as giving a method for expressing context-free protocols. Although replication could have implications for decidability of typechecking, sublanguages of MPST! have been identified for which typechecking is decidable and are still expressive enough to capture the examples given. Therefore, this chapter has so far addressed some aspects of the thesis statement—i.e., MPSTs have been extended to express practical distributed protocols (specifically client-server protocols). However, the communication model used by MPST! is non-representative of what is observed in practice for designing distributed protocols at low levels on the network stack. Hence, replication and first-class roles are for now parked; instead, the next chapter addresses an extension, and modifications, to the core calculus to more accurately model *failure-prone* communication.

Chapter 5

Nihilism: Failure-Prone Communication

5.1 Introduction

The magpie is a bird with deep ties to British folklore. The first known mention of their counting for fortune telling dates back to 1780, where John Brand writes what is thought to be one of the original versions of the magpie rhyme [17]:

"*One for sorrow, Two for mirth, Three for a funeral, And four for a birth.*"

The natural reaction of a person who spots a solitary magpie may be to scan the surrounding area for its companion. Alas, if no one is immediately visible, the person desperately waits—hoping a second magpie comes their way. But how long should one wait? The reality is that it is *impossible* to know the difference between *no magpie* and a magpie that has *not yet arrived*. To computer scientists, this is a well known *impossibility result* [2]. In the study of *distributed systems* and *fault tolerance*, mechanisms must be employed to approximate the impossibility result of determining whether a message has been *lost* or *delayed*—e.g. by using a *timeout*. Hence, the computer scientist who spots a lonely magpie knows to only wait some fixed amount of time before *assuming* that no other magpie is coming and accepting their sorrowful fate. This mechanism is sometimes referred to as an *imperfect failure detector*, and is the core principle of the language presented in this chapter: MAGπ.

To date, most session type theories are designed for *concurrent*, as opposed to (failure-prone) *distributed*, processes—i.e., it is commonly assumed that communication failures do not occur. For the few (and rapidly increasing) works that do consider failures, heavy assumptions are made that impede their viability for realistic distributed protocols. For instance, *asynchronous* theories [75, 57, 103] use *message queues* to model distributed communication under "TCP-like" assumptions: messages are guaranteed to be delivered and messages from a single sender do not get reordered. *Affine sessions* [84, 38, 22] focus more on how to safely *cancel* or *replace* a protocol based on errors that may be observed at runtime, rather than design protocols revolved around the expectation of some communication errors. *Coordinator model* approaches [1, 25,

115] assume some degree of reliability, be it as a central resilient process, a reliable broadcast, or fixed synchronisation points. The protocols this thesis aims to formalise lie outwith these assumptions.

This chapter presents MAGπ: the first Multiparty, Asynchronous and Generalised π-calculus using *nondeterministic timeouts* as *imperfect failure detectors*. The language models *granular* communication failures at the level of messages, i.e., *message loss, delay, and reordering*; allowing higher level failures to be modelled through sequences of these granular failures (e.g. link failures may be represented by dropping all messages between two participants). The language and type system is parametric on a novel notion of *reliability*, allowing users to specify the failure-prone communication links within a network on a per-program basis.

Example 5.1 (Failure-Prone Load Balancer). Recall the deadlock free load balancer from Example 4.1. The protocol involves four roles: a client c, the server srv, and two workers w_1 and w_2. To demonstrate the configurable notion of reliability in MAGπ, the server and workers are assumed to have reliable communication links with each other, whilst the client is assumed to be distributed. Reliability in MAGπ is defined from the perspective of each participant, and is specified by a mapping from a role to its set of reliable roles. For this load balancer example, reliability $\mathfrak{R}_{lb}$ is configured as:

$$\mathfrak{R}_{lb} \;=\; c \mapsto \emptyset,\ srv \mapsto \{w_1, w_2\},\ w_1 \mapsto \{srv\},\ w_2 \mapsto \{srv\}$$

This reliability configuration implies that client requests, along with responses from workers, may be dropped at runtime. The following describes a protocol for this failure-prone load balancer, where a client may attempt its request at most twice, whilst a server will instruct workers to terminate if it is idle for too long.

$$\Psi_{lb} = \{srv : S_{srv},\ w_1 : S_w,\ w_2 : S_w,\ c : S_c\}$$

$$S_{srv} = \&\left\{\begin{array}{l} c : \mathsf{req}(\mathtt{Int}).\oplus\left\{\begin{array}{l} w_1 : \mathsf{fw}(\mathtt{Int}).\oplus w_2 : \mathsf{stop}().\mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}).\oplus w_1 : \mathsf{stop}().\mathbf{end} \end{array}\right. \\ \clock.\ \oplus w_1 : \mathsf{stop}().\oplus w_2 : \mathsf{stop}().\mathbf{end} \end{array}\right.$$

$$S_w = \&srv : \left\{\begin{array}{l} \mathsf{fw}(\mathtt{Int}).\oplus c : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \mathsf{stop}().\mathbf{end} \end{array}\right.$$

$$S_c = \oplus srv : \mathsf{req}(\mathtt{Int}).\&\left\{\begin{array}{l} w_1 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ w_2 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \clock.\ \oplus srv : \mathsf{req}(\mathtt{Int}).\&\left\{\begin{array}{l} w_1 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ w_2 : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \clock.\mathbf{end} \end{array}\right. \end{array}\right.$$

The first key difference here compared to the core calculus is the use of *timeout branches*. Since the server expects to receive a message from an unreliable source (the client ***c***), the branch type *must* include a failure-handling timeout branch (🕐). This is a path in a branch type which should be invoked if failure is *assumed* to have occurred. Therefore, the server type now describes behaviour for having successfully received the request, where it forwards the request to one of the workers; or follows the timeout branch, where it instructs the workers to terminate.

It is also key to note that the server no longer informs the client of the worker that was chosen to service the request. Instead, the client is now capable of waiting for a message from either of the workers, i.e., MAGπ is enabled with *undirected branching*.

Receiving the response from a worker is also unreliable, thus the client must define a timeout branch. It is key to note that since timeouts are nondeterministic, a single timeout is possible of modelling multiple failures. For instance, the client timeout (at runtime) could trigger due to: *(i)* the initial request being dropped or delayed; *(ii)* the response being dropped or delayed; or *(iii)* the server terminating before any request reaches it. Note that, if the initial request is lost, the server remains available to handle the second attempt sent by the client. If the server terminates before the second request is sent, the final timeout will terminate the client.

Contributions

The main contribution of this chapter is the first integration of timeouts into MPST as *imperfect failure detectors* with *asynchronous bag semantics* and *failure-prone* communication. The chapter studies the guarantees that can be determined through MPST in this distributed setting. Concretely:

1. Section 5.2 presents MAGπ, the first session-typed language with *nondeterministic timeouts*, *asynchronous bag semantics* and *message inconsistencies*. The language is formalised in Section 5.2.1, the type system in Section 5.2.3, and the metatheory is proven in Section 5.3.

2. In Section 5.4, key observations of using MPST with bag buffers and non-deterministic timeouts are discussed in detail. It is shown that standard notions of *session subtyping* and *safety* cannot be integrated into a MPST system with type-level bag buffers (Section 5.4.1); and non-deterministic timeouts in the type semantics are expressive enough to model various forms of failure (Section 5.4.2).

3. Section 5.5 demonstrates how MAGπ can be reconfigured to model communication over different network protocols: from the User Datagram Protocol (UDP), using the lowest

level of reliability assumptions; to the Real-time Transport Protocol (RTP), using asynchronous *queue* semantics and message loss; to the Transmission Control Protocol (TCP), using asynchronous queue semantics and fully reliable networks.

Related work is discussed in Section 5.6 and Section 5.7 concludes the chapter.

5.2 MAGπ

This section introduces MAGπ, an extension of the core calculus with *asynchronous bag buffers*, *failure-semantics*, and *non-deterministic timeouts*.

5.2.1 Language

The languages discussed thus far have used *synchronous* communication, i.e., both sends and receives are blocking actions. This is a convenient approach to programming high-level applications, but is not representative of real-world distributed communication. *Asynchronous* messaging instead models non-blocking sends, and blocking receives. This is more in-line with what is observed in reality. When a message is sent, instead of waiting to synchronise with the receiving counterpart, it instead synchronises with a buffer that is always available—allowing the send process to continue its operations immediately. Receiving processes are blocked until they can synchronise with the buffer.

Buffers can be implemented with different data structures. Commonly, a partial ordered list is used to preserve the order of messages sent between specific participants; modelling communication under "TCP-like" assumptions (e.g. as in [103, Section 7]). This thesis, for the first time, studies asynchronous MPST using *bag buffers*—i.e., buffers are multisets of messages, allowing for total message reordering, modelling communication under "UDP-like" assumptions. Buffers are formalised in Definition 5.2, and are used in the specification of MAGπ, given in Figure 5.1.

Definition 5.2 (Buffers). A message $\mathcal{M}$ is a 4-tuple $(\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle \vec{d} \rangle)$ of sender, receiver, label and payload. A bag buffer $\mathcal{B}$ maps sessions to multisets of messages.

$$\mathcal{B} ::= \emptyset \mid \mathcal{B}, s : \widetilde{\mathcal{M}}$$

A message can be inserted into the buffer using the insertion operation $\mathcal{B} \hookleftarrow \langle s, \mathcal{M} \rangle$:

$$\begin{array}{rcll} \mathcal{B} \hookleftarrow \langle s, \mathcal{M} \rangle & ::= & \mathcal{B},\ s : \mathcal{M} & \text{if } s \notin \mathcal{B} \\ \mathcal{B},\ s : \widetilde{\mathcal{M}} \hookleftarrow \langle s, \mathcal{M}' \rangle & ::= & \mathcal{B},\ s : \widetilde{\mathcal{M}}, \mathcal{M}' & \end{array}$$

$$\begin{array}{rl}
\textit{Concretes} & d ::= s[\mathbf{q}] \mid v \\
\textit{Channels} & c ::= x \mid s[\mathbf{q}] \\
\textit{Binders} & b ::= x \\
\textit{Values} & V ::= c \mid v \\
\\
\textit{Processes} & P, Q ::= (\nu s^{\mathfrak{R}})P \mid P|Q \mid \sum_{i \in I} P_i^{\oplus} \mid P^{\&} \mid P^{\&}, \circlearrowleft . Q \mid \mathrm{X}\langle \vec{V} \rangle \mid \mathbf{0} \\
\textit{Send Process} & P^{\oplus} ::= c \oplus [\mathbf{p}]\, \mathsf{m}\langle \vec{V} \rangle \;.\; P \\
\textit{Receive Process} & P^{\&} ::= c \,\&_{i \in I} [\mathbf{p_i}]\, \mathsf{m}_i(\vec{b}_i) \;.\; P_i \\
\\
\textit{Programs} & \mathscr{P} ::= (P :: \mathcal{B}, D) \\
\textit{Definitions} & D ::= \emptyset \mid D, \mathrm{X} \mapsto (\vec{x}, \vec{T}, P) \\
\textit{Protocols} & \Psi ::= \{\mathbf{p} : S_{\mathbf{p}}\}_{\mathbf{p} \in I}
\end{array}$$

Figure 5.1: Syntax of MAGπ

Names, values, and binders. A *session name*, ranged over by $s, s', \ldots$, represents a collection of interconnected participants. A *role* is a participant in a multiparty communication protocol, and each *communication endpoint* $s[\mathbf{q}]$ is obtained by indexing a session name with a role. The primitives of the calculus are *sessions with roles* $s[\mathbf{p}]$, and basic values v, both of which can be abstracted using *variables* (x, y).

Processes. Processes are ranged over by $P, Q, R, \ldots$ Unchanged from the core calculus are *parallel composition* $P|Q$, denoting P and Q running in parallel; *process call* $\mathrm{X}\langle \vec{V} \rangle$ modelling recursion; *inaction* $\mathbf{0}$ denoting the *empty process*; and *choice-guarded output* $\sum_{i \in I} c \oplus [\mathbf{p_i}]\, \mathsf{m}_i\langle \vec{V}_i \rangle \;.\; P_i$, allowing a nondeterministic send along c to any role $\mathbf{p_i}$ with label m_i and payload $\vec{V}_i$, with the process continuing as P_i. (Note the channel used for sending in a choice must be consistent across all paths.) The remaining constructors have been updated.

MAGπ assumes that all communication is *unreliable* (i.e., prone to message loss) unless specified otherwise—this is done when creating a new session. *Session restriction* $(\nu s^{\mathfrak{R}})P$ is now annotated with a *reliability configuration* $\mathfrak{R}$, defining any reliable links between roles in the session over which it is safe to assume that communication errors do not occur. Reliability is formalised in Definition 5.3 below, and will be revisited in the process semantics (Section 5.2.2).

Definition 5.3 (Reliability). The reliability mapping for a session $\mathfrak{R}$ is a mapping from

roles to (possibly empty) sets of roles.

$$\mathfrak{R} ::= \emptyset \mid \mathfrak{R}, \boldsymbol{p} : \{\boldsymbol{q_i}\}_{i \in I}$$

Reliability context $\mathcal{R}$ maps sessions to their reliability configuration.

$$\mathcal{R} ::= \emptyset \mid \mathcal{R}, s : \mathfrak{R}$$

As a shorthand notation, $\mathfrak{R}_s$ refers to the reliability context $s : \mathfrak{R}$.

Branching receive $c\&_{i \in I}[\boldsymbol{p_i}]\,\mathsf{m}_i(\vec{b_i})\ .\ P_i$ now allows for the ability to expect a message from multiple participants, nondeterministically synchronising with any first message that becomes available. The syntax denotes a process waiting on channel c for one of a set of messages from role $\boldsymbol{p_i}$ with label m_i, binding the received data to variables $\vec{b_i}$ before continuing according to P_i.

The second key difference with the receive construct is the optional use of the *timeout* branch $P^{\&}, \textcircled{\scriptsize 🕐}.\, Q$. Timeouts are used as the (imperfect) *failure detection mechanism* in MAGπ. If a failure is *assumed* to have lost or prevented the incoming message, then process Q is initiated. Failures are said to be assumed, as opposed to detected, since the calculus reflects the impossibility result of distinguishing between a delayed *vs* lost message. Thus, it is possible for a processes to prematurely timeout without its corresponding message having been lost—just like the real-world! This will become clear with the process semantics (Definition 5.4).

Furthermore, it is key to note that timeouts are non-deterministic—they model an arbitrary and unknown duration of time a process waits before assuming a failure has occurred. This is sufficient for the calculus, as its purpose is to be used to study communication protocols in this unreliable setting; real implementations of the language may specify durations for timeouts, though this is tangential to this thesis.

Programs, Definitions, and Protocols. A *program* is what is written by an end-user of the language; it is a tuple $(P :: \mathcal{B}, D)$ of a "main" process P, with buffer $\mathcal{B}$, and process *definitions* D. For programs to be well-formed (henceforth assumed), the buffer initiated for a program should be empty. Definitions map process names (ranged over by X,Y,...) to process bodies $(\vec{x}, \vec{T}, P)$ to be used for recursion, where $\vec{x}$ is a list of accepted arguments, $\vec{T}$ is a list of types for said arguments, and P is the process in which the arguments are used. Finally, *protocols* are *session type* specifications of roles in a given session. Formally, Ψ is a set of role-session type pairs, $\{\boldsymbol{p} : S_{\boldsymbol{p}}\}_{\boldsymbol{p} \in I}$ for a non-empty I, defining the communication patterns that each participant of a session should follow. These will be used to annotate sessions.

Process reduction $P :: \mathcal{B} \to_{D;\mathcal{R}} P' :: \mathcal{B}'$

$$\text{R-Snd}\quad s[\boldsymbol{p}] \oplus [\boldsymbol{q}]\,\mathsf{m}\langle\vec{d}\rangle \;.\; P :: \mathcal{B} \;\to_{D;\mathcal{R}}\; P :: \mathcal{B} \leftarrow \langle s, (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle\vec{d}\rangle)\rangle$$

$$\text{R-Rcv}\quad \frac{\mathcal{M}' = (\boldsymbol{p_k} \triangleright \boldsymbol{q} : \mathsf{m}_k\langle\vec{d}\rangle) \qquad k \in I}{s[\boldsymbol{q}] \&_{i\in I} [\boldsymbol{p_i}]\mathsf{m}_i(\vec{b_i}) \;.\; P_i\;[,\!\!\circlearrowright.\; Q] :: \mathcal{B},\; s : \widetilde{\mathcal{M}}, \mathcal{M}' \;\to_{D;\mathcal{R}}\; P_k\{\vec{d}/\vec{b_k}\} :: \mathcal{B}, s : \widetilde{\mathcal{M}}}$$

$$\text{R-}\circlearrowright\quad s[\boldsymbol{q}] \&_{i\in I} [\boldsymbol{p_i}]\mathsf{m}_i(\vec{b_i}) \;.\; P_i,\; \circlearrowright.\; Q :: \mathcal{B} \;\to_{D;\mathcal{R}}\; Q :: \mathcal{B}$$

$$\text{R-Drop}\quad \frac{\mathcal{M}' = (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle\vec{d}\rangle) \qquad \boldsymbol{q} \notin \mathfrak{R}(\boldsymbol{p})}{P :: \mathcal{B}, s : \widetilde{\mathcal{M}}, \mathcal{M}' \;\to_{D;\mathcal{R},\mathfrak{R}_s}\; P :: \mathcal{B}, s : \widetilde{\mathcal{M}}}$$

$$\text{R-+}\quad \frac{j \in I}{\textstyle\sum_{i\in I} P_i^{\oplus} :: \mathcal{B} \;\to_{D;\mathcal{R}}\; P_j^{\oplus} :: \mathcal{B}}$$

$$\text{R-X}\quad \frac{D(\mathrm{X}) = (\vec{x}, \vec{T}, P)}{\mathrm{X}\langle\vec{d}\rangle :: \mathcal{B} \;\to_{D;\mathcal{R}}\; P\{\vec{d}/\vec{x}\} :: \mathcal{B}}$$

$$\text{R-}|\quad \frac{P :: \mathcal{B} \to_{D;\mathcal{R}} P' :: \mathcal{B}'}{P \mid Q :: \mathcal{B} \to_{D;\mathcal{R}} P' \mid Q :: \mathcal{B}'}$$

$$\text{R-}\nu\quad \frac{P :: \mathcal{B} \to_{D;\mathcal{R},\mathfrak{R}_s} P' :: \mathcal{B}'}{(\nu s^{\mathfrak{R}})\,P :: \mathcal{B} \;\to_{D;\mathcal{R}}\; (\nu s^{\mathfrak{R}})\,P' :: \mathcal{B}'}$$

$$\text{R-}\equiv\quad \frac{P_1 :: \mathcal{B} \;\to_{D;\mathcal{R}}\; P_2 :: \mathcal{B}' \qquad P_1 :: \mathcal{B} \equiv P_1' :: \mathcal{B} \qquad P_2 :: \mathcal{B}' \equiv P_2' :: \mathcal{B}'}{P_1' :: \mathcal{B} \;\to_{D;\mathcal{R}}\; P_2' :: \mathcal{B}'}$$

Figure 5.2: Operational semantics MAGπ.

5.2.2 Process Semantics

Using the standard notion of structural congruence for processes (Figure 3.5), asynchronous and failure-prone process reduction is formalised in Definition 5.4 using bag buffers.

Definition 5.4 (Process reduction). A process with buffer $P :: \mathcal{B}$ reduces parametric on process definitions D and reliability context $\mathcal{R}$, written $P \to_{D;\mathcal{R}} P'$, via the operational semantics defined in Figure 5.2. Multistep reductions are expressed using the reflexive and transitive closure, i.e., $P \to^*_{D;\mathcal{R}} P'$.

Since MAGπ is asynchronous, sending and receiving are now decoupled. Intuitively, sending inserts a new message into the buffer, whereas receiving consumes a message from the buffer. Rule R-Snd uses the message insertion operation (Definition 5.2) to place a message into the buffer whilst advancing the send process to its continuation. Rule R-Rcv advances a receive process to the corresponding continuation if there exists a message in the buffer matching the sender and message label defined in the process. Doing so removes the message from the buffer. This rule also applies to processes that optionally define a failure-handling timeout branch.

$$
\begin{array}{rl}
\textit{Static types} & T ::= S \mid B \\
\textit{Basic types} & B ::= \texttt{Nat} \mid \texttt{Real} \mid \texttt{String} \mid \cdots \\
\textit{Session types} & S ::= S^{\oplus} \mid S^{\&} \mid S^{\&}, \textcircled{1}.S' \mid \mu \mathrm{t}.S \mid \mathrm{t} \mid \mathbf{end} \\
\textit{Selection type} & S^{\oplus} ::= \oplus_{i \in I} \boldsymbol{p_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \\
\textit{Branching type} & S^{\&} ::= \&_{i \in I} \boldsymbol{p_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \\
\textit{Message type} & M ::= (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T}))
\end{array}
$$

Figure 5.3: Syntax of MAGπ Types

Failure semantics. The process semantics model *total message reordering*, *message loss*, and *message delay*. Message reordering is a result of using buffers defined as multisets (bags), thus being implicitly unordered out-of-the-box. Message loss is modelled directly from rule R-Drop, which can remove a message from the buffer at any point, provided that the message is not part of an assumed reliable link.

Dropped messages are intended to be handled via *timeouts*. To this end, rule R-$\textcircled{1}$ reduces a receiving process to its timeout branch continuation. Notably, there are no prerequisites for a timeout to occur. Since timeouts are nondeterministic (can happen at any point regardless of what is in the buffer), it is possible for a process to timeout even though a valid consumable message is in the buffer. This models message delay.

The parametric reliability context is key to building semi-reliable systems. The context is populated via rule R-ν, stating that a process can reduce under a session restriction provided the internal process can reduce using the extended reliability context.

The remaining rules are defined similar to the core calculus. Rule R-+ nondeterministically reduces a choice to one of its paths; rule R-X looks up a process name in the definitions, reducing to the found process substituted with any passed arguments; rule R-$|$ allows processes to reduce under parallel composition and rule R-$\equiv$ allows for reduction up-to congruence.

5.2.3 Types

The syntax of MAGπ types is given in Figure 5.3.

Syntax of types. Static types consist of either *session types* or *basic types*. The former is used to type *endpoints*, whilst the latter is used to type basic values and variables.

Session types describe patterns of communication. *Selection* $\oplus_{i \in I} \boldsymbol{p_i}{:}\mathsf{m}_i(\vec{T}_i).S_i$ represents a role's *internal* choice of sending a message to one of a set of participants $\boldsymbol{p_i}$, having label m_i and payload of types $\vec{T}_i$. The channel should then be used in accordance to the continuation type of the selected path S_i. *Branching* $\&_{i \in I} \boldsymbol{p_i}{:}\mathsf{m}_i(\vec{T}_i).S_i$ refers to an *external* choice, where any one of a set of messages could be received from any $\boldsymbol{p_i}$, described by the labels m_i and payloads $\vec{T}_i$.

The communication pattern after the branch follows the description of the continuation type of the branched path, by S_i. Both branching and selection assume non-empty sets of paths I.

Notably different from the standard definition of session types given in Section 3.2.2, branching now supports: *(i) undirected branching*, i.e., the ability to anticipate messages from possibly multiple roles; and *(ii)* an optional *timeout branch* $S^{\&}, \textcircled{t}.S'$, specifying a failure-handling subprotocol S' that should be followed if a failure is assumed at runtime.

Recursion is modelled using tail-recursive binders $\mu \mathrm{t}.S$, binding t in the continuation type S. Recursion variables cannot be free and must appear guarded under branching/selection types. Lastly, type **end** marks the end of a communication pattern.

To type buffers, *message types* $(\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T}))$ are introduced. A multiset of message types will be used to type the buffer of a given session. These can be thought of as type-level buffers, which are defined within a typing context, as seen in Figure 5.4.

A type context Γ maps endpoints to session types $s[\boldsymbol{p}] : S$, variables to static types $x : T$, and is now extended to map session names to multisets of message types $s : \widetilde{M}$. Context splitting and addition are also extended to reflect the updated context. As before, splitting and addition ensure linear resources are not lost or copied, whilst allowing entities with basic types to be copied and merged. Session names are handled differently: like endpoints, they cannot be copied from splitting, but are allowed to be merged using context addition. Furthermore, the *buffer extraction* operation is introduced, which can be used to separate type buffers (i.e., session name mappings) from all other entries in the context.

Type Checking

Unlike the languages discussed thus far, MAGπ does not rely on the standard notion of *session subtyping*. Instead, a different preorder on types is used—*compatibility* (Definition 5.5)—for reasons discussed in detail in Section 5.4.

Definition 5.5 (Compatible). The compatibility relation $\sqsubseteq$ is co-inductively defined on types by the following inference rules:

$$\frac{(\vec{T}_i \sqsubseteq \vec{T}'_i)_{i\in I} \qquad (S_i \sqsubseteq S'_i)_{i\in I}}{\&_{i\in I}\boldsymbol{p}_i{:}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq \&_{i\in I}\boldsymbol{p}_i{:}\mathsf{m}_i(\vec{T}'_i).S'_i} \qquad \frac{(\vec{T}_i \sqsupseteq \vec{T}'_i)_{i\in I} \qquad (S_i \sqsubseteq S'_i)_{i\in I}}{\oplus_{i\in I\cup J}\boldsymbol{p}_i{:}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq \oplus_{i\in I}\boldsymbol{p}_i{:}\mathsf{m}_i(\vec{T}'_i).S'_i}$$

$$\frac{S_1^{\&} \sqsubseteq S_2^{\&} \qquad S'_1 \sqsubseteq S'_2}{S_1^{\&}, \textcircled{t}.S'_1 \sqsubseteq S_2^{\&}, \textcircled{t}.S'_2} \qquad \frac{S\{\mu \mathrm{t}.S/\mathrm{t}\} \sqsubseteq S'}{\mu \mathrm{t}.S \sqsubseteq S'} \qquad \frac{S \sqsubseteq S'\{\mu \mathrm{t}.S'/\mathrm{t}\}}{S \sqsubseteq \mu \mathrm{t}.S'} \qquad \frac{}{T \sqsubseteq T}$$

Compatibility requires all types other than selection to be equivalent, i.e., the left and right elements of the relation are identical. For selection types, the left element is allowed to contain

Typing contexts

$$\Gamma ::= \emptyset \mid \Gamma, s[\mathbf{p}] : S \mid \Gamma, x : T \mid \Gamma, s : \widetilde{M}$$

Context splitting $\boxed{\Gamma = \Gamma_1 \cdot \Gamma_2}$

$$\frac{}{\emptyset = \emptyset \cdot \emptyset} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, x : B = (\Gamma_1, x : B) \cdot (\Gamma_2, x : B)} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : S = (\Gamma_1, c : S) \cdot \Gamma_2}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, s : \widetilde{M} = (\Gamma_1, s : \widetilde{M}) \cdot \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, s : \widetilde{M} = \Gamma_1 \cdot (\Gamma_2, s : \widetilde{M})} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : S = \Gamma_1 \cdot (\Gamma_2, c : S)}$$

Context addition $\boxed{\Gamma_1 + \Gamma_2 = \Gamma}$

$$\frac{}{\Gamma + \emptyset = \Gamma} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{(\Gamma_1, x : B) + (\Gamma_2, x : B) = \Gamma, x : B} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{(\Gamma_1, s : \widetilde{M}_1) + (\Gamma_2, s : \widetilde{M}_2) = \Gamma, s : \widetilde{M}_1, \widetilde{M}_2}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma \quad s \notin \mathsf{dom}(\Gamma_2)}{(\Gamma_1, s : \widetilde{M}) + \Gamma_2 = \Gamma, s : \widetilde{M}} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad s \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + (\Gamma_2, s : \widetilde{M}) = \Gamma, s : \widetilde{M}}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma \quad c \notin \mathsf{dom}(\Gamma_2)}{(\Gamma_1, c : T) + \Gamma_2 = \Gamma, c : T} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad c \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + (\Gamma_2, c : T) = \Gamma, c : T}$$

Buffer Extraction $\boxed{\Gamma = \Gamma_1 ; \Gamma_2}$

$$\frac{\Gamma = \Gamma_1, \Gamma_2 \qquad \Gamma_1 = \{c_i : T_i\}_{i \in I} \qquad \Gamma_2 = \{s_j : \widetilde{M}_j\}_{j \in J}}{\Gamma = \Gamma_1 ; \Gamma_2}$$

Figure 5.4: Type context operations.

more paths than the right. In other words, compatibility describes subtyping of selection types, and equivalence of any other type.

To identify contexts that have used up all linear resources, the end predicate (Definition 5.6) is once again used. Note that an end-typed context should not contain any type-level buffers.

Definition 5.6 (End-typed environment). A context is *end-typed*, written $\mathsf{end}(\Gamma)$, iff its channels have type **end**, and it is void of buffer types.

$$\frac{}{\mathsf{end}(\emptyset)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(c : \mathbf{end}, \Gamma)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(x : B, \Gamma)}$$

Definition 5.7 (Garbage Collector Predicate). The garbage collector predicate gc checks whether session types in a context can be matched to the payloads of buffer types. It is used to garbage collect session types that are unconsumed due to message loss.

$$\frac{\mathsf{end}(\Gamma)}{\mathsf{gc}(\Gamma\,;\, s:\emptyset)} \qquad \frac{\forall i \in 1..n : \mathsf{end}(T_i) \text{ or } \Gamma_i' \vdash c_i : T_i \qquad \mathsf{gc}(\Gamma\,;\, s:\widetilde{M})}{\mathsf{gc}(\Gamma(\cdot\,\Gamma_i')_{i\in 1..n}\,;\, s:(\boldsymbol{p}\triangleright\boldsymbol{q}:\mathsf{m}((T_i)_{i\in 1..n})),\widetilde{M})} \text{ for any } c_i$$

Message types are *not* linear. It is possible for a context to contain message types that no longer appear in the process buffer (due to message loss). For this reason, type buffers are not treated as linear resources, and will be handled separately in the typing rules. However, before presenting the typing rules, there is one issue with having non-linear message types: *payloads*. It is possible that messages, which are not treated linearly, carry linearly typed payloads. Hence, Definition 5.7 introduces the *garbage collector predicate*, which checks that any leftover linear types can be attributed to the payloads of unused messages.

Figure 5.5 presents the typing rules for MAGπ. There are five typing judgements: *(i)* the *value judgement* $\Gamma \vdash V : T$ assigns type T to value V under context Γ; *(ii)* the *buffer judgement* $\Gamma \vdash \mathcal{B}$ verifies that a buffer $\mathcal{B}$ is well-typed under context Γ; *(iii)* the *process judgement* $\Gamma \vdash_D P$ checks whether a process P is well-typed under a context Γ and process definitions D; *(iv)* the *process with buffer judgement* $\Gamma \vdash_D P :: \mathcal{B}$ lifts the buffer and process judgements into one; and *(v)* the *program judgement* $\vdash \mathscr{P}$ checks whether a program is self-enclosed.

The value judgement has three rules. Rule T-S types variables and role indexed sessions to static types, up-to compatibility. Rule T-Wkn allows weakening, i.e., if value V has type T under some environment Γ_1, and there exists an end-typed environment Γ_2 such that $\Gamma_1 + \Gamma_2 = \Gamma$, then the rule allows the value to be typed under the larger context Γ. Rule T-B types a basic value v to type B iff v belongs to B.

The buffer judgement has four rules. A message in the buffer is typed using rule T-Msg iff there exists a corresponding message in the type buffer (along with the types for any payloads in the message). Rule T-Msg is designed to be inductively applied until there are no more messages in the *process* buffer. Rule T-Wkn$_2$ caters for weakening by allowing types which can be garbage collected to be added into the context. This is vital for typing process buffers that have previously dropped messages. Rule T-Empty$_1$ types a buffer with no messages $\mathcal{B}, s:\emptyset$ iff the context can type the remaining buffers $\mathcal{B}$. Lastly, when the process buffer is empty and all type buffers should have been removed by weakening, rule T-Empty$_2$ types the empty buffer under an end-typed context.

Typing Rules for Values and Buffers $\boxed{\Gamma \vdash V : T}$ $\boxed{\Gamma \vdash \mathcal{B}}$

$$\text{T-Sub}\ \frac{T \sqsubseteq T'}{c : T \vdash c : T'} \qquad \text{T-Wkn}_1\ \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad \Gamma_1 \vdash V : T \quad \mathsf{end}(\Gamma_2)}{\Gamma \vdash V : T} \qquad \text{T-B}\ \frac{v \in B}{\emptyset \vdash v : B} \qquad \text{T-Wkn}_2\ \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad \Gamma_1 \vdash \mathcal{B} \quad \mathsf{gc}(\Gamma_2)}{\Gamma \vdash \mathcal{B}} \qquad \text{T-Empty}_1\ \frac{\Gamma \vdash \mathcal{B}}{\Gamma \vdash \mathcal{B}, s : \emptyset}$$

$$\text{T-Empty}_2\ \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash \emptyset} \qquad \text{T-Msg}\ \frac{(\Gamma_i \vdash d_i : T_i)_{i \in 1..n} \quad \Gamma\,;\, \Gamma' + s : \widetilde{M} \vdash \mathcal{B}, s : \widetilde{\mathcal{M}}}{\Gamma\,(\cdot\,\Gamma_i)_{i \in 1..n}\,;\, \Gamma' \cdot s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}((T_i)_{i \in 1..n})), \widetilde{M} \vdash \mathcal{B}, s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle (d_i)_{i \in 1..n} \rangle), \widetilde{\mathcal{M}}}$$

Typing Rules for Programs and Processes $\boxed{\vdash \mathscr{P}}$ $\boxed{\Gamma \vdash_D P :: \mathcal{B}}$ $\boxed{\Gamma \vdash_D P}$

$$\text{T-}\mathscr{P}\ \frac{\emptyset \vdash_D P :: \mathcal{B} \quad \forall i \in I : \{\overrightarrow{b_i : T_i}\} \vdash_D Q_i}{\vdash (P :: \mathcal{B}, D = \{\mathrm{X}_i \mapsto (\vec{b}_i, \vec{T}_i, Q_i)\}_{i \in I})} \qquad \text{T-}\mathcal{B}\ \frac{\Gamma \vdash_D P \quad \Gamma_d\,;\, \Gamma_s \vdash \mathcal{B}}{\Gamma \cdot \Gamma_d\,;\, \Gamma_s \vdash_D P :: \mathcal{B}} \qquad \text{T-}\nu\ \frac{\varphi(\mathtt{assoc}_s(\Psi)) \quad \varphi \text{ is a } \mathfrak{R}_s\text{-}\textit{safety} \text{ property} \quad s \notin \Gamma \quad \Gamma + \mathtt{assoc}_s(\Psi) \vdash_D P}{\Gamma \vdash_D (\nu s^{\mathfrak{R}} : \Psi)\ P}$$

$$\text{T-}\&\ \frac{\Gamma_\& \vdash c : \&_{i \in I} \boldsymbol{q_i} : \mathsf{m}_i(\vec{T}_i).S_i\,[,\, \textcircled{t}.S'] \quad (\Gamma + c : S_i + \overrightarrow{b_i : T_i} \vdash_D P_i)_{i \in I} \quad [\Gamma + c : S' \vdash_D Q]}{\Gamma \cdot \Gamma_\& \vdash_D c\,\&_{i \in I}\,[\boldsymbol{q_i}]\,\mathsf{m}_i(\vec{b}_i)\ .\ P_i\,[,\, \textcircled{t}.Q]} \qquad \text{T-}\oplus\ \frac{\Gamma_\oplus \vdash c : \oplus \boldsymbol{q} : \mathsf{m}(\vec{T}).S \quad (\Gamma_i \vdash V_i : T_i)_{i \in 1..n} \quad \Gamma + c : S \vdash_D P}{\Gamma \cdot \Gamma_\oplus(\cdot\,\Gamma_i)_{i \in 1..n} \vdash_D c \oplus [\boldsymbol{q}]\,\mathsf{m}\langle (V_i)_{i \in 1..n} \rangle\ .\ P}$$

$$\text{T-X}\ \frac{D(\mathrm{X}) = (\vec{x}, \vec{T}, P) \quad \forall i \in 1..n : \Gamma_i \vdash V_i : T_i \quad \mathsf{end}(\Gamma_0)}{\Gamma_0(\cdot\,\Gamma_i)_{i \in 1..n} \vdash_D \mathrm{X}\langle (V_i)_{i \in 1..n} \rangle} \qquad \text{T-}|\ \frac{\Gamma_1 \vdash_D P_1 \quad \Gamma_2 \vdash_D P_2}{\Gamma_1 \cdot \Gamma_2 \vdash_D P_1 | P_2} \qquad \text{T-}+\ \frac{(\Gamma \vdash_D P_i^{\oplus})_{i \in I}}{\Gamma \vdash_D \sum_{i \in I} P_i^{\oplus}} \qquad \text{T-}\mathbf{0}\ \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash_D \mathbf{0}}$$

Figure 5.5: Typing rules.

Example 5.8 (Garbage Collection). Consider the following process buffer and type context:

$$\{s[\boldsymbol{r}] : S\,;\, s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(S))\} \vdash s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle s[\boldsymbol{r}] \rangle)$$

This type judgement holds via rule T-Msg. Observe that the payload of the message in the buffer ($s[\boldsymbol{r}]$) is typed by the linear part of the context ($s[\boldsymbol{r}] : S$) since the type matches that which is indicated in the payload type (the S in $(\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(S))$).

It is possible for the message in the process buffer to be dropped at runtime, leaving the following context and empty buffer:

$$\Gamma = \{s[\boldsymbol{r}] : S\,;\, s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(S))\} \vdash s : \emptyset$$

Since message loss only affects the process buffer, the context now contains a message

type and session type (of the dropped payload) that will not be used. The judgement, however, still holds by rule T-Wkn$_2$. The rule uses the garbage collector predicate (Definition 5.7) to ensure that the session type in the context matches the payload of the message type, i.e., $\exists c \in \Gamma : \Gamma(c) \sqsubseteq S$. The context contains an entry with a compatible type to that in the payload, therefore $\mathsf{gc}(\Gamma)$. By rule T-Wkn$_2$, it then follows that $\emptyset \vdash s : \emptyset$, which in turn holds by rule T-Empty$_1$ and rule T-Empty$_2$.

The program judgement is only used by rule T-$\mathscr{P}$ to type a program $(P :: \mathcal{B}, D)$. This judgement does not use a type environment, and ensures that programs are *closed*. This is achieved by requiring the main process with buffer $P :: \mathcal{B}$ to be well-typed under an empty context. Furthermore, process definitions are also required to be closed by typing them under a context containing only the arguments defined for that process name.

The process with buffer judgement also only contains one rule. Rule T-$\mathcal{B}$ uses the buffer extraction operation to isolate the type buffers and use them (along with any payload types in Γ_d) to type the process buffer. The remaining context is used to type the process.

The final judgement types processes under type context Γ. Most rules are unchanged from the core calculus (only rules T-ν and T-& have been updated).

Rule T-ν is what makes the type system *generalised*. Instead of forcing a specific property on types, session restriction requires that the defined protocol adheres to the minimum requirements for subject reduction. Specifically, the rule requires proof of $\varphi(\mathtt{assoc}_s(\Psi))$, where φ is any $\mathfrak{R}$-*safety* property (where $\mathfrak{R}$ is the reliability configuration of the restricted session s)—the *largest* safety property will be defined later. If the session protocol adheres to these requirements, and the restricted session has not been previously established, then the restricted process can be typed using the newly added context.

Rule T-& types the receive process by checking that the channel used to receive can be mapped to a branching session type with matching role and branch labels. Then, every continuation process should be typed using the respective branch from the continuation session type, along with any new binders specified in the payload. If a timeout branch is defined, then the timeout process Q should be well-typed under the context with the updated mapping of the endpoint to the timeout session type.

Rule T-$\oplus$ types a send process. First, the channel used for sending should be typed to a selection session type with a matching role and message label (it is key to note that this accounts for compatibility). Then, the values sent as the message payload should be typed to the types specified in the selection session type, under some split context. Lastly, the continuation process should be well-typed under the remainder of the original context, with the channel updated to the continuation session type.

Rule T-X checks that the type of every value used in a process call match the expected types defined in the process definition D. Rule T-| states that process composition is well-typed under

a context if that context can be split into two sub-environments that type each of the composed processes. Rule T-+ requires all processes in a nondeterministic choice to be typed under the same environment. Lastly, a terminated process is typed by rule T-**0** if any remaining session types in the type environment are *end-typed*.

5.2.4 Type Semantics

Context reduction (Definition 5.9) models type-level communication. Since, MAGπ is asynchronous, the LTS specification has changed from that specified in the core calculus—the single communication action is now split into two separate actions. These are *(i) enqueuing*, adding a message into the type buffer; and *(ii) dequeuing*, synchronising a branch type with a message in the buffer. In addition, a timeout action is also added to model failures (a detailed discussion of what, and how, failures can be modelled using timeouts, is the subject of Section 5.4.2).

Definition 5.9 (Context reduction). The type LTS is updated to allow for asynchrony and timeouts. The new definition for actions A, is given below.

$$\text{Actions} \quad A ::= s{:}\boldsymbol{p}{\oplus}\boldsymbol{q}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{p}{\oplus}\boldsymbol{q}{:}\mathsf{m} \mid s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m} \mid s{:}\boldsymbol{p}{:}\textcircled{\scriptsize t}$$

From left to right, these read as: *output*, *enqueue*, *input*, *dequeue*, and *timeout*. The type LTS is defined by the transitions listed in Figure 5.6. Context *reduction*, $\Gamma \rightarrow \Gamma'$, is defined iff $\Gamma \xrightarrow{A} \Gamma'$, where $A \in \{s{:}\boldsymbol{p}{\oplus}\boldsymbol{q}{:}\mathsf{m}, s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m}, s{:}\boldsymbol{p}{:}\textcircled{\scriptsize t}\}$. The transitive and reflexive closure of context reduction is written as $\rightarrow^*$; and $\Gamma\rightarrow$ represents the existence of some transition from Γ.

Unchanged from the synchronous version, using rule Γ-& (resp. rule Γ-$\oplus$), a branching session type (resp. selection) can transition via an input (resp. output) label to reach any one of its continuation types. Importantly, the session, roles, message labels and payload types of the action correspond to what is dictated in type. Rule Γ-& also applies to types that define a failure-handling timeout branch.

Communication is now modelled as two separate actions. Rule Γ-Enq states that if a context can perform an output, then it may enqueue the message into the type buffer. Importantly, this relies on the fact that buffer types are merged through context addition (as defined in Figure 5.4). On the receiving end, rule Γ-Deq advances branching types to their continuations whilst consuming a message from the type buffer if the sent payload types are compatible with those expected in the branching type.

Rule Γ-$\textcircled{\scriptsize t}$ advances branching types to their timeout branch without affecting the type buffer. Lastly, the remaining rules allow for transitions modulo context composition, context splitting, and recursive unfolding.

Context Reduction $\boxed{\Gamma \xrightarrow{A} \Gamma'}$

$$\Gamma\text{-}\& \quad \frac{S = \&_{i\in I} \boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i' \left[, \oplus. S''\right] \qquad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q_k}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S_k'}$$

$$\Gamma\text{-}\textsc{Deq} \quad \frac{\Gamma \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}(\vec{T})} \Gamma' \qquad \vec{T}' \sqsubseteq \vec{T}}{\Gamma, s : (\boldsymbol{q} \triangleright \boldsymbol{p} : \mathsf{m}(\vec{T}')), \widetilde{M} \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}} \Gamma' + s : \widetilde{M}}$$

$$\Gamma\text{-}\oplus \quad \frac{S = \oplus_{i\in I} \boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i' \qquad k \in I}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q_k}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S_k'}$$

$$\Gamma\text{-}\textsc{Enq} \quad \frac{\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \Gamma'}{\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}} \Gamma' + s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T}))}$$

$$\Gamma\text{-}\oplus \quad \frac{S = \&_{i\in I} \boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i', \oplus. S''}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}:\oplus} s[\boldsymbol{p}] : S''} \qquad \Gamma\text{-}\textsc{Cong}_1 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma, \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''} \qquad \Gamma\text{-}\textsc{Cong}_2 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma \cdot \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''} \qquad \Gamma\text{-}\mu \quad \frac{\Gamma \cdot c : S\{\mu \mathrm{t}.S/\mathrm{t}\} \xrightarrow{A} \Gamma'}{\Gamma \cdot c : \mu \mathrm{t}.S \xrightarrow{A} \Gamma'}$$

Figure 5.6: Type semantics.

Example 5.10 (Failure-Prone Load Balancer: Type Reduction). Recalling the failure-prone load balancer from Example 5.1, the type semantics are now demonstrated using the prescribed protocol. There are two reductions that are immediately available. Either the client sends its request, inserting the message into the type buffer; or the server can timeout. Consider the server remains active until a request is received.

$$\begin{array}{l} s[\boldsymbol{c}] : S_{\boldsymbol{c}},\ s[\boldsymbol{srv}] : S_{\boldsymbol{srv}},\ s[\boldsymbol{w_1}] : S_{\boldsymbol{w}},\ s[\boldsymbol{w_2}] : S_{\boldsymbol{w}} \\ \rightarrow \\ \quad s[\boldsymbol{c}] : \&\left\{\begin{array}{l} \boldsymbol{w_1} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \boldsymbol{w_2} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \oplus. \oplus \boldsymbol{srv} : \mathsf{req}(\texttt{Int}).\&\left\{\begin{array}{l} \boldsymbol{w_1} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \boldsymbol{w_2} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \oplus.\mathbf{end} \end{array}\right. \end{array}\right. \quad \begin{array}{l} s[\boldsymbol{srv}] : S_{\boldsymbol{srv}}, \\ s[\boldsymbol{w_1}] : S_{\boldsymbol{w}}, \\ s[\boldsymbol{w_2}] : S_{\boldsymbol{w}}, \end{array} \\ \quad s : (\boldsymbol{c} \triangleright \boldsymbol{srv} : \mathsf{req}(\texttt{Int})) \end{array}$$

Given the type semantics is now asynchronous, the selection type reduces via rule Γ-Enq, enqueuing the request into the type buffer. At this point, there are now three possible paths in the LTS. The message could be successfully received, the server could timeout, or the client could timeout. Consider a timeout at the client, modelling either a delay or

the failure of the initial request.

$$
\begin{array}{l}
\rightarrow \\
s[c] : \oplus srv : \mathsf{req}(\mathtt{Int}).\& \left\{ \begin{array}{ll} w_1 : \mathsf{ans}(\mathtt{String}).\mathbf{end} & s[srv] : S_{srv}, \\ w_2 : \mathsf{ans}(\mathtt{String}).\mathbf{end} & , \; s[w_1] : S_w, \\ \textcircled{t}.\mathbf{end} & s[w_2] : S_w, \end{array} \right. \\
s : (c \triangleright srv : \mathsf{req}(\mathtt{Int})) \\
\rightarrow \\
s[c] : \& \left\{ \begin{array}{ll} w_1 : \mathsf{ans}(\mathtt{String}).\mathbf{end} & s[srv] : S_{srv}, \\ w_2 : \mathsf{ans}(\mathtt{String}).\mathbf{end} & , \; s[w_1] : S_w, \\ \textcircled{t}.\mathbf{end} & s[w_2] : S_w, \end{array} \right. \\
s : (c \triangleright srv : \mathsf{req}(\mathtt{Int})), (c \triangleright srv : \mathsf{req}(\mathtt{Int}))
\end{array}
$$

The server can now receive any of the requests via rule Γ-Deq, consuming one of the messages from the buffer. Note that there is no difference in whether the initial, or the second, request is consumed.

$$
\begin{array}{l}
\rightarrow \\
s[c] : \& \left\{ \begin{array}{ll} w_1 : \mathsf{ans}(\mathtt{String}).\mathbf{end} & \\ w_2 : \mathsf{ans}(\mathtt{String}).\mathbf{end} & , \\ \textcircled{t}.\mathbf{end} & \end{array} \right. \quad s[srv] : \oplus \left\{ \begin{array}{l} w_1 : \mathsf{fw}(\mathtt{Int}).\oplus w_2 : \mathsf{stop}().\mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}).\oplus w_1 : \mathsf{stop}().\mathbf{end} \end{array} \right. , \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad s[w_1] : S_w, \; s[w_2] : S_w, \\
s : (c \triangleright srv : \mathsf{req}(\mathtt{Int}))
\end{array}
$$

At this point, the types can keep reducing until either the client receives a response from a worker, or until the client triggers its next timeout. Regardless of whether or not the response is received, all session types eventually reach **end** and the type buffer will (at least) contain the client request. This models the fact that either the request was dropped, or was delayed with an infinite upper-bound. Note that in MAGπ, as in the real-world, it is impossible to distinguish between these two scenarios.

5.3 Metatheory

The metatheory blueprint is no different for an asynchronous calculus compared to the steps taken for the synchronous calculi presented previously. The main results are still *subject reduction* (Section 5.3.1) and *session fidelity* (Section 5.3.2), which are dependent on some largest *safety* (resp. *fidelity*) property. These results together provide a framework with which specific process properties can be verified from their protocols. Given the approach to generalised MPST metatheory has been sufficiently demonstrated in the previous chapters, this section focuses on the main two results, and concludes with a discussion on *decidability* in Section 5.3.3.

5.3.1 Subject Reduction

The previous type systems discussed used safety properties that are built on the same standard principles as other generalised MPST theories (e.g. [103, 118, 13, 51]). These safety properties all reject the presence of *unexpected messages* at any point in a protocol. The issue this raises in MAGπ is that the very definition of an *unexpected* message is contradictory to bag buffers.

Example 5.11 (Unexpected Message?). Consider two roles communicating two messages in opposite order.

$$\boldsymbol{p} : \&\boldsymbol{q}{:}\mathsf{m}.\&\boldsymbol{q}{:}\mathsf{m}' \qquad \boldsymbol{q} : \oplus\boldsymbol{p}{:}\mathsf{m}'.\oplus\boldsymbol{p}{:}\mathsf{m}$$

This protocol is not safe (Definition 3.13), nor is it safe$_!$ (Definition 4.20), neither is it safe for standard (queue-buffer) asynchronous theories [103]. However, under asynchronous bag-buffer semantics, as is used in MAGπ (Figure 5.6), these types describe a terminating protocol.

Bag buffer semantics introduce unique protocols which may contain unexpected messages but still describe desirable behaviour. This is because, since messages are unordered, a message being unexpected at a given point in time does not mean *(i)* an expected message will not eventually arrive; nor that *(ii)* the unexpected message will not eventually be expected.

The philosophy adopted in MAGπ is that safety should accept protocols such as those described in Example 5.11. To do this, safety is weakened to allow for unexpected messages, now only rejecting *incompatible* messages. (It is for this reason that standard session subtyping is not considered in MAGπ, a more detailed discussion of this is presented in Section 5.4.)

Definition 5.12 (Safety). φ is a $\mathcal{R}$-*safety* property on type environment Γ iff:

φ-**S** $\varphi(\Gamma)$ implies if $\Gamma \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge\ (\boldsymbol{q} \triangleright \boldsymbol{p} : \mathsf{m}(\vec{T}')) \in \Gamma(s)$ then $\mathsf{all}_a\left(\Gamma \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}}\right)$;

φ-$\mathcal{R}$ $\varphi(\Gamma, s[\boldsymbol{p}] : \&_{i\in I}\boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i)$ and $\mathcal{R}(s) = \mathfrak{R}$ implies $\forall k \in I : \boldsymbol{q_k} \in \mathfrak{R}(\boldsymbol{p})$;

φ-μ $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu\mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu\mathrm{t}.S/\mathrm{t}\})$;

φ-$\rightarrow$ $\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$.

$\mathcal{R}$-$\mathsf{safe}(\Gamma)$ is written iff $\varphi(\Gamma)$ and φ is a $\mathcal{R}$-*safety* property.

The change reflecting the shift from unexpected to incompatible messages can be seen in φ-S. The condition states that for safe contexts, if a message m can be input at $\boldsymbol{p}$ from $\boldsymbol{q}$, and a message matching these roles and label exists in the type buffer, then it should be possible for $\boldsymbol{p}$ to dequeue the message. Note that, by rule Γ-Deq, this means that the payload types in the sent

message should be compatible with the types in the branch. This condition should hold for *all* messages in the buffer prefixed with the same message label, formally defined below.

Definition 5.13 (All Async Enabled). A context with transition $\Gamma \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}}$ is said to be enabled for *all* labels m iff a context can perform the transition for every path labelled by m. Formally:

$$\mathsf{all}_a\left(\Gamma \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}}\right) ::= \Gamma \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}} \wedge\ \Gamma = \Gamma_s; \Gamma_0^{\mathcal{B}} \cdot s : \widetilde{M} \wedge \forall (\boldsymbol{p}' \triangleright \boldsymbol{q}' : \mathsf{m}'(\vec{T}')) \in \widetilde{M} :$$
$$\boldsymbol{p}' = \boldsymbol{p} \wedge \boldsymbol{q}' = \boldsymbol{q} \wedge \mathsf{m}' = \mathsf{m} \implies \Gamma_s; (\boldsymbol{p}' \triangleright \boldsymbol{q}' : \mathsf{m}'(\vec{T}')) \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}}$$

Also unique to this version of safety is condition φ-$\mathcal{R}$, stating that if a branch type does not contain a timeout branch, then it must be the case that all roles in the branch are considered reliable. In other words, by taking the contrapositive of the implication, if a message is expected from at least one unreliable role, then the branching type must define a timeout branch. Lastly, conditions φ-μ and φ-$\rightarrow$ require safety to hold after recursive unfolding and context reduction.

Using this definition of safety, subject reduction in MAGπ is presented in Theorem 5.14.

Theorem 5.14 (Subject Reduction). If $P :: \mathcal{B} \rightarrow_{D;\mathcal{R}} P' :: \mathcal{B}'$ and $\Gamma \vdash_D P :: \mathcal{B}$ with $\mathcal{R}$-safe(Γ), then $\exists \Gamma'$ s.t. $\Gamma \rightarrow^* \Gamma'$ and $\Gamma' \vdash_D P' :: \mathcal{B}'$ with $\mathcal{R}$-safe(Γ').

Proof. By induction on the derivation of $P :: \mathcal{B} \rightarrow_{D;\mathcal{R}} P' :: \mathcal{B}'$. The following demonstrates the failure cases.

The assumptions are: **(A1)** $\Gamma \vdash_D P :: \mathcal{B}$; **(A2)** $\mathcal{R}$-safe(Γ); and **(A3)** $P :: \mathcal{B} \rightarrow_{D;\mathcal{R}} P' :: \mathcal{B}'$. The proof follows similar steps to the previous subject reduction proofs. First, the shape of $P :: \mathcal{B}$ and $P' :: \mathcal{B}'$ are inferred from (A3) and the reduction rule in question. Then, by inversion of typing rules and (A1), the shape of the types can be determined. Lastly, (A2) may possibly be used if the context needs to reduce via a communication action, and the reduced process is shown to be typed under a possibly reduced context.

Case R-🕐:

$P :: \mathcal{B} = s[\boldsymbol{q}]\&_{i \in I}[\boldsymbol{p_i}]\mathsf{m}_i(\vec{b_i}) \,.\, Q_i,\ 🕐.\, Q' :: \mathcal{B}$ (by R-🕐 and (A3)) (1)

$P' :: \mathcal{B}' = Q' :: \mathcal{B}$ (by R-🕐 and (A3)) (2)

$\Gamma = \Gamma_P \cdot \Gamma_d; \Gamma_s \vdash_D P :: \mathcal{B}$ (by (1), (A1), T-$\mathcal{B}$) (3)
$\Gamma_P = \Gamma_0 \cdot \Gamma_\& \vdash_D s[q] \&_{i \in I} [p_i] \mathsf{m}_i(\vec{b}_i) \,.\, Q_i, \textcircled{\scriptsize t}.\, Q'$ (by (3), T-$\mathcal{B}$) (4)
$\Gamma_d; \Gamma_s \vdash \mathcal{B}$ (by (3), T-$\mathcal{B}$) (5)
$\Gamma_\& \vdash s[q] : \&_{i \in I} p_i : \mathsf{m}_i(\vec{T}_i).S_i, \textcircled{\scriptsize t}.S'$ (by (4), T-&) (6)
$\Gamma_0 + s[q] : S' \vdash_D Q'$ (by (4), (6), T-&) (7)

$\Gamma_\& = \Gamma'_\&, s[q] : S^\&$ and $\mathsf{end}(\Gamma'_\&)$ (by (6) and T-Wkn$_1$) (8)

Without loss of generality, assume any end-typed channels in Γ_P are split into Γ_0.

$\Gamma_P \to \Gamma_0 + s[q] : S'$ (by (4), (6), (8), Γ-$\textcircled{\scriptsize t}$) (9)
$\Gamma \to \Gamma' = \Gamma_0 + s[q] : S' + \Gamma_d; \Gamma_s$ (by (3), (9), Γ-Cong$_2$) (10)
$\Gamma' \vdash_D P' :: \mathcal{B}$ (by (5), (7), T-$\mathcal{B}$) (11)
$\mathcal{R}$-safe(Γ') (by (A2), (10), and φ-$\to$) (12)

Case R-Drop:

$P :: \mathcal{B} = P :: \mathcal{B}'', s : \widetilde{\mathcal{M}}, \mathcal{M}'$ (by R-Drop and (A3)) (1)
$P' :: \mathcal{B}' = P :: \mathcal{B}'', s : \widetilde{\mathcal{M}}$ (by R-Drop and (A3)) (2)

$\Gamma = \Gamma_P \cdot \Gamma_d; \Gamma_s \vdash_D P :: \mathcal{B}$ (by (1), (A1), T-$\mathcal{B}$) (3)
$\Gamma_P \vdash_D P$ (by (1), (A1), T-$\mathcal{B}$) (4)
$\Gamma_d; \Gamma_s \vdash \mathcal{B}'', s : \widetilde{\mathcal{M}}, \mathcal{M}'$ (by (1), (4), T-$\mathcal{B}$) (5)

$\Gamma_0 (\cdot \Gamma'_{d_i})_{i \in 1..n} \,;\, \Gamma'_s \cdot s : (p \triangleright q : \mathsf{m}((T_i)_{i \in 1..n})), \widetilde{M} \vdash \mathcal{B}'', s : \widetilde{\mathcal{M}}, \mathcal{M}'$ (by (5), T-Msg) (6)
where $\Gamma_d; \Gamma_s = \Gamma_0 (\cdot \Gamma'_{d_i})_{i \in 1..n} \,;\, \Gamma'_s \cdot s : (p \triangleright q : \mathsf{m}((T_i)_{i \in 1..n})), \widetilde{M}$ (cont. of (6)) (7)
and $\mathcal{M}' = (p \triangleright q : \mathsf{m}\langle (d_i)_{i \in 1..n} \rangle)$ (cont. of (7)) (8)

$\Gamma_0; \Gamma'_s + s : \widetilde{M} \vdash \mathcal{B}'', s : \widetilde{\mathcal{M}}$ (by (6) and T-Msg) (9)
$\forall i \in 1..n : \Gamma'_{d_i} \vdash d_i : T_i$ (by (6) and T-Msg) (10)
$\mathsf{gc}((\Sigma_{i \in 1..n} \Gamma'_{d_i}) \,;\, s : (p \triangleright q : \mathsf{m}((T_i)_{i \in 1..n})))$ (by (10) and Definition 5.7) (11)
$\Gamma_d \,;\, \Gamma_s \vdash \mathcal{B}'$ (by (7), (9), (11) and T-Wkn$_2$) (12)
$\Gamma \vdash P' :: \mathcal{B}'$ (by (12), (4) and T-$\mathcal{B}$) (13)

Further cases of the proof can be seen in Theorem B.3 (the subject reduction proof for MAGπ! will subsume this proof). □

5.3.2 Session Fidelity

The fidelity property—on which session fidelity is dependent upon—must also be updated to reflect the change of rejecting incompatible rather than unexpected messages. The updated property is presented in Definition 5.15. With this, session fidelity is presented in Theorem 5.16; the one-role predicate is recalled from the core calculus (Definition 3.20), with the exception that processes are now derived from the restricted grammar of Figure 5.7.

$$\begin{array}{ll} \textit{Processes} & P,Q ::= \sum_{i\in I} P_i^{\oplus} \mid P^{\&} \mid P^{\&},^{\textcircled{t}}.Q \mid \mathrm{X}\langle \vec{V} \rangle \mid \mathbf{0} \\ \textit{Programs} & \mathscr{P} ::= ((\nu s^{\mathfrak{R}} : \Psi)\, P_1 \mid \cdots \mid P_n :: \mathcal{B}, D) \end{array}$$

Figure 5.7: Session Fidelity Restricted Syntax for MAGπ

Definition 5.15 (Fidelity). φ is a $\mathcal{R}$-*fidelity* property on type environment Γ iff:

(i) $\varphi(\Gamma)$ implies if $\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge \Gamma \xrightarrow{\vec{A}} \Gamma' \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}}$ then $\left(\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{r}:\mathsf{m}'(\vec{T}')} \implies \forall \overrightarrow{A'} : \Gamma \xrightarrow{\overrightarrow{A'}} \Gamma'' \xrightarrow{s:\boldsymbol{r}\&\boldsymbol{p}:\mathsf{m}'}\right)$;

(ii) $\varphi(\Gamma, s[\boldsymbol{p}] : \&_{i\in I}\boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i)$ and $\mathcal{R}(s) = \mathfrak{R}$ implies $\forall k \in I : \boldsymbol{q_k} \in \mathfrak{R}(\boldsymbol{p})$;

(iii) $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu \mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu \mathrm{t}.S/\mathrm{t}\})$;

(iv) $\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$.

$\mathcal{R}$-fid(Γ) is written iff $\varphi(\Gamma)$ and φ is a $\mathcal{R}$-*fidelity* property.

Theorem 5.16 (Session Fidelity). Assume $\Gamma \vdash_D P :: \mathcal{B}$ with $\mathcal{R}$-fid(Γ) and one-role(P). Then, $\Gamma \rightarrow$ implies $\exists \Gamma', P', \mathcal{B}'$ s.t. *(i)* $\Gamma \rightarrow \Gamma'$; *(ii)* $P :: \mathcal{B} \rightarrow^*_{D;\mathcal{R}} P' :: \mathcal{B}'$; *(iii)* $\Gamma' \vdash_D P' :: \mathcal{B}'$; and *(iv)* one-role$(P')$.

Proof. (Sketch.) By induction on the derivation of $\Gamma \rightarrow$. There are three possible cases, where Γ is enabled with a reduction by either Γ-Enq, Γ-Deq, or Γ-$\textcircled{t}$. For Γ-Enq, the process is shown to be able to mimic at least one path in the selection (via compatibility). For Γ-$\textcircled{t}$, the process is shown to also define a timeout branch, and therefore can always at least match the timeout action.

For the case of Γ-Deq, if the branch type defines a timeout, then (at least) Γ is also enabled with a timeout action which the process can follow. If the branch does not define a timeout, then by φ-$\mathcal{R}$, it must be that all messages in the branch are reliable. Therefore, any message in the type buffer enabling the dequeue action must also live in the process buffer, since it could not have been dropped by rule R-Drop. Thus, the process could at least mimic one of the dequeue actions enabled on the context.

A detailed version of the proof is given in Theorem B.6 (the MAGπ! proof subsumes this one). □

5.3.3 Decidability

The general result of decidability in MAGπ is similar to that of MPST!, i.e., decidability of typechecking is dependent on the decidability of the chosen safety property.

Theorem 5.17 (Decidablility of typechecking). If φ is decidable, then typechecking is decidable.

Proof. Since typing rules in Figure 5.5 can be deterministically applied based on the structure of a process P, and a typing context need only be split a finite number of times to separate all linear types, there are a finite number of contexts that can be tried for each rule. □

However, as discussed in [11], combining session types with type buffers affects the decidability of typechecking. This previous work shows that partially ordered FIFO buffers and session types result in a formalism capable of encoding Turing machines; whilst decidability results for session types with *bag buffers* are yet to be explored. State-of-the-art generalised MPST theory has provided strategies for identifying protocols that live within a subset of the language in question for which typechecking becomes decidable [102, Appendix G], when considering partially ordered FIFO buffers. The following proves that one of these strategies, *boundedness*, also applies to MAGπ.

Definition 5.18 (Boundedness). It is said that Γ is *k-bound* iff $\exists k \in \mathbb{N} : \Gamma \rightarrow^* \Gamma'$ implies $k > \sum_{s \in \mathsf{dom}(\Gamma')} |\Gamma'(s)|$. Γ is said to be *bounded* iff $\exists k$ *finite*: Γ is *k-bound*.

Theorem 5.19 (Bounded Decidability). Assuming Γ is *bounded*, typechecking under Γ is decidable.

Proof. Since Γ is *bounded*, then the type buffer will only ever contain a finite number of types. Therefore, a finite number of states can be used to represent the type buffer at any point in a context's reduction. This means that Γ has a finite transition system, and thus typechecking is decidable. □

Example 5.20 (Boundedness of the failure-prone load balancer). Observe that the load balancer protocol specified in Example 5.1 is 4-*bound*. (The buffer contains at most 4 messages when the client issues both requests and the server triggers its timeout.) Thus, the example lives within the subset of MAGπ for which typechecking is decidable.

5.4 Key Observations

MAGπ is the first asynchronous MPST language designed using *bag buffers*, *failure semantics* and *imperfect failure detectors*. This section provides further discussion on some key observations from studying this novel setting.

5.4.1 Subtyping vs Compatibility

Standard session subtyping breaks subject reduction when using asynchronous bag buffers.

This chapter presented a different preorder to subtyping: *compatibility* (Definition 5.5). To explain why this is needed, it is important to first understand the relationship between *safety*, *message ordering*, and *subtyping*.

Standard session subtyping allows a process to possibly implement more branches in a receive process than what is specified in the type. For instance, consider the following typing judgement of a type context and process written using the core calculus:

$$s[p] : q\&\mathsf{m},\ s[q] : p\oplus\{\mathsf{m},\mathsf{m}'\} \quad \vdash_D \ s[p][q]\&\{\mathsf{m},\ \mathsf{m}'\} \mid s[q][p]\oplus\mathsf{m}'$$

Observe that: *(i)* the process can take a step that cannot be mimicked by the type; and *(ii)* the process is well-typed under the context because of subtyping. The metatheory is reliant on the *safety property* to reject protocols susceptible to this issue—note that the above context is *not safe* w.r.t. Definition 3.13.

Now consider the following judgement written in MAGπ:

$$\begin{array}{l} s[p] : q\&\mathsf{m}.q\&\mathsf{m}', \\ s[q] : p\oplus\mathsf{m}.p\oplus\mathsf{m}' \end{array} \quad \vdash_D \ s[p]\&[q]\,\mathsf{m}\ .\ s[p]\&[q]\,\mathsf{m}' \mid s[q]\oplus[p]\,\mathsf{m}\ .\ s[q]\oplus[p]\,\mathsf{m}'$$

Indeed, this process is well-typed under the context, which is considered safe by Definition 5.12. If however, compatibility in MAGπ were to be replaced with the standard notion of session subtyping, then the following process would also be well-typed:

$$\begin{array}{l} s[p] : q\&\mathsf{m}.q\&\mathsf{m}', \\ s[q] : p\oplus\mathsf{m}.p\oplus\mathsf{m}' \end{array} \quad \vdash_D \ s[p]\&\left\{\begin{array}{l} [q]\,\mathsf{m}\ .\ s[p]\&[q]\,\mathsf{m}' \\ [q]\,\mathsf{m}'\ .\ P \end{array}\right. \quad \mid s[q]\oplus[p]\,\mathsf{m}\ .\ s[q]\oplus[p]\,\mathsf{m}'$$

Once again, the process can make a reduction which the type cannot match (where message m' is delivered first). Thus, a design choice had to be made: *(i)* do away with subtyping to avoid the issue altogether; or *(ii)* look for a safety property that rejects protocols such as the one described above. This thesis argues that the former is the more practical option for programmers and protocol designers.

Consider a safety property that rejects unexpected messages and apply it to asynchronous bag semantics. Then, a safe alternative to the previous protocol would be the following:

$$s[p] : \; q\&\begin{cases} \mathsf{m}.q\&\mathsf{m}' \\ \mathsf{m}'.q\&\mathsf{m} \end{cases}, \; s[q] : p \oplus \mathsf{m}.p \oplus \mathsf{m}'$$

The issue with this becomes obvious once the sender type is extended.

$$s[p] : \; q\&\begin{cases} \mathsf{m}.q\&\begin{cases} \mathsf{m}'.q\&\mathsf{m}'' \\ \mathsf{m}''.q\&\mathsf{m}' \end{cases} \\ \mathsf{m}'.q\&\begin{cases} \mathsf{m}.q\&\mathsf{m}'' \\ \mathsf{m}''.q\&\mathsf{m} \end{cases} \\ \mathsf{m}''.q\&\begin{cases} \mathsf{m}.q\&\mathsf{m}' \\ \mathsf{m}'.q\&\mathsf{m} \end{cases} \end{cases}, \; s[q] : p \oplus \mathsf{m}.p \oplus \mathsf{m}'.p \oplus \mathsf{m}''$$

Indeed, in order for the context to remain safe (for some definition that rejects unexpected messages), the size of the branch grows exponentially w.r.t. the number of messages sent.

For this reason, MAGπ opts to replace subtyping with a preorder that does not allow for larger branches, and uses a weaker safety property which accepts unexpected messages (rejecting only *incompatible* messages). It is these design choices that also allow for the introduction of undirected branching.

5.4.2 Failure Models

Timeouts are agnostic of failure models.

This is not a novel observation, however it is interesting to examine the extent of generality gained through timeouts as a failure-detection mechanism.

Firstly, MAGπ uses timeouts to model message loss in the type semantics without needing to directly remove types from the type buffer. The inspiration for this is drawn from the two generals problem [2], i.e., it is impossible to distinguish between messages that are lost or delayed. Therefore, the type semantics model both using the same construct. Message loss is modelled by taking a timeout branch when a valid message could have been dequeued, and leaving the message in the type buffer in all future paths. Note, that this also models message delay with an infinite upper-bound. This is demonstrated in Example 5.21. The benefit of this approach being that it models the real worst-case scenario; the drawback being the complexity of verification (as discussed in Section 5.3.3).

Example 5.21 (Message loss vs message delay). Consider a client p making a request to a service. The service offers two endpoints (q, r), allowing for a degree of fault tolerance (i.e., if a request to one endpoint fails, clients may try the other). The client attempts to reach each endpoint in-turn, reporting the result to an output role o. The following are types in a context Γ, where p and o can communicate reliably, and q and r have the same type.

$$s[p] : \oplus q\text{:req}.\& \left\{ \begin{array}{l} q\text{:res}.\oplus o\text{:res} \\ \clock.\, \oplus r\text{:req}.\& \left\{ \begin{array}{l} r\text{:res}.\oplus o\text{:res} \\ q\text{:res}.\oplus o\text{:res} \\ \clock.\, \oplus o\text{:ko} \end{array} \right. \end{array} \right. \qquad s[q], s[r] : \& \left\{ \begin{array}{l} p\text{:req}.\oplus p\text{:res} \\ \clock.\, \mathbf{end} \end{array} \right. \qquad s[o] : \& \left\{ \begin{array}{l} p\text{:res} \\ p\text{:ko} \end{array} \right.$$

Note that $\Gamma \to^* \Gamma'$ such that p sends the request and then triggers its timeout, thus Γ' contains:

$$s[p] : \oplus r\text{:req}.\& \left\{ \begin{array}{l} r\text{:res}.\oplus o\text{:res} \\ q\text{:res}.\oplus o\text{:res} \\ \clock.\, \oplus o\text{:ko} \end{array} \right. \qquad s[q], s[r] : \& \left\{ \begin{array}{l} p\text{:req}.\oplus p\text{:res} \\ \clock.\, \mathbf{end} \end{array} \right. \qquad s[o] : \& \left\{ \begin{array}{l} p\text{:res} \\ p\text{:ko} \end{array} \right. \qquad s : \{(p \triangleright q : \mathsf{req}())\}$$

To consider the loss of message $(p \triangleright q : \mathsf{req}())$ is to consider all paths from Γ' where the message remains in the type buffer, i.e., $\{\Gamma'' \mid \Gamma' \to^* \Gamma'' \wedge (p \triangleright q : \mathsf{req}()) \in \Gamma''(s)\}$. This is indistinguishable from an infinite message delay.

Table 5.1 generalises message loss from the previous example and formalises various failure models as paths of a context. These paths capture subgraphs of the entire LTS produced by a context that could be considered in isolation to study a protocol under different failure models. This demonstrates how nondeterministic timeouts can generalise over various types of failures.

5.5 Network Assumptions

An aim of this thesis is to study the efficacy of MPST for verifying behavioural properties of programs designed to run over failure-prone networks. To this end, MAGπ has been designed to model message loss, delay, and (total) reordering, and has a type system that has been shown to be capable of verifying communication-centric properties of protocols using timeouts as imperfect failure detectors. With high confidence, this thesis affirms the ability of using MPST in this setting. What is left to be discussed is how this chosen setting relates to real-world network protocols.

This section now briefly discusses a range of network protocols, highlighting how MAGπ

Message loss	Loss of message M from Γ where $M \in \Gamma(s)$ is captured by all reductions $\{\Gamma' \mid \Gamma \to^* \Gamma' \wedge M \in \Gamma'(s)\}$, i.e., all paths from Γ where M remains in the type buffer.
Crash-stop	Crash-stop failure of a role $\boldsymbol{p}$ in s at point Γ is captured by all reductions $\{\Gamma' \mid \Gamma \xrightarrow{\vec{A}}{}^* \Gamma' \wedge \forall s{:}\boldsymbol{p}\oplus\boldsymbol{q}{:}\mathsf{m} \in \vec{A} : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T})) \in \Gamma'(s)\}$ for any $\boldsymbol{q}, \mathsf{m}, \vec{T}$, i.e., all paths from Γ where every message sent from $\boldsymbol{p}$ is lost.
Link failure	Link failure between roles $\boldsymbol{p}$ and $\boldsymbol{q}$ in s at point Γ is captured by all reductions $\{\Gamma' \mid \Gamma \xrightarrow{\vec{A}}{}^* \Gamma' \wedge \forall s{:}\boldsymbol{p}\oplus\boldsymbol{q}{:}\mathsf{m} \in \vec{A} : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T})) \in \Gamma'(s) \wedge \forall s{:}\boldsymbol{q}\oplus\boldsymbol{p}{:}\mathsf{m}' \in \vec{A} : (\boldsymbol{q} \triangleright \boldsymbol{p} : \mathsf{m}'(\vec{T}')) \in \Gamma'(s)\}$ for any $\mathsf{m}, \vec{T}, \mathsf{m}', \vec{T}'$, i.e., all paths from Γ where every message sent between $\boldsymbol{p}$ and $\boldsymbol{q}$ is lost.

Table 5.1: Failure models as context paths.

can be reconfigured or extended in order to model communication over each protocol.

5.5.1 User Datagram Protocol

The User Datagram Protocol (UDP) [93] is a transport layer protocol designed to favour speed of transmission over reliability. The protocol does not implement any *handshaking*, and is susceptible to non-Byzantine faults.

MAGπ has been designed to model communication over a network with similar assumptions to using UDP. Out of the box, MAGπ uses bag buffers (modelling total message reordering), and directly simulates message loss through the process semantics (rule R-Drop). Furthermore, using MAGπ's configurable notion of reliability, any links in a program modelling communication of distributed processes can be treated as *unreliable*.

There is, however, one missing link preventing MAGπ from completely modelling the assumptions of UDP: *message duplication*, which is left to future work. At the current stage, it appears as if extending MAGπ to model message duplication is non-trivial and will require dedicated study, which is why it was not considered for this thesis. The challenges lie in identifying sublanguages for which the generalised type system is decidable.

To elaborate, consider a naïve approach where messages in the process buffer can nondeterministically duplicate. Then, to type these extra messages in the process buffer, one approach may be to also simulate message duplication in the type semantics. This would allow the types to simulate any message duplication that may occur at runtime. The issue with this approach is that buffer boundedness is no longer an effective method for determining whether a protocol lives within a sublanguage of MAGπ for which type checking is decidable (since messages in

the buffer may arbitrarily duplicate). Research questions that could be investigated are: *(i)* Are there other techniques for establishing whether a protocol lives within a sublanguage of MAGπ (with duplication) for which type checking is decidable? And, *(ii)* are there other methods of modelling such a setting without directly simulating message duplication in the type semantics?

5.5.2 Reliable Data Protocol

The Reliable Data Protocol (RDP) [52] is a transport layer protocol designed to acknowledge messages using handshakes, and prioritises transfer of data within packets by reducing the complexity of packet headers (e.g. by removing sequencing number). The result is a protocol which guarantees message delivery, but not order. These network assumptions can be modelled exactly in MAGπ using fully reliable configurations for sessions, guaranteeing that no message loss occurs. (Message reordering is still modelled using bag buffers.)

5.5.3 Real-Time Transport Protocol

The Real-Time Transport Protocol (RTP) [44] is a network protocol built on-top of UDP, to provide a fast method of transferring data in-order at the cost of reliability. Mainly used for transferring video and audio, the protocol guarantees ordered delivery with the possibility of message loss.

Modelling communication over RTP in MAGπ would require adjustments to aspects of the language. The following outlines some necessary changes to the language design (although the metatheory is not re-proven using these definitions).

Given MAGπ already models message loss, the changes required are focused towards guaranteeing ordered transmission of messages. This could be simulated by adopting *queue* buffers, as opposed to bags, as is standard in most asynchronous theories. Following work by Scalas and Yoshida [103, Section 7], queue buffers can be enabled with *congruence rules* allowing for *partial message reordering*, such that messages in the queue may only swap position if they involve different participants. It is conjectured that the safety property may also be updated to reflect guarantees of ordering—it would be interesting to explore whether the standard notions of safety and subtyping can be used within such a system.

5.5.4 Transmission Control Protocol

The Transmission Control Protocol (TCP) [96] is a transport layer protocol guaranteeing both delivery and order of packets. It is the standard protocol simulated by asynchronous session type theories. Following the adjustments mentioned for RTP, MAGπ could model communication over TCP if it were enabled with queue buffers and used fully reliable configurations for established sessions. Doing this would result in a formalism similar to that of Scalas and

Yoshida [103, Section 7], the very work which the core calculus (Chapter 3) is based upon.

5.6 Related Work

MAGπ is the *first* asynchronous MPST theory to consider total message reordering via bag buffers. Furthermore, it is unique w.r.t. its failure model compared to any other work studying the use of MPST in failure-prone settings—i.e., it is the first MPST theory to integrate nondeterministic timeouts into types as *imperfect failure detectors*. The related work discussed below gives an overview of different approaches to using MPST in failure-prone settings.

One method of reasoning about arbitrary failures in types is through the use of *affine* type systems. Affine types describe resources that must be used *at most once* (rather than the "*exactly once*" imposed by linear typing). *Affine session types* [65, 84, 38, 22] describe session protocols that may be prematurely cancelled in the event of runtime failure, and possibly also describe a new protocol to be followed in its stead. A benefit of affine session types is that they may be agnostic to the type of error experienced at runtime, offloading the responsibility of error-detection to other mechanisms (which are generally assumed to provide accurate feedback). Compared to MAGπ, affine type systems address a different set of issues altogether—they aim to enable application-level programmers with expressive error-handling mechanisms, whereas MAGπ aims to provide a verification framework for low-level network programming.

Common failure models considered are those of *crash-stop* and *crash-recovery*, where the unit of failure is a node crash. This, of course, assumes a higher level of abstraction than assumed by MAGπ—where the unite of failure is message loss. The crash-stop model assumes that nodes may, at any point, crash and remain inoperable for the remainder of the program's runtime; whilst crash-recovery assumes that crashed nodes may at some point be restarted.

Viering et al. [116] present a MPST theory for event-driven distributed systems assuming the crash-recovery failure model. The theory assumes the existence of a centralised robust monitor, responsible for observing all other processes and restarting them if they crash. Chen et al. [25] remove the need for a centralised reliable node, instead equipping a type system with *synchronisation points*; allowing participants to return to specific points in their protocols and retry communication if failures are detected. These works demonstrate effective techniques for describing communication protocols that should be retried until successful, relying on other mechanisms to detect and recover processes as appropriate. On the other hand, MAGπ provides low-level techniques for designing protocols which cannot assume such mechanisms.

Most similar to MAGπ is work by Barwell et al. [13, 12], where generalised session type theory is extended to reason about crash-stop failures. Their approach is unique w.r.t. the previously discussed works as failure detection is not offloaded to a central reliable system, but rather occurs within each participant individually and can affect the progression of a participant's type (as in MAGπ). For instance, a participant may be waiting for a message, realise that said mes-

sage will not arrive because of some failure, and instead initiate a failure-handling subprotocol. The main difference between these works and MAGπ is the underlying failure model, which has significant impact on expressivity and decidability. In a language and type system using asynchronous queue semantics and crash-stop failures [12], the top-down approach to MPST can be leveraged to obtain a decidable sublanguage for specifying *live* protocols. On the other hand, MAGπ models more low-level network failures, but the only technique that has so far been identified for restricting to a decidable sublanguage is by verifying boundedness of the type buffer.

Whereas the aforementioned work (as well as MAGπ) handle failure by specifying failure-handling subprotocols, a different approach is to use *default values*. Adameit et al. [1] consider an environment free from a centralised reliable node where unstable *links* between participants can fail. Their type system is equipped with the concept of *optional blocks*, outlining parts of a protocol that may be susceptible to failures. Payloads that should be received within said optional blocks can be annotated with *default values*, substituting the data that is not received due to communication failure; guaranteeing termination and deadlock-freedom. This work is generalised to MPST with a more fine-grained failure model by Peters et al. [89]. The extended work introduces type-level *failure annotations*, allowing users to specify (on a per-action basis) whether: *(i)* nodes may crash and messages can be lost; *(ii)* nodes may crash but messages cannot be dropped; or *(iii)* both nodes and messages are reliable. Lost messages are handled via default values, whilst node crashes are handled via default branches (i.e., failure-handling subprotocols, similar to timeout branches in MAGπ).

5.7 Conclusion

This chapter presented MAGπ—a Multiparty, Asynchronous and Generalised π-calculus—the first integration of timeouts into MPST as *imperfect failure detectors* with *asynchronous bag semantics* and *failure-prone* communication. The study has uncovered interesting technical details about session types for *asynchronous bag semantics*, e.g. standard subtyping breaks subject reduction even without undirected branching/selection; and the safety of undirected branching is dependent on the preorder chosen for subtyping. By default, the language has been configured to model a communication setting similar to the User Datagram Protocol, and strategies to how aspects of the language can be adapted were given for modelling communication over various network protocols. In order to have a focused study on typing failure-prone communication, replication and first-class roles were not introduced in MAGπ; implying that certain impracticalities outlined in Chapter 4 have been reintroduced.

The next chapter addresses these issues, by merging the language features of MPST! and MAGπ into a single language. The resulting formalism uncovers how replication plays a vital role in designing fault-tolerant client-server systems. The developed language, MAGπ!, lever-

ages replication to simplify the design of fault-tolerant protocols and demonstrates how its type system can be used to specify real-world network protocols.

Chapter 6

Realism: Typing Fault-Tolerant Servers

6.1 Introduction

"Failures in distributed communication are inevitable"—network protocols are designed with this philosophy in mind, and are appropriately built to be *fault-tolerant*. Fault tolerance refers to the use of *redundancy* within a computational system in order to achieve some equivalence in output when comparing program traces both with and without a degree of failure [68]. For example, in a system with stateless participants and crash-stop failures, fault tolerance may refer to having n copies of a service provider to withstand $n-1$ faults (node crashes of said service providers) [40]. For stateful systems susceptible to non-Byzantine faults, consensus algorithms (e.g. Paxos [66], Raft [86]) provide fault tolerance up-to f simultaneous node crashes in systems of $2f+1$ nodes. For more low-level systems where message-loss is the unit of failure (the focus of this thesis), the client-server paradigm [104, 112] provides fault tolerance by implementing redundancy through the use of *infinitely available servers*.

This chapter re-introduces replication and first-class roles, this time as an extension to MAGπ instead of the core calculus. The new language—MAGπ!—not only benefits from the practicality and expressivity of these constructs as demonstrated in Chapter 4, but also allows for the specification of fault-tolerant client-server systems. This is a result of the inherently fault-tolerant nature of replication; e.g. a server does not need to handle dropped client requests, as it will remain available to handle any retries issued by the client. This methodology is central to languages such as Erlang [7, 23] and Elixir [109], which are used in practice to build large, scalable and fault-tolerant distributed applications.

Example 6.1 (Fault-Tolerant Load Balancer). Recall the load balancer examples written in MPST! (Example 4.2) and MAGπ (Example 5.1). These approaches are now blended together to write a protocol for a fault-tolerant load balancer, written in MAGπ!. The example uses replication and first-class roles to define infinitely available servers in a

system susceptible to failure, demonstrating the inherently fault-tolerant nature of replication. Consider the reliability configuration $\mathfrak{R}_{lb}$ below:

$$\mathfrak{R}_{lb} \;=\; c \mapsto \{o\},\; o \mapsto \{c\},\; srv \mapsto \{w_1, w_2\},\; w_1 \mapsto \{srv\},\; w_2 \mapsto \{srv\}$$

The load balancer in consideration includes one server srv which can reliably forward client requests to one of two workers w_1 or w_2. The client c makes requests to the server, reporting the output to role o. If the server is successfully reached and an answer is obtained, then the client reports the answer back to o; otherwise it sends o a noop. The protocol for the fault-tolerant load balancer is described as:

$$\Psi_{lb} = \{srv : S_{srv},\; w_1 : S_w,\; w_2 : S_w,\; c : S_c,\; o : S_o\}$$

$$S_{srv} = !\alpha\&\mathsf{req}(\mathtt{Int}).\oplus \begin{cases} w_1 : \mathsf{fw}(\mathtt{Int}, \alpha).\mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}, \alpha).\mathbf{end} \end{cases}$$

$$S_w = !srv\&\mathsf{fw}(\mathtt{Int}, \beta).\oplus \beta : \mathsf{ans}(\mathtt{String}).\mathbf{end}$$

$$S_c = \mu \mathrm{X}.\oplus \begin{cases} srv : \mathsf{req}(\mathtt{Int}).\& \begin{cases} w_1 : \mathsf{ans}(\mathtt{String}).\oplus o : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ w_2 : \mathsf{ans}(\mathtt{String}).\oplus o : \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \clock.\mathrm{X} \end{cases} \\ o : \mathsf{noop}().\mathbf{end} \end{cases}$$

$$S_o = \&c : \begin{cases} \mathsf{ans}(\mathtt{String}).\mathbf{end} \\ \mathsf{noop}().\mathbf{end} \end{cases}$$

The protocol captures a failure-tolerant load balancer servicing client requests, where clients may attempt their request *any number* of times.

The client type begins with a recursive binder, immediately followed by a selection of either sending the request or reporting a noop. If the request is sent, it waits for the unreliable response from either worker. A received response instructs the client to send the answer to the output; otherwise it triggers its timeout, restarting the protocol.

The server is defined as a universal receive, waiting for a request from any client, binding role α to the client name that messages it within its continuation type. After receiving a request, the server forwards it to one of the workers. Importantly, the server no longer defines a timeout branch—the server is replicated, and thus it can safely be assumed that the process will be available to handle the next client request if the client decides to retry. This way, the server is made fault-tolerant (up to f dropped requests for clients that retry $f+1$ times). In fact, observe that even if communication between the server and workers is unreliable, the protocol would not need to change (since workers are also replicated).

Contributions

The contribution of this chapter is a MPST theory integrating together for the first time *replication* and *first-class roles* with *asynchronous bag semantics* and *message inconsistencies*, as a means of formalising, and verifying, fault-tolerant client-server protocols. Specifically:

1. Section 6.2 presents MAGπ!, combining the novel features of the previously studied languages MPST! and MAGπ. The language demonstrates how replication not only promotes modular design of components in client-server systems, but also that replication simplifies failure-handling in servers as they are considered fault-tolerant by nature. The metatheory of the language is proven in Section 6.3.

2. MAGπ! is expressive enough to describe real-world distributed protocols. This is demonstrated in Section 6.4, where the practicality of the type system is tested by formalising a number of examples inspired by distributed network protocols.

This chapter does not present novel features that have not been introduced in the previous chapters. Rather it adapts the concepts of Chapter 4 to the setting introduced in Chapter 5. For this reason, with the aim of reducing repetition, the chapter only presents the syntax and semantics of MAGπ!, moving typechecking and other definitions to appendix B. Furthermore, related work for this chapter is omitted as it is identical to those of the previous two chapters combined. Lastly, Section 6.5 concludes the chapter.

6.2 MAGπ!

This section combines the extensions of the previous two chapters under a single language, MAGπ!, demonstrating how replication naturally integrates into designing fault-tolerant communication protocols.

For simplicity, the language presented assumes the session fidelity assumptions—single session communication with no delegation—throughout the entire chapter. It is key to note that the restricted language is sufficient for demonstrating the key novelties of the integration of these extensions; allowing for multiple sessions will only increase the number of *safe* protocols captured by the type system, and has no effect on property verification. All examples shown live in the restricted language.

6.2.1 Syntax

Figure 6.1 shows the syntax for MAGπ!, combining the novel constructs of the previous two chapters—i.e., replication and first-class roles, with bag buffers and timeouts.

$$
\begin{array}{rl}
\textit{Payload Data} & d ::= v \mid \boldsymbol{q} \\
\textit{Concretes} & r ::= v \mid \boldsymbol{q} \mid s[\boldsymbol{q}] \\
\textit{Channels} & c ::= x \mid s[\boldsymbol{q}] \\
\textit{Binders} & b ::= x \mid \boldsymbol{\alpha} \\
\textit{Names} & a ::= c \mid \boldsymbol{\alpha} \\
\textit{Values} & V ::= b \mid r \\
\textit{Roles} & \boldsymbol{\rho} ::= \boldsymbol{q} \mid \boldsymbol{\alpha}
\end{array}
$$

$$
\begin{array}{rl}
\textit{Processes} & P, Q ::= \textstyle\sum_{i\in I} P_i^{\oplus} \mid P^{\&} \mid P^{\&}, \textcircled{\scriptsize t}.Q \mid !P^{\&} \mid P^{!} \mid \mathrm{X}\langle \vec{V} \rangle \mid \mathbf{0} \\
\textit{Send Process} & P^{\oplus} ::= c \oplus [\boldsymbol{\rho}]\, \mathsf{m}\langle \vec{V} \rangle \,.\, P \\
\textit{Receive Process} & P^{\&} ::= c \,\&_{i\in I} [\boldsymbol{\rho_i}]\, \mathsf{m}_i(\vec{b_i}) \,.\, P_i \\
\textit{Universal Receive} & P^{!} ::= !c[\boldsymbol{\alpha}] \,\&_{i\in I} \mathsf{m}_i(\vec{b_i}) \,.\, P_i
\end{array}
$$

$$
\begin{array}{rl}
\textit{Programs} & \mathscr{P} ::= ((\nu s^{\mathfrak{R}} : \Psi)\, \mathbb{P} :: \widetilde{\mathcal{M}}, D) \\
\textit{Top-Level Processes} & \mathbb{P} ::= P_1 | \cdots | P_n \\
\textit{Definitions} & D ::= \emptyset \mid D, \mathrm{X} \mapsto (\vec{b}, \vec{T}, P) \\
\textit{Protocols} & \Psi ::= \{\boldsymbol{p} : S_{\boldsymbol{p}}\}_{\boldsymbol{p}\in I} \\
\textit{Reliability} & \mathfrak{R} ::= \emptyset \mid \mathfrak{R}, \boldsymbol{p} : \{\boldsymbol{q_i}\}_{i\in I} \\
\textit{Messages} & \mathcal{M} ::= (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle \vec{d} \rangle)
\end{array}
$$

Figure 6.1: Syntax of MAGπ!

Names, values, and binders. A *session name*, ranged over by $s, s', \ldots$, represents a collection of interconnected participants. A *role* is a participant in a multiparty communication protocol, and each *communication endpoint* $s[\boldsymbol{q}]$ is obtained by indexing a session name with a role.

MAGπ! supports the novel feature of *first-class* roles, meaning that a role may be communicated as part of a message. To this end, a role $\boldsymbol{\rho}$ may either be a concrete role ($\boldsymbol{p}$, $\boldsymbol{q}$, $\boldsymbol{r}$,...) or a *role variable* ($\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$,...).

Payload data d is used when sending a message (at runtime) and is either a basic value, or a concrete role. (Endpoints are no longer required in payloads as the session fidelity restrictions prevent them from being used.) Concretes are basic values, roles, or session with role. They are the runtime arguments accepted by recursive calls. Channels c represent endpoints, which can either be a specific session with role, or abstracted by a variable. The binders b are either variables x (abstracting either endpoints or basic values), or role variables $\boldsymbol{\alpha}$.

Lastly, a name a is either an endpoint, a variable, or a role variable, whereas a value V is either a channel, a basic value, or a role.

Processes. Processes are ranged over by $P, Q, R, \ldots$:

Choice-guarded output $\sum_{i\in I} c \oplus [\boldsymbol{\rho_i}]\, \mathsf{m}_i\langle \vec{V}_i\rangle \,.\, P_i$, allows a nondeterministic send along c to role $\boldsymbol{\rho_i}$ with label m_i and payload $\vec{V}_i$, with the process continuing as P_i. (Again, still observing the well-formedness condition that all channels in the choice should be the same.) *Branching receive* $c\&_{i\in I}[\boldsymbol{\rho_i}]\, \mathsf{m}_i(\vec{b}_i) \,.\, P_i$ denotes a process waiting on channel c for one of a set of messages from role $\boldsymbol{\rho_i}$ with label m_i, binding the received data to variables $\vec{b}_i$ before continuing according to P_i. Like in MAGπ, undirected branching (the ability to expect a message from different participants) is allowed. Furthermore, as in MPST!, the subject of a communication action is indicated via $\boldsymbol{\rho}$, which can either be a hard-coded role, or a role variable.

In MAGπ!, the receive process may *either* be prefixed with a bang !, identifying it as replicated; *or* postfixed with a timeout branch $\mathbin{⏲}.\, Q$. Key to the results of this language is the separation of replication and timeouts—i.e., a replicated process need not define a failure-handling timeout branch as it is inherently fault-tolerant. This will be expanded upon later; for now it suffices to observe that the language syntax purposefully does not allow timeout branches on replicated processes.

Universal receive $c[\boldsymbol{\alpha}]\&_{i\in I}\mathsf{m}_i(\vec{b}_i) \,.\, P_i$ denotes a process waiting on channel c for one of a set of messages m_i from *any* role, binding the sender role to $\boldsymbol{\alpha}$ and received data to variables $\vec{b}_i$ before continuing according to P_i. Universal receive is an explicit construct in MAGπ! to isolate it from undirected branching. Therefore, the language supports either the ability to wait for a message from any role (universal receive), or the ability to expect a message from a number of different (statically specified) roles; but not both together.

Lastly, as in the previously discussed languages, *process call* $\mathrm{X}\langle \vec{V}\rangle$ models recursion and $\mathbf{0}$ denotes the *empty process*.

Programs, Definitions, and Protocols. As previously mentioned, programs in MAGπ! will be assumed to abide by the session fidelity assumptions throughout the entire chapter. Thus, the main process of a program takes the form $(\nu s^{\mathfrak{R}} : \Psi)\, P_1 | \cdots | P_n :: \widetilde{\mathcal{M}}$.

Breaking it down, this refers to a single session program $(\nu s^{\mathfrak{R}} : \Psi)$ with reliability configuration $\mathfrak{R}$ and session protocol Ψ. Unchanged from MAGπ, the reliability configuration is a mapping from roles to (possibly empty) sets of roles $(\boldsymbol{p} : \{\boldsymbol{q_i}\}_{i\in I})$, defining any reliable links in the session. Protocol Ψ is a set of role-session type pairs, $\{\boldsymbol{p} : S_{\boldsymbol{p}}\}_{\boldsymbol{p}\in I}$ for a non-empty I, defining the communication patterns that each participant of a session should follow.

The restriction is hiding a parallel composition of processes $P_1 | \cdots | P_n$. These will be the processes playing distinct roles within the single session. Having parallel composition as a top-level construct removes the ability to fork processes at runtime. However, this would go against session fidelity assumptions anyway, so no expressivity is lost w.r.t. the state-of-the-art, whilst simplifying the metatheory.

The main process is initialised with a buffer. Since programs are guaranteed to only contain a

$$
\begin{array}{rl}
\textit{Static types} & T ::= S \mid B \mid \boldsymbol{\rho} \\
\textit{Basic types} & B ::= \texttt{Nat} \mid \texttt{Real} \mid \texttt{String} \mid \cdots \\
\textit{Session types} & S ::= S^{\oplus} \mid S^{\&} \mid !S^{\&} \mid S^{\&}, \textcircled{t}.S' \mid S^{!} \mid \mu \mathrm{t}.S \mid \mathrm{t} \mid \mathbf{end} \\
\textit{Selection type} & S^{\oplus} ::= \oplus_{i \in I} \boldsymbol{\rho_i} : \mathsf{m}_i(\vec{T}_i).S_i \\
\textit{Branching type} & S^{\&} ::= \&_{i \in I} \boldsymbol{\rho_i} : \mathsf{m}_i(\vec{T}_i).S_i \\
\textit{Universal branching type} & S^{!} ::= !\boldsymbol{\alpha} \&_{i \in I} \mathsf{m}_i(\vec{T}_i).S_i \\
\textit{Role singletons} & \boldsymbol{\rho} ::= \boldsymbol{q} \mid \boldsymbol{\alpha} \\
\textit{Runtime types} & U ::= S \mid (U_1 | U_2) \\
\textit{Message type} & M ::= (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T}))
\end{array}
$$

Figure 6.2: Syntax of MAGπ! Types

single session, buffers in MAGπ! are simplified to multisets of messages $\widetilde{\mathcal{M}}$ (omitting the session mapping). For a program to be well-formed, there should be no messages in the initial buffer (i.e., $\widetilde{\mathcal{M}} = \emptyset$).

Lastly, process definitions are standard, mapping process names (ranged over by X,Y,...) to process bodies $(\vec{b}, \vec{T}, P)$ to be used for recursion, where $\vec{b}$ is a list of accepted arguments, $\vec{T}$ is a list of types for said arguments, and P is the process in which the arguments are used.

Types

The syntax of MAGπ! types is given in Figure 6.2.

Combining the novel features of the previous two chapters, session types in MAGπ! support all the standard constructs along with *first-class roles*, *universal receives*, *nondeterministic time-outs* attached to receive branches, as well as *runtime types* and *message types*. Definition 6.2 defines type contexts to be identical to those used in MPST!, extended with type-buffers.

Definition 6.2 (Type Context). Type context Γ maps endpoints to runtime types, variables to static types, session names to multisets of message types (acting as the type-buffer), and may contain role-variable singletons.

$$\Gamma ::= \emptyset \mid \Gamma, s[\boldsymbol{p}] : U \mid \Gamma, x : T \mid \Gamma, s : \widetilde{M} \mid \boldsymbol{\alpha} : \boldsymbol{\alpha}$$

6.2.2 Semantics

Using the standard notion of structural congruence for processes (Figure 3.5), process reduction for MAGπ! is formalised in Definition 6.3.

Process reduction $\boxed{\mathbb{P} :: \widetilde{\mathcal{M}} \to_{D,\mathfrak{R}} \mathbb{P}' :: \widetilde{\mathcal{M}}'}$

R-SND
$$s[\mathbf{p}] \oplus [\mathbf{q}]\,\mathsf{m}\langle \vec{d} \rangle \,.\, P :: \widetilde{\mathcal{M}} \;\to_{D;\mathfrak{R}}\; P :: \widetilde{\mathcal{M}}, (\mathbf{p} \triangleright \mathbf{q} : \mathsf{m}\langle \vec{d} \rangle)$$

R-+
$$\frac{j \in I}{\sum_{i\in I} P_i^{\oplus} :: \widetilde{\mathcal{M}} \;\to_{D;\mathfrak{R}}\; P_j^{\oplus} :: \widetilde{\mathcal{M}}}$$

R-RCV
$$\frac{\mathcal{M}' = (\mathbf{p_k} \triangleright \mathbf{q} : \mathsf{m}_k\langle \vec{d} \rangle) \qquad k \in I}{s[\mathbf{q}]\&_{i\in I}[\mathbf{p_i}]\mathsf{m}_i(\vec{b_i}) \,.\, P_i\ [,\circlearrowright.\ Q] :: \widetilde{\mathcal{M}}, \mathcal{M}' \;\to_{D;\mathfrak{R}}\; P_k\{\vec{d}/\vec{b_k}\} :: \widetilde{\mathcal{M}}}$$

R-|
$$\frac{P :: \widetilde{\mathcal{M}} \to_{D;\mathfrak{R}} P' :: \widetilde{\mathcal{M}}'}{P \mid \mathbb{P} :: \widetilde{\mathcal{M}} \to_{D;\mathfrak{R}} P' \mid \mathbb{P} :: \widetilde{\mathcal{M}}'}$$

R-!$_1$
$$\frac{R = !s[\mathbf{q}]\&_{i\in I}[\mathbf{p_i}]\mathsf{m}_i(\vec{b_i}) \,.\, P_i \qquad \mathcal{M}' = (\mathbf{p_k} \triangleright \mathbf{q} : \mathsf{m}_k\langle \vec{d} \rangle) \qquad k \in I}{R :: \widetilde{\mathcal{M}}, \mathcal{M}' \;\to\; R \mid P_k\{\vec{d}/\vec{b_k}\} :: \widetilde{\mathcal{M}}}$$

R-!$_2$
$$\frac{R = !s[\mathbf{q}][\boldsymbol{\alpha}]\&_{i\in I}\mathsf{m}_i(\vec{b_i}) \,.\, P_i \qquad \mathcal{M}' = (\mathbf{p} \triangleright \mathbf{q} : \mathsf{m}_k\langle \vec{d} \rangle) \qquad k \in I}{R :: \widetilde{\mathcal{M}}, \mathcal{M}' \;\to\; R \mid P_k\{\vec{d}/\vec{b_k}\}\{\mathbf{p}/\boldsymbol{\alpha}\} :: \widetilde{\mathcal{M}}}$$

R-$\circlearrowright$
$$s[\mathbf{q}]\&_{i\in I}[\mathbf{p_i}]\mathsf{m}_i(\vec{b_i}) \,.\, P_i,\ \circlearrowright.\ Q :: \widetilde{\mathcal{M}} \;\to_{D;\mathfrak{R}}\; Q :: \widetilde{\mathcal{M}}$$

R-DROP
$$\frac{\mathcal{M}' = (\mathbf{p} \triangleright \mathbf{q} : \mathsf{m}\langle \vec{d} \rangle) \qquad \mathbf{q} \notin \mathfrak{R}(\mathbf{p})}{\mathbb{P} :: \widetilde{\mathcal{M}}, \mathcal{M}' \;\to_{D;\mathfrak{R}}\; \mathbb{P} :: \widetilde{\mathcal{M}}}$$

R-X
$$\frac{D(\mathrm{X}) = (\vec{x}, \vec{T}, P)}{\mathrm{X}\langle \vec{d} \rangle :: \widetilde{\mathcal{M}} \;\to_{D;\mathfrak{R}}\; P\{\vec{d}/\vec{x}\} :: \widetilde{\mathcal{M}}}$$

R-$\equiv$
$$\frac{\mathbb{P}_1 :: \widetilde{\mathcal{M}} \;\to_{D;\mathfrak{R}}\; \mathbb{P}_2 :: \widetilde{\mathcal{M}}' \qquad \mathbb{P}_1 :: \widetilde{\mathcal{M}} \equiv \mathbb{P}_1' :: \widetilde{\mathcal{M}} \qquad \mathbb{P}_2 :: \widetilde{\mathcal{M}}' \equiv \mathbb{P}_2' :: \widetilde{\mathcal{M}}'}{\mathbb{P}_1' :: \widetilde{\mathcal{M}} \;\to_{D;\mathfrak{R}}\; \mathbb{P}_2' :: \widetilde{\mathcal{M}}'}$$

Figure 6.3: Operational semantics MAGπ!.

Definition 6.3 (Process reduction). A single session process $(\nu s^{\mathfrak{R}})\mathbb{P} :: \widetilde{\mathcal{M}}$ reduces iff its restricted process with buffer reduces parametric on some process definitions D and reliability configuration $\mathfrak{R}$, written $\mathbb{P} :: \widetilde{\mathcal{M}} \to_{D;\mathfrak{R}} \mathbb{P}' :: \widetilde{\mathcal{M}}'$, via the operational semantics defined in Figure 6.3. Multistep reduction is expressed using the reflexive and transitive closure, i.e., $\mathbb{P} :: \widetilde{\mathcal{M}} \to^*_{D;\mathfrak{R}} \mathbb{P}' :: \widetilde{\mathcal{M}}'$.

Since MAGπ! is restricted to a single session, a reliability context as defined in the previous chapter is not needed. Instead, process reduction is parametric on a single reliability configuration, mapping roles to sets of reliable roles, for the single session under consideration.

Once again, communication is *asynchronous*. Rule R-Snd places a message into the buffer whilst advancing the send process to its continuation. Rule R-Rcv advances a receive process to the corresponding continuation if there exists a message in the buffer matching the sender and message label defined in the process. Doing so removes the message from the buffer. This rule also applies to processes that optionally define a failure-handling timeout branch.

Replication. Rules R-$!_1$ and R-$!_2$ describe communication with a *replicated* process R. As expected, rule R-$!_1$ is similar to R-Rcv but the replicated process remains unchanged and the continuation P_k is evaluated in parallel. Rule R-$!_2$ handles the case where the replicated process does not receive from a specific role, but instead allows communication with an *arbitrary* role: the rule binds the sending role to $\boldsymbol{\alpha}$ in the replicated continuation. This is referred to as *universal receive*.

Failure semantics. The process semantics model *total message reordering*, *message loss*, and *message delay*. Message reordering is a result of using buffers defined as multisets (bags), thus being implicitly unordered out-of-the-box. Message loss is modelled directly from rule R-Drop, which can remove a message from the buffer at any point, provided that the message is not part of an assumed reliable link.

Dropped messages are intended to be handled via *timeouts*. To this end, rule R-$\oplus$ reduces a receiving process to its timeout branch continuation. Notably, there are no prerequisites for a timeout to occur. Since timeouts are nondeterministic (can happen at any point regardless of what is in the buffer), it is possible for a process to timeout even though a valid consumable message is in the buffer. This models message delay.

The remaining rules are defined similar to the core calculus. Rule R-+ nondeterministically reduces a choice to one of its paths; rule R-X looks up a process name in the definitions, reducing to the found process substituted with any passed arguments; rule R-| allows processes to reduce under parallel composition and rule R-$\equiv$ allows for reduction up-to congruence.

Definition 6.4 (Context reduction). The type LTS is updated to allow for asynchrony, timeouts, and role substitution. The actions are defined as follows:

$$\text{Actions} \quad A ::= s{:}\boldsymbol{q}{\oplus}\boldsymbol{p}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{q}{\oplus}\boldsymbol{p}{:}\mathsf{m} \mid s{:}\boldsymbol{q}\&\boldsymbol{\rho}{:}\mathsf{m}(\vec{T}) \mid s{:}\boldsymbol{q}\&\boldsymbol{p}{:}\mathsf{m} \mid s{:}\boldsymbol{q}{:}\oplus$$

From left to right, these read as: *output*, *enqueue*, *input*, *dequeue*, and *timeout*. The type LTS relies is defined by the transitions listed in Figure 6.4. Context *reduction*, $\Gamma \to \Gamma'$, is defined iff $\Gamma \xrightarrow{A} \Gamma'$, where $A \in \{s{:}\boldsymbol{p}{\oplus}\boldsymbol{q}{:}\mathsf{m}, s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m}, s{:}\boldsymbol{p}{:}\oplus\}$. The transitive and reflexive closure of context reduction is written as $\to^*$; and $\Gamma \to$ represents the existence of some transition from Γ.

The type semantics are explained by example in Example 6.5.

Example 6.5 (Fault-Tolerant Load Balancer: Type Reduction). Recall the load balancer from Example 6.1. The fault-tolerant nature of replication can be seen from the LTS

Context Reduction $\boxed{\Gamma \xrightarrow{A} \Gamma'}$

$$\Gamma\text{-}\& \quad \frac{k \in I \qquad S = \&_{i\in I} \boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S'_i\,[,\oplus.S'']}{s[\boldsymbol{q}] : S \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q_k}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S'_k} \qquad \Gamma\text{-}! \quad \frac{k \in I \qquad R = !\boldsymbol{\alpha}\&_{i\in I}\mathsf{m}_i(\vec{T}_i).S_i}{s[\boldsymbol{p}] : R \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{\alpha}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : R \,|\, S_k} \qquad \Gamma\text{-}\oplus \quad \frac{k \in I \qquad S = \oplus_{i\in I} \boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S'_i}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q_k}:\mathsf{m}_k(\vec{T}_k)} s[\boldsymbol{p}] : S'_k}$$

$$\Gamma\text{-DEQ} \quad \frac{\Gamma \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{\rho}:\mathsf{m}(\vec{\alpha},\vec{T})} \Gamma' \qquad \vec{T}' \sqsubseteq \vec{T}}{\Gamma, s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{r},\vec{T}')), \widetilde{M} \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}} \Gamma'\{\vec{r}/\vec{\alpha}\}\{p/\rho\} + s : \widetilde{M}} \qquad \Gamma\text{-ENQ} \quad \frac{\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \Gamma'}{\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}} \Gamma' + s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T}))}$$

$$\Gamma\text{-}\oplus \quad \frac{S = \&_{i\in I} \boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S'_i, \oplus.S''}{s[\boldsymbol{p}] : S \xrightarrow{s:\boldsymbol{p}:\oplus} s[\boldsymbol{p}] : S''} \qquad \Gamma\text{-CONG}_1 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma, \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''} \qquad \Gamma\text{-CONG}_2 \quad \frac{\Gamma \xrightarrow{A} \Gamma'}{\Gamma \cdot \Gamma'' \xrightarrow{A} \Gamma' + \Gamma''} \qquad \Gamma\text{-}\mu \quad \frac{\Gamma \cdot c : S\{\mu t.S/t\} \xrightarrow{A} \Gamma'}{\Gamma \cdot c : \mu t.S \xrightarrow{A} \Gamma'}$$

Figure 6.4: Type semantics.

produced by this protocol.

$$s[\boldsymbol{c}] : S_{\boldsymbol{c}},\ s[\boldsymbol{srv}] : S_{\boldsymbol{srv}},\ s[\boldsymbol{w_1}] : S_{\boldsymbol{w}},\ s[\boldsymbol{w_2}] : S_{\boldsymbol{w}},\ s[\boldsymbol{o}] : S_{\boldsymbol{o}}$$
$$\rightarrow$$
$$s[\boldsymbol{c}] : \&\left\{\begin{array}{l} \boldsymbol{w_1} : \mathsf{ans}(\texttt{String}).\oplus \boldsymbol{o} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \boldsymbol{w_2} : \mathsf{ans}(\texttt{String}).\oplus \boldsymbol{o} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \oplus.\mu \mathrm{X}.\oplus \left\{\begin{array}{l} \boldsymbol{srv} : \mathsf{req}(\texttt{Int}). \\ \quad \&\left\{\begin{array}{l} \boldsymbol{w_1} : \mathsf{ans}(\texttt{String}).\oplus \boldsymbol{o} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \boldsymbol{w_2} : \mathsf{ans}(\texttt{String}).\oplus \boldsymbol{o} : \mathsf{ans}(\texttt{String}).\mathbf{end} \\ \oplus.\mathrm{X} \end{array}\right. \\ \boldsymbol{o} : \mathsf{noop}().\mathbf{end} \end{array}\right. \end{array}\right.$$
$$s[\boldsymbol{srv}] : S_{\boldsymbol{srv}},\ s[\boldsymbol{w_1}] : S_{\boldsymbol{w}},\ s[\boldsymbol{w_2}] : S_{\boldsymbol{w}},\ s[\boldsymbol{o}] : S_{\boldsymbol{o}}$$
$$s : (\boldsymbol{c} \triangleright \boldsymbol{srv} : \mathsf{req}(\texttt{Int}))$$

The client begins by sending the request, enqueuing the message in the type buffer (by Γ-Enq). In this version, if the client times out it simply restarts the protocol. Additionally, the servers do not need to define timeout branches as they will remain infinitely available to handle client requests. Consider the client triggers its timeout a few times, resulting in the following context reductions (by continuous applications of Γ-⊕ and Γ-Enq).

$$
\begin{array}{l}
\rightarrow^* \\
s[c] : S_c,\ s[srv] : S_{srv},\ s[w_1] : S_w,\ s[w_2] : S_w,\ s[o] : S_o \\
s : (c \triangleright srv : \mathsf{req}(\mathtt{Int})), (c \triangleright srv : \mathsf{req}(\mathtt{Int})), (c \triangleright srv : \mathsf{req}(\mathtt{Int})) \\
\rightarrow \\
s[srv] : !\alpha\&\mathsf{req}(\mathtt{Int}).\oplus \left\{ \begin{array}{l} w_1 : \mathsf{fw}(\mathtt{Int}, \alpha).\mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}, \alpha).\mathbf{end} \end{array} \right. \mid \oplus \left\{ \begin{array}{l} w_1 : \mathsf{fw}(\mathtt{Int}, c).\mathbf{end} \\ w_2 : \mathsf{fw}(\mathtt{Int}, c).\mathbf{end} \end{array} \right. \\
s[c] : S_c,\ s[w_1] : S_w,\ s[w_2] : S_w,\ s[o] : S_o \\
s : (c \triangleright srv : \mathsf{req}(\mathtt{Int})), (c \triangleright srv : \mathsf{req}(\mathtt{Int}))
\end{array}
$$

Upon receiving the request, the server substitutes the role name of the client (c) into its continuation type (via Γ-Deq), placing the substituted continuation in parallel to the replicated type (by Γ-!). From here, the server may forward the request to the worker, or it may even receive one of the delayed client messages. Observe that it is possible for the LTS to reach the following configuration.

$$
\begin{array}{l}
\rightarrow^* \\
s[c] : S_c,\ s[srv] : S_{srv},\ s[w_1] : S_w,\ s[w_2] : S_w,\ s[o] : S_o \\
s : (w_1 \triangleright c : \mathsf{ans}(\mathtt{String})), (w_2 \triangleright c : \mathsf{ans}(\mathtt{String})), (w_1 \triangleright c : \mathsf{ans}(\mathtt{String}))
\end{array}
$$

Note that this protocol is not bounded. In fact, the client could infinitely timeout and send another request, increasing the size of the buffer. However, the example clearly demonstrates the blended semantics of universal receives and role substitution with non-deterministic timeouts and bag buffers.

6.3 Metatheory

The standard results are once again proven for this final language of the thesis. The presentation is simplified by omitting auxiliary definitions and lemmas that are similar to those used in the previous chapters. More details can be found in appendix B.

6.3.1 Subject Reduction and Session Fidelity

The largest safety property on which subject reduction is dependent upon is given in Definition 6.6. The property is similar to that used in MAGπ, extended to support first-class roles. In particular, the main safety condition (φ-S) now allows for universal receives, and the role condition (φ-$\boldsymbol{\rho}$) requires the type to be void of free role variables. Importantly, the results presented also assume the separation of replicated and non-replicated message labels, as formalised

in Definition 4.10. Following the safety property, *subject reduction* is presented in Theorem 6.7.

Definition 6.6 (Safety). φ is a $!\mathcal{R}$-*safety* property on type environment Γ iff:

φ-**S** $\varphi(\Gamma)$ implies if $\Gamma \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{\rho}:\mathsf{m}(\vec{T})} \wedge (\boldsymbol{r} \triangleright \boldsymbol{q} : \mathsf{m}(\vec{T}')) \in \Gamma(s) \wedge \boldsymbol{\rho} \in \{\boldsymbol{r}, \boldsymbol{\alpha}\}$ then $\mathsf{all}_a\left(\Gamma \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{r}:\mathsf{m}}\right)$;

φ-$\mathcal{R}$ $\varphi(\Gamma, s[\boldsymbol{p}] : \&_{i\in I}\boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i)$ and $\mathcal{R}(s) = \mathfrak{R}$ implies $\forall k \in I : \boldsymbol{q_k} \in \mathfrak{R}(\boldsymbol{p})$;

φ-$\boldsymbol{\rho}$ $\varphi(\Gamma)$ implies $\mathsf{frv}(\Gamma) = \emptyset$;

φ-μ $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu \mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu \mathrm{t}.S/\mathrm{t}\})$;

φ-$\rightarrow$ $\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$.

$\mathcal{R}$-$\mathsf{safe}_!(\Gamma)$ is written iff $\varphi(\Gamma)$ and φ is a $!\mathcal{R}$-*safety* property.

Theorem 6.7 (Subject Reduction). If $P :: \mathcal{B} \rightarrow_{D;\mathcal{R}} P' :: \mathcal{B}'$ and $\Gamma \vdash_D P :: \mathcal{B}$ with $\mathcal{R}$-$\mathsf{safe}_!(\Gamma)$, then $\exists \Gamma'$ s.t. $\Gamma \rightarrow^* \Gamma'$ and $\Gamma' \vdash_D P' :: \mathcal{B}'$ with $\mathcal{R}$-$\mathsf{safe}_!(\Gamma')$.

Proof. (Sketch.) By induction on the derivation of $P :: \mathcal{B} \rightarrow_{D;\mathcal{R}} P' :: \mathcal{B}'$. For the communication and timeout actions, the type can be shown to mimic the action performed by the process. For R-Drop, it is shown that the same context can type a process buffer with or without a particular message (via weakening). The other cases can either be immediately typed using the same context (R-+, R-X), or follow from the inductive hypothesis (R-|, R-$\equiv$). A more detailed proof is given in Theorem B.3. □

The fidelity property Definition 6.8 is also similar to the version used in MAGπ, updated to cater for first-class roles. Different from the previous chapters however is the session fidelity theorem. In fact, the standard formalism for session fidelity does not hold for MAGπ!, due to the differences in how message loss is handled between processes and types. The problem is demonstrated in Example 6.9.

Definition 6.8 (Fidelity). φ is a $!\mathcal{R}$-*fidelity* property on type environment Γ iff:

(i) $\varphi(\Gamma)$ implies if $\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge \Gamma \xrightarrow{\vec{A}} \Gamma' \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}}$ then $\left(\Gamma \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{r}:\mathsf{m}'(\vec{T}')} \implies \forall \vec{A'} : \Gamma \xrightarrow{\vec{A'}} \Gamma'' \xrightarrow{s:\boldsymbol{r}\&\boldsymbol{p}:\mathsf{m}'}\right)$;

(ii) $\varphi(\Gamma, s[\boldsymbol{p}] : \&_{i\in I}\boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i)$ and $\mathcal{R}(s) = \mathfrak{R}$ implies $\forall k \in I : \boldsymbol{q_k} \in \mathfrak{R}(\boldsymbol{p})$;

(iii) $\varphi(\Gamma)$ implies $\mathsf{frv}(\Gamma) = \emptyset$;

(iv) $\varphi(\Gamma \cdot s[\boldsymbol{p}] : \mu \mathrm{t}.S)$ implies $\varphi(\Gamma \cdot s[\boldsymbol{p}] : S\{\mu \mathrm{t}.S/\mathrm{t}\})$;

(v) $\varphi(\Gamma)$ and $\Gamma \rightarrow \Gamma'$ implies $\varphi(\Gamma')$.

$\mathcal{R}$-$\mathsf{fid}_!(\Gamma)$ is written iff $\varphi(\Gamma)$ and φ is a $!\mathcal{R}$-*fidelity* property.

Example 6.9 (Reducing types, stuck process). The following protocol types a process which may reach a state in which the process is terminated but the type context can still reduce.

$$p : !\&q{:}\mathsf{m}.\oplus q{:}\mathsf{m}'.\mathbf{end} \qquad q : \oplus p{:}\mathsf{m}.\&\{p{:}\mathsf{m}.\mathbf{end}, \textcircled{\scriptsize t}.\mathbf{end}\}$$

This occurs when the initial request to the server is dropped from the process buffer at runtime, since message loss is not directly modelled in the type semantics.

Despite this, it is still possible to establish a meaningful session fidelity result. Note that the only scenario where a process cannot mimic any type reduction is if the process terminates and leftover messages remain in the type buffer that were dropped in the process buffer. It should be the case that, whenever this occurs, the only leftover messages are requests to a replicated server. This is because all non-replicated messages that are failure-prone must be handled by a timeout (as per φ-$\mathcal{R}$), and so the process would always be able to at least mimic the timeout reduction from the type.

Replicated messages however are not handled by timeouts, since replicated processes are infinitely available. This means that, since types do not drop messages, requests to servers that get dropped will lead to processes that are eventually stuck whilst the types can reduce. This does not deter from the fact that the processes are following their prescribed protocol, as the only times processes cannot mimic types is when all clients have successfully terminated and the infinitely available servers handle requests which are delayed past the clients' lifetimes. (A very possible scenario which the types must simulate to determine faithful properties on the protocol.) For this reason, session fidelity is updated with an extra clause. The new theorem states that either processes can always mimic at least one path of the LTS, or processes are stuck and the only available context reductions are requests to replicated servers.

Theorem 6.10 (Session Fidelity). Assume $\Gamma \vdash_D P :: \mathcal{B}$ with $\mathcal{R}$-$\mathsf{fid}_!(\Gamma)$ and $\mathsf{one\text{-}role}_!(P)$. Then, $\Gamma\!\rightarrow$ implies either:

1. if $P :: \mathcal{B} \rightarrow_{D;\mathcal{R}}$, then $\exists \Gamma', P', \mathcal{B}'$ s.t. *(i)* $\Gamma\!\rightarrow\Gamma'$; *(ii)* $P :: \mathcal{B} \rightarrow^*_{D;\mathcal{R}} P' :: \mathcal{B}'$; *(iii)* $\Gamma' \vdash_D P' :: \mathcal{B}'$; and *(iv)* $\mathsf{one\text{-}role}_!(P')$; or
2. if $P :: \mathcal{B} \not\rightarrow_{D;\mathcal{R}}$, then $\forall \Gamma'$ s.t. $\Gamma\!\rightarrow\Gamma' : \Gamma \xrightarrow{A} \Gamma' \wedge A = s{:}p\&q{:}\mathsf{m} \wedge \mathsf{m} \in \mathsf{R}$.

Proof. (Sketch.) By induction on the derivation of $\Gamma\!\rightarrow$. There are three possible cases, where Γ is enabled with a reduction by either Γ-Enq, Γ-Deq, or Γ-$\textcircled{\scriptsize t}$. For Γ-Enq, the process is shown to be able to mimic at least one path in the selection (via compatibility). For Γ-$\textcircled{\scriptsize t}$, the process is shown to also define a timeout branch, and therefore can always at least match the timeout action. The case for Γ-Deq is more involved. There are three subcases to consider based on the shape of the branch type.

First, if the branch type does not define a timeout branch, then by φ-$\mathcal{R}$ all possible messages are reliable. Hence, by rule R-Drop, any messages in the type buffer must also be in the process buffer (as they could not have been dropped). Therefore, the process can follow at least one of the dequeue actions.

Second, if the branch does define a timeout branch. Then, by compatibility, the process also defines a timeout branch. Therefore, the process can always at least mimic the timeout action.

Lastly, if the branch is replicated. Then, the dequeue action involves a message that is either in the process buffer or is not. If the message exists in the process buffer, then the process can mimic the dequeue action. If the message is not in the process buffer, then the process can either reduce by mimicking a different action (as shown in the previous cases); or the process is stuck. If the process is stuck and is well-typed under Γ, then the only possible shape of the context is if all messages in the type buffer are towards replicated branches. Therefore, all reduction actions are via messages that live in R.

A detailed version of the proof is given in Theorem B.6. □

6.4 Examples

The protocols written in MPST! in Section 4.4 are valid syntax in MAGπ!, and thus all of those examples can also be expressed in this asynchronous and failure-prone calculus. This section, however, aims to demonstrate the usability of MAGπ! for formalising low-level fault-tolerant network protocols. Thus, the following provides examples inspired by real-world protocols.

6.4.1 Ping Utility

The Ping utility is a network administration tool built on top of the Internet Control Message Protocol [94], and used to test the reachability of hosts on a network. Here, a *host* refers to another computer device connected to the network.

The communication pattern of the utility is straightforward. A host sends a ping message to its destination host, who replies with a pong. If the response is successfully received, the round-trip time is output; if the source triggers a timeout before the reply is received, an error is reported.

Every host in a network should implement Ping, thus, all hosts have a replicated process waiting for the ping message. All hosts should also provide an interface for requesting a ping to another host, i.e., all hosts can act as both the source and the destination. Therefore, the following types describe the behaviour of processes that are implemented on every device in a network.

$$S_p = !\alpha\&\text{ping}.\oplus\alpha\text{:pong}.\mathbf{end}$$

$$S_r = !\beta\&\text{ping-req}(\delta).\oplus\delta\text{:ping}.\&\left\{\begin{array}{l}\delta\text{:pong}.\oplus\beta\text{:ok}(\texttt{Time}).\mathbf{end},\\ \text{🕐}.\oplus\beta\text{:error}(\texttt{String}).\mathbf{end}\end{array}\right.$$

The protocol description focuses on the behaviour of the server. It is the responsibility of the client to ensure that it correctly uses the program interface. With the power of session types, users no longer need to fear implementing a client incorrectly! The type system can help verify whether a given program adheres to the intended use of the protocol. Consider the following program as an example, where three hosts in a network all check the reachability of each other host in a single try, and only initiate their main protocol if all hosts are reached. (Where reliability is omitted, assume unreliable links.)

$$\mathfrak{R} = h_1^c \mapsto \{h_1^p, h_1^r\},\ h_2^c \mapsto \{h_2^p, h_2^r\},\ h_3^c \mapsto \{h_3^p, h_3^r\}$$

$$\Psi = \{$$

$$h_1^p : S_p,\ h_1^r : S_r,\ h_1^c : \oplus h_1^r\text{:ping-req}(h_2^p)\ .\ \&\left\{\begin{array}{l} h_1^r\text{:ok}(\texttt{Time})\ .\ \oplus h_1^r\text{:ping-req}(h_3^p)\ .\\ \qquad\&\left\{\begin{array}{l} h_1^r\text{:ok}(\texttt{Time})\ .\ S^m_{h_1}\\ h_1^r\text{:error}(\texttt{String})\ .\ \mathbf{end}\end{array}\right.\\ h_1^r\text{:error}(\texttt{String})\ .\ \mathbf{end}\end{array}\right.$$

$$h_2^p : S_p,\ h_2^r : S_r,\ h_2^c : \oplus h_2^r\text{:ping-req}(h_1^p)\ .\ \&\left\{\begin{array}{l} h_2^r\text{:ok}(\texttt{Time})\ .\ \oplus h_2^r\text{:ping-req}(h_2^p)\ .\\ \qquad\&\left\{\begin{array}{l} h_2^r\text{:ok}(\texttt{Time})\ .\ S^m_{h_2}\\ h_2^r\text{:error}(\texttt{String})\ .\ \mathbf{end}\end{array}\right.\\ h_1^r\text{:error}(\texttt{String})\ .\ \mathbf{end}\end{array}\right.$$

$$h_3^p : S_p,\ h_3^r : S_r,\ h_3^c : \oplus h_3^r\text{:ping-req}(h_1^p)\ .\ \&\left\{\begin{array}{l} h_3^r\text{:ok}(\texttt{Time})\ .\ \oplus h_3^r\text{:ping-req}(h_2^p)\ .\\ \qquad\&\left\{\begin{array}{l} h_3^r\text{:ok}(\texttt{Time})\ .\ S^m_{h_3}\\ h_3^r\text{:error}(\texttt{String})\ .\ \mathbf{end}\end{array}\right.\\ h_1^r\text{:error}(\texttt{String})\ .\ \mathbf{end}\end{array}\right.$$

$$\}$$

The above is a simple example showing how the Ping utility can be used to test the stability of a network before initiating another protocol. The example begins to hint at the benefit of using MAGπ! types for protocol formalisation, as it clarifies how clients should interact with servers.

6.4.2 Learning Switch

A switch is a device in a computer network tasked with keeping track of the hosts connected to that network. It does this by storing a table of addresses for each host connected to it. Typically, the address table is either manually written (and therefore static), or is dynamically updated as hosts on the network make themselves known to the switch. The following describes a program for a learning switch with a dynamic address table.

There are three server-side processes in this example: *(i)* a monitor m, waiting for client requests; *(ii)* a table t, used to keep track of the address table; and *(iii)* a switch interface s, used to start and stop the switch. Following from the insight gathered on replication and recursion in Section 4.4, it makes sense that roles m and s are replicated (since they handle client requests), whilst role t should be recursive (since it handles data that should be accessed sequentially). The following types describe the server-side protocol for the switch.

$$S_s = !\theta\& \left\{ \begin{array}{l} \mathsf{start}(\texttt{Set (Role)}).\oplus t:\mathsf{set}(\texttt{Set (Role)}). \\ \qquad \& \left\{ \begin{array}{l} t:\mathsf{ok}.\oplus\theta:\mathsf{ok}.\mathbf{end} \\ t:\mathsf{already\_started}.\oplus\theta:\mathsf{error}(\texttt{String}).\mathbf{end} \\ t:\mathsf{already\_stopped}.\oplus\theta:\mathsf{error}(\texttt{String}).\mathbf{end} \end{array} \right. \\ \mathsf{stop}.\oplus t:\mathsf{stop}.\oplus\theta:\mathsf{ok}.\mathbf{end} \end{array} \right.$$

$$S_t^{init} = \&s:\mathsf{set}(\texttt{Set (Role)}).\oplus m:\mathsf{start}.\oplus s:\mathsf{ok}.S_t$$

$$S_t = \mu \mathrm{t}.\& \left\{ \begin{array}{l} s:\mathsf{set}(\texttt{Set (Role)}).\oplus s:\mathsf{already\_started}.\mathrm{t} \\ m:\mathsf{get}.\oplus m:\mathsf{table}(\texttt{Set (Role)}).\& \left\{ \begin{array}{l} m:\mathsf{done}.\mathrm{t} \\ m:\mathsf{add}(\beta).\mathrm{t} \\ m:\mathsf{remove}(\gamma).\mathrm{t} \end{array} \right. \\ s:\mathsf{stop}.!\&\{m:\mathsf{get}.\oplus m:\mathsf{fin}.\mathbf{end},\ s:\mathsf{set}(\texttt{Set (Role)}).\oplus s:\mathsf{already\_stopped}.\mathbf{end}\} \end{array} \right.$$

$$S_m = \&t:\mathsf{start}.!\alpha\& \left\{ \begin{array}{l} \mathsf{connect}.\oplus t:\mathsf{get}.\& \left\{ \begin{array}{l} t:\mathsf{fin}.\mathbf{end} \\ t:\mathsf{table}(\texttt{Set (Role)}).\oplus t:\mathsf{add}(\alpha).\oplus\alpha:\mathsf{ok}.\mathbf{end} \end{array} \right. \\ \mathsf{disconnect}.\oplus t:\mathsf{get}.\& \left\{ \begin{array}{l} t:\mathsf{fin}.\mathbf{end} \\ t:\mathsf{table}(\texttt{Set (Role)}). \\ \qquad \oplus t:\mathsf{remove}(\alpha).\oplus\alpha:\mathsf{ok}.\mathbf{end} \end{array} \right. \\ \mathsf{is\_connected?}(\delta). \\ \qquad \oplus t:\mathsf{get}.\& \left\{ \begin{array}{l} t:\mathsf{fin}.\mathbf{end} \\ t:\mathsf{table}(\texttt{Set (Role)}).\oplus \left\{ \begin{array}{l} \alpha:\mathsf{yes}.\oplus t:\mathsf{done}.\mathbf{end} \\ \alpha:\mathsf{no}.\oplus t:\mathsf{done}.\mathbf{end} \end{array} \right. \end{array} \right. \end{array} \right.$$

The switch begins idle. The monitor is waiting for a start message from the table, which is waiting for a set message from the interface. The interface offers two services, intended to be used by a network administrator. These services either start the switch, initialising the table with pre-established hosts, or stop the switch, making it unresponsive to any further requests.

A start request to the switch interface begins the initialisation phase. The switch sends the initial set of addresses (roles) to the table process and waits for a response. An ok message symbolises that initialisation was successful; otherwise error handling messages are received, indicating that the switch has already been previous started or stopped—the appropriate error message is then forwarded to the network administrator. To better understand the behaviour of the monitor and table, sample process definitions are provided below.

$$
\begin{array}{l}
D_{ls} = \{ \\
\quad \mathrm{Mon}(x : S_m) \mapsto \\
\quad\quad x\&[t]\,\mathsf{start} \,.\, !x[\alpha]\&\{ \\
\quad\quad\quad \mathsf{connect} \,.\, x\oplus[t]\,\mathsf{get} \,.\, x\&\{\,[t]\,\mathsf{fin} \,.\, \mathbf{0},\ \ [t]\,\mathsf{table}(A) \,.\, x\oplus[t]\mathsf{add}(\alpha) \,.\, x\oplus[\alpha]\mathsf{ok} \,.\, \mathbf{0}\,\} \\
\quad\quad\quad \mathsf{disconnect} \,.\, x\oplus[t]\,\mathsf{get} \,.\, x\&\{[t]\,\mathsf{fin} \,.\, \mathbf{0},\ [t]\,\mathsf{table}(A) \,.\, x\oplus[t]\mathsf{remove}(\alpha) \,.\, x\oplus[\alpha]\mathsf{ok} \,.\, \mathbf{0}\} \\
\quad\quad\quad \mathsf{is\_connected?}(\delta) \,.\, x\oplus[t]\,\mathsf{get} \,.\, x\&\{\,[t]\,\mathsf{fin} \,.\, \mathbf{0},\ \ [t]\,\mathsf{table}(A) \,.\, \mathsf{if}\ \delta \in A \\
\quad\quad\quad\quad \mathsf{then}\ x \oplus [\alpha]\,\mathsf{yes} \,.\, x \oplus [t]\,\mathsf{done} \,.\, \mathbf{0} \\
\quad\quad\quad\quad \mathsf{else}\ x \oplus [\alpha]\,\mathsf{no} \,.\, x \oplus [t]\,\mathsf{done} \,.\, \mathbf{0} \\
\quad\quad\quad \} \\
\quad\quad \} \\
\\
\quad \mathrm{TableInit}(x : S_t^{init}) \mapsto \\
\quad\quad x\&[s]\,\mathsf{set}(A) \,.\, x\oplus[m]\,\mathsf{start} \,.\, x\oplus[s]\,\mathsf{ok} \,.\, \mathrm{Table}\langle x,\, A\rangle \\
\quad \mathrm{Table}(x : S_t,\ A : \texttt{Set (Role)}) \mapsto \\
\quad\quad x\&\{ \\
\quad\quad\quad [s]\,\mathsf{set}(\_A') \,.\, x\oplus[s]\,\mathsf{already\_started} \,.\, \mathrm{Table}\langle x,\, A\rangle \\
\quad\quad\quad [m]\,\mathsf{get} \,.\, x\oplus[m]\,\mathsf{table}\langle A\rangle \,.\, x\& \left\{ \begin{array}{l} [m]\,\mathsf{done} \,.\, \mathrm{Table}\langle x,\, A\rangle \\ [m]\,\mathsf{add}(\beta) \,.\, \mathrm{Table}\langle x,\, A\cup\{\beta\}\rangle \\ [m]\,\mathsf{remove}(\gamma) \,.\, \mathrm{Table}\langle x,\, A\setminus\{\gamma\}\rangle \end{array} \right. \\
\quad\quad\quad [s]\,\mathsf{stop} \,.\, !x\&\{\,[m]\,\mathsf{get} \,.\, x \oplus [m]\,\mathsf{fin} \,.\, \mathbf{0},\ \ [s]\,\mathsf{set}(\_A') \,.\, x\oplus[s]\,\mathsf{already\_stopped} \,.\, \mathbf{0}\,\} \\
\quad\quad \} \\
\\
\quad \mathrm{Switch}(x : S_s) \mapsto \\
\quad\quad !x[\theta]\&\{ \\
\quad\quad\quad \mathsf{start}(A) \,.\, x\oplus[t]\,\mathsf{set}\langle A\rangle \,.\, x\&\{ \\
\quad\quad\quad\quad [t]\,\mathsf{ok} \,.\, x\oplus[\theta]\,\mathsf{ok} \,.\, \mathbf{0} \\
\quad\quad\quad\quad [t]\,\mathsf{already\_started} \,.\, x\oplus[\theta]\,\mathsf{error}\langle\text{“Already started.”}\rangle \,.\, \mathbf{0} \\
\quad\quad\quad\quad [t]\,\mathsf{already\_stopped} \,.\, x\oplus[\theta]\,\mathsf{error}\langle\text{“Already stopped.”}\rangle \,.\, \mathbf{0} \\
\quad\quad\quad \} \\
\quad\quad\quad \mathsf{stop} \,.\, x\oplus[t]\,\mathsf{stop} \,.\, x\oplus[\theta]\,\mathsf{ok} \,.\, \mathbf{0} \\
\quad\quad \} \\
\}
\end{array}
$$

The table begins by waiting for a set message with the initial statically-set addresses. Then, upon receipt, starts the monitor and responds with ok to the interface. The set-up for the switch is now complete.

The monitor now waits for clients to connect, disconnect, or query the status of another role. To avoid race conditions in the server, the table is a recursive process, handling read and writes to the table sequentially. When the table is acquired by a monitor process, the recursive nature of the table acts as a lock. The lock is only released upon receiving an action to either add to the table, remove from the table, or leave it unchanged.

Consider the following processes with one administrator (a), two hosts (h_1, h_2), and the switch. The example assumes an inbuilt sleep functionality and specific timeout values for processes, both written in milliseconds.

$$\mathfrak{R} = s \mapsto \{t, a\},\ t \mapsto \{s\},\ m \mapsto \{t\},\ a \mapsto \{s\}$$

$$\Psi = s : S_s,\ t : S_t^{init},\ m : S_m,$$

$$a : \oplus s\text{:start}(\texttt{Set (Role)}).\oplus s\text{:stop.}\mathbf{end}$$

$$h_1 : \oplus m\text{:connect.}\& \left\{ \begin{array}{l} m\text{:ok.}!\omega\&\text{print}(\texttt{Binary}).\mathbf{end} \\ \textcircled{\scriptsize t}.\oplus m\text{:connect.}\& \left\{ \begin{array}{l} m\text{:ok.}!\omega\&\text{print}(\texttt{Binary}).\mathbf{end} \\ \textcircled{\scriptsize t}.\mathbf{end} \end{array} \right. \end{array} \right.$$

$$h_2 : \oplus m\text{:is\_connected?}(h_1).\& \left\{ \begin{array}{l} m\text{:yes.}\oplus h_1\text{:print}(\texttt{Binary}).\mathbf{end} \\ m\text{:no.}\mathbf{end} \\ \textcircled{\scriptsize t}.\mathbf{end} \end{array} \right.$$

$$\begin{array}{l} P = (\nu s^{\mathfrak{R}} : \Psi) \\ \quad \text{Switch}\langle s[s]\rangle \mid \text{TableInit}\langle s[t]\rangle \mid \text{Mon}\langle s[m]\rangle \\ \quad \mid s[a] \oplus [s]\,\text{start}\langle\emptyset\rangle\ .\ \text{sleep}(1000)\ .\ s[a] \oplus [s]\,\text{stop}\ .\ \mathbf{0} \\ \quad \mid s[h_1] \oplus [m]\,\text{connect}\ .\ s[h_1]\& \left\{ \begin{array}{l} [m]\,\text{ok}\ .\ !s[h_1][\omega]\&\text{print}(job)\ .\ \text{queue\_print}(job)\ .\ \mathbf{0} \\ \textcircled{\scriptsize t}(100)\,.\,s[h_1] \oplus [m]\,\text{connect}\ . \\ \qquad s[h_1]\& \left\{ \begin{array}{l} [m]\,\text{ok}\ .\ !s[h_1][\omega]\&\text{print}(job)\ . \\ \qquad \text{queue\_print}(job)\ .\ \mathbf{0} \\ \textcircled{\scriptsize t}(100)\,.\,\mathbf{0} \end{array} \right. \end{array} \right. \\ \quad \mid \text{sleep}(250)\ .\ s[h_2] \oplus [m]\,\text{is\_connected?}\langle h_1\rangle\ . \\ \qquad s[h_2]\& \left\{ \begin{array}{l} [m]\,\text{yes}\ .\ s[h_2] \oplus [h_1]\,\text{print}\langle\text{serialise}(\text{“./thesis.pdf”})\rangle\ .\ \mathbf{0} \\ [m]\,\text{no}\ .\ \mathbf{0} \\ \textcircled{\scriptsize t}(100)\,.\,\mathbf{0} \end{array} \right. \end{array}$$

The program (P, D_{ls}) attempts to connect a printer to the switch. If successful, the printer offers its print service to any role on the network, where it simply queues a job for printing. Since the printer is not sophisticated enough to offer appropriate responses to clients, as well as

to reduce the traffic towards it, clients should query the switch to determine whether the printer is accessible.

This example demonstrates once again how replication and first-class roles lend themselves well to describing client-server systems. In fact, the switch was presented separately to the remainder of the program since replication allows components to be designed modularly. Furthermore, the program exemplifies how replication is useful for simplifying failure handling by leveraging the client-server paradigm—note how the behaviour of the switch does not include a single timeout. Instead, timeouts are used by clients to issue retries to the infinitely available servers. Lastly, the use of nondeterministic timeouts in the type semantics is enough to model the behaviour of this failure-prone program, and can be used to verify communication-centric properties.

6.5 Conclusion

This chapter presented MAGπ!, combining the novel features of *replication* and *first-class roles* with *nondeterministic timeouts* into a single type system. The language demonstrated how replication lends itself well to describing client-server systems in failure-prone networks, and can be used to simplify the design of fault-tolerant protocols. Although typechecking is not decidable, strategies introduced in the previous chapters can be combined to determine whether protocols designed in MAGπ! live within a subset of the language for which typechecking is decidable.

With respect to the third research question of this thesis, MAGπ! was used to formalise a number of examples inspired by real-world network protocols. Replication is particularly useful for describing low-level systems protocols, as processes are often designed to be infinitely available as a means of handling message loss.

Chapter 7

Conclusion

This thesis explored the use of multiparty session types (MPST) within client-server, failure-prone, and fault-tolerant systems. The work demonstrated how replication and first-class roles can be integrated into MPST to better model the client-server paradigm, and nondeterministic timeouts within asynchronous type semantics can model various forms of failure within communicating systems. The two approaches are ultimately combined into a single type system and shown to be expressive enough to verify properties on programs built to provide a degree of fault-tolerance.

Exploring further techniques for restricting protocols to subsets of the established languages for which typechecking is decidable would improve the usability of the studied type systems. Avenues for investigation include studying encodability of the λ-calculus in MPST! types, taking inspiration from Milner's encoding of λ in π using replication [79]. It is conjectured that such an encoding exists, but would be reliant on first-class roles, hinting at the fact that maybe the type system is decidable without first-class roles (but still using replication). This would be a significant result as the expressiveness of types are still increased when using only replication (e.g. binary trees can still be represented). Results of decidability (even for sublanguages) would help to build tools for putting the generalised type systems developed in this thesis into practice. For instance, a model checking tool could be used for verifying process properties of protocols, similar to the tool developed by Scalas and Yoshida [103].

Another route of investigation could be to explore methods of modelling more failure models in MAGπ. Specifically, a barrier for modelling the full range of *non-Byzantine faults* is *message duplication*. This thesis did not consider the possibility of message duplication as it resulted in difficulties for identifying sublanguages for which type checking is decidable. (Observe that boundedness would not be a suitable strategy if message duplication is directly modelled in the type semantics.) Overcoming this hurdle would result in a type system that faithfully models communication over networks susceptible to non-Byzantine faults, opening the gates to using MPST with complex fault-tolerant protocols such as consensus algorithms.

Appendix A

Proofs for MPST!

A.1 Lemmata

A.1.1 Context Operations

Lemma A.1 verifies that context addition in MPST! is the left inverse of context splitting; i.e., the pieces of a split context can be added back together to form the original context.

Lemma A.1. If $\Gamma = \Gamma_1 \cdot \Gamma_2$, then $\Gamma_1 + \Gamma_2 = \Gamma$.

Proof. By induction on the derivation of $\Gamma = \Gamma_1 \cdot \Gamma_2$.

Case 1: $\dfrac{}{\Gamma = \emptyset = \emptyset \cdot \emptyset}$

$\emptyset + \emptyset = \emptyset$	(by Figure 4.5)	(1)
Case 1 holds.	(by (1))	(2)

Case 2: $\dfrac{\Gamma' = \Gamma_1 \cdot \Gamma_2}{\Gamma = \Gamma', c : U = (\Gamma_1, c : U) \cdot \Gamma_2}$

$c \notin \mathsf{dom}(\Gamma')$	(from the hyp. and def. of context composition)	(1)
$c \notin \mathsf{dom}(\Gamma_2)$	(by (1) and the hyp.)	(2)
$\Gamma_1 + \Gamma_2 = \Gamma'$	(the ind. hyp.)	(3)
$(\Gamma_1, c : U) + \Gamma_2 = \Gamma', c : U$	(by (2), (3), and Figure 4.5)	(4)
Case 2 holds.	(by (4))	(5)

Case 3: $\dfrac{\Gamma' = \Gamma_1 \cdot \Gamma_2}{\Gamma = \Gamma', c : U = \Gamma_1 \cdot (\Gamma_2, c : U)}$

$c \notin \mathsf{dom}(\Gamma')$ (from the hyp. and def. of context composition) (1)
$c \notin \mathsf{dom}(\Gamma_1)$ (by (1) and the hyp.) (2)
$\Gamma_1 + \Gamma_2 = \Gamma'$ (the ind. hyp.) (3)
$\Gamma_1 + (\Gamma_2, c : U) = \Gamma', c : U$ (by (2), (3), and Figure 4.5) (4)
Case 3 holds. (by (4)) (5)

Case 4: $\dfrac{\Gamma' = \Gamma_1 \cdot \Gamma_2}{\Gamma = \Gamma', x : B = \Gamma_1, x : B \cdot \Gamma_2, x : B}$

$\Gamma_1 + \Gamma_2 = \Gamma'$ (the ind. hyp.) (1)
$\Gamma_1, x : B + \Gamma_2, x : B = \Gamma', x : B$ (by (1) and Figure 4.5) (2)
Case 4 holds. (by (2)) (3)

Case 5: $\dfrac{\Gamma' = \Gamma_1 \cdot \Gamma_2}{\Gamma = \Gamma', \alpha : \alpha = \Gamma_1, \alpha : \alpha \cdot \Gamma_2, \alpha : \alpha}$

$\Gamma_1 + \Gamma_2 = \Gamma'$ (the ind. hyp.) (1)
$\Gamma_1, \alpha : \alpha + \Gamma_2, \alpha : \alpha = \Gamma', \alpha : \alpha$ (by (1) and Figure 4.5) (2)
Case 5 holds. (by (2)) (3)

Case 6: $\dfrac{\Gamma' = \Gamma_1 \cdot \Gamma_2}{\Gamma = \Gamma', c : U_1 | U_2 = \Gamma_1, c : U_1 \cdot \Gamma_2, c : U_2}$

$\Gamma_1 + \Gamma_2 = \Gamma'$ (the ind. hyp.) (1)
$\Gamma_1, c : U_1 + \Gamma_2, c : U_2 = \Gamma', c : U_1 | U_2$ (by (1) and Figure 4.5) (2)
Case 6 holds. (by (2)) (3)

□

Lemma A.2 states that contexts not containing session types can be added into the type environment with no effect on typing a process.

Lemma A.2. If $\Gamma \vdash_D P$ and $\Gamma' = \{\widetilde{b : T}\}$ such that $\forall b \in \mathsf{dom}(\Gamma') : \Gamma'(b) \not\leqslant S$, then $\Gamma + \Gamma' \vdash_D P$

Proof. Firstly observe that by the hypothesis and Definition 4.7, $\mathsf{end}(\Gamma')$. Furthermore, since Γ' only contains non-linear types, by Figure 4.5, addition of Γ' distributes over context splits. (Since all non-linear types are merged when added.) Now proceed by rule induction on $\Gamma \vdash_D P$.

Case T-0: $\dfrac{\mathsf{end}(\Gamma)}{\Gamma \vdash_D \mathbf{0}}$

$\mathsf{end}(\Gamma)$ (by T-**0**) (1)

$\mathsf{end}(\Gamma + \Gamma')$ (by Definition 4.7, Figure 4.5, and since $\mathsf{end}(\Gamma')$) (2)

$\Gamma + \Gamma' \vdash_D \mathbf{0}$ (by (2), T-**0**) (3)

Case T-**0** holds. (by (3)) (4)

Case T-⊕: $\dfrac{\Gamma_\oplus \vdash c : \oplus \boldsymbol{\rho}{:}\mathsf{m}(\vec{T}).S \qquad \Gamma_r \vdash \boldsymbol{\rho} : \boldsymbol{\rho} \qquad (\Gamma_i \vdash V_i : T_i)_{i\in 1..n} \qquad \Gamma_0 + c : S \vdash_D P'}{\Gamma = \Gamma_0 \cdot \Gamma_\oplus \cdot \Gamma_r (\cdot \Gamma_i)_{i\in 1..n} \vdash_D P = c[\boldsymbol{\rho}] \oplus \mathsf{m}\langle (V_i)_{i\in 1..n}\rangle . P'}$

$\Gamma + \Gamma' = (\Gamma_0 + \Gamma') \cdot (\Gamma_\oplus + \Gamma') \cdot (\Gamma_r + \Gamma')(\cdot \Gamma_i + \Gamma')_{i\in 1..n}$ (by Figure 4.5) (1)

$(\Gamma_0 + \Gamma') + c : S \vdash_D P'$ (by ind. hyp) (2)

$\Gamma_\oplus + \Gamma' \vdash c : \oplus \boldsymbol{\rho}{:}\mathsf{m}(\vec{T}).S$ (by T-Wkn, $\mathsf{end}(\Gamma')$) (3)

$\Gamma_r + \Gamma' \vdash \boldsymbol{\rho} : \boldsymbol{\rho}$ (by T-Wkn, $\mathsf{end}(\Gamma')$) (4)

$\forall i \in 1..n : \Gamma_i + \Gamma' \vdash V_i : T_i)_{i\in 1..n}$ (by T-Wkn, $\mathsf{end}(\Gamma')$) (5)

$\Gamma + \Gamma' \vdash_D P$ (by (2), (3), (4), (5), T-⊕) (6)

Case T-⊕ holds. (by (6)) (7)

Case T-&, T-!: Similar to Case T-⊕.

Case T-|: $\dfrac{\Gamma_1 \vdash_D P_1 \qquad \Gamma_2 \vdash_D P_2}{\Gamma = \Gamma_1 \cdot \Gamma_2 \vdash_D P = P_1 | P_2}$

$\Gamma = (\Gamma_1 + \Gamma') \cdot (\Gamma_2 + \Gamma')$ (by Figure 4.5) (1)

Case T-| holds. (by (1), the ind. hyp. and T-|) (2)

Case T-X: $\dfrac{D(\mathrm{X}) = (\vec{x}, \vec{T}, P) \qquad \forall i \in 1..n : \Gamma_i \vdash V_i : T_i \qquad \mathsf{end}(\Gamma_0)}{\Gamma = \Gamma_0 (\cdot \Gamma_i)_{i\in 1..n} \vdash_D P = \mathrm{X}\langle (V_i)_{i\in 1..n}\rangle}$

$\Gamma + \Gamma' = \Gamma_0 + \Gamma' (\cdot \Gamma_i + \Gamma')_{i\in 1..n}$ (by Figure 4.5) (1)

$\mathsf{end}(\Gamma_0 + \Gamma')$ (since $\mathsf{end}(\Gamma_0)$ and $\mathsf{end}(\Gamma')$) (2)

$\forall i \in 1..n : \Gamma_i + \Gamma' \vdash V_i : T_i$ (by T-Wkn) (3)

Case T-X holds. (by (2), (3), T-X) (4)

Case T-+, T-ν: $\dfrac{(\Gamma \vdash_D P_i^{\oplus})_{i\in I}}{\Gamma \vdash_D \sum_{i\in I} P_i^{\oplus}}$ and $\dfrac{\varphi(\mathtt{assoc}_s(\Psi)) \qquad \varphi \text{ is a } !\text{-}\textit{safety} \text{ property} \qquad s \notin \Gamma \qquad \Gamma + \mathtt{assoc}_s(\Psi) \vdash_D P}{\Gamma \vdash_D (\nu s : \Psi)\, P}$

Both hold directly from the ind. hyp. □

A.1.2 Substitution

Lemma A.3. If $\Gamma_1 + \Gamma_2 = \Gamma$, then $\Gamma_1\{q/\alpha\} + \Gamma_2\{q/\alpha\} = \Gamma\{q/\alpha\}$.

Proof. By induction on the derivation of $\Gamma_1 + \Gamma_2 = \Gamma$.

Case 1: $\dfrac{}{\Gamma + \emptyset = \Gamma}$. Since $\emptyset\{q/\alpha\} = \emptyset$, then $\Gamma\{q/\alpha\} + \emptyset = \Gamma\{q/\alpha\}$.

Case 2 and 3: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, \alpha : \alpha + \Gamma_2, \alpha : \alpha = \Gamma, \alpha : \alpha}$ and $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, x : B + \Gamma_2, x : B = \Gamma, x : B}$

Hold by the ind. hyp. and since $\{\alpha : \alpha\}\{q/\alpha\} = \{\alpha : \alpha\}$ and $\{x : B\}\{q/\alpha\} = \{x : B\}$ (by Figure 4.8).

Case 4: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma \qquad a \notin \mathsf{dom}(\Gamma_2)}{\Gamma_1, a : T + \Gamma_2 = \Gamma, a : T}$

First, observe that by Figure 4.8, substitution does not affect context domains.

$(\Gamma_1, a : T)\{q/\alpha\} = (\Gamma_1\{q/\alpha\}), (a : T\{q/\alpha\})$	(by Figure 4.8)	(1)
$\Gamma_1\{q/\alpha\} + \Gamma_2\{q/\alpha\} = \Gamma\{q/\alpha\}$	(ind. hyp.)	(2)
$a \notin \mathsf{dom}(\Gamma_2\{q/\alpha\})$	(since $a \notin \mathsf{dom}(\Gamma_2)$)	(3)
$(\Gamma_1, a : T)\{q/\alpha\} + \Gamma_2\{q/\alpha\} = (\Gamma\{q/\alpha\}), (a : T\{q/\alpha\})$	(by (1), (2), (3), +)	(4)
Case 4 holds.	(by (4), Figure 4.8)	(5)

Case 5: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma \qquad a \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + \Gamma_2, a : T = \Gamma, a : T}$. Similar to Case 4.

Case 6: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, c : U_1 + \Gamma_2, c : U_2 = \Gamma, c : U_1 | U_2}$. Holds directly from the ind. hyp.

□

Lemma A.4. If $\Gamma \vdash V : T$, then $\Gamma\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$.

Proof. By rule induction on $\Gamma \vdash V : T$.

Case T-Sub: $\dfrac{T \leqslant T'}{\Gamma = c : T \vdash (V = c) : T'}$. RTP: $(c : T)\{q/\alpha\} \vdash c\{q/\alpha\} : T'\{q/\alpha\}$

Note that if c is a variable and T is *not* a session type, then role substitution has no effect and thus the case would hold trivially. Consider now $T = S$ and $T' = S'$.

$S \leqslant S'$ (by assumption) (1)
$S\{q/\alpha\} \leqslant S'\{q/\alpha\}$ (by (1) and lemma A.9) (2)
$(c : S)\{q/\alpha\} \vdash c : S'\{q/\alpha\}$ (by (2) and rule T-Sub) (3)
$c\{q/\alpha\} = c$ (since $c = x$ or $c = s[p]$, for any x, s, p) (4)
Case T-Sub holds. (by (3), (4), and since $S = T$ and $S' = T'$) (5)

Case T-Wkn: $\dfrac{\Gamma_1 + \Gamma_2 = \Gamma \qquad \Gamma_1 \vdash V : T \qquad \mathsf{end}(\Gamma_2)}{\Gamma \vdash V : T}$. RTP: $\Gamma\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$

$\Gamma_1\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$ (ind. hyp.) (1)
$\mathsf{end}(\Gamma_2)$ (by assumption) (2)
$\Gamma_2 = \Gamma_2\{q/\alpha\}$ (by Figure 4.8) (3)
$\Gamma_1\{q/\alpha\} + \Gamma_2 = \Gamma\{q/\alpha\}$ (by hyp., (3), and lemma A.3) (4)
$\Gamma\{q/\alpha\} \vdash V\{q/\alpha\} : T\{q/\alpha\}$ (by (1), (2), (4)) (5)
Case T-Wkn holds. (by (5)) (6)

Case T-α: $\dfrac{}{\alpha : \alpha \vdash \alpha : \alpha}$. RTP: $(\alpha : \alpha)\{q/\alpha\} \vdash (\alpha\{q/\alpha\}) : (\alpha\{q/\alpha\})$

It is key to note that the substitution has no effect on role singletons in the type context, but does indeed substitute the role variables in the value and value type positions. Therefore, it must be shown that: $\alpha : \alpha \vdash q : q$.

$\emptyset + \alpha : \alpha = \alpha : \alpha$ (by Figure 4.5) (1)
$\emptyset \vdash q : q$ (by rule T-q) (2)
$\mathsf{end}(\alpha : \alpha)$ (by Definition 4.7) (3)
$\alpha : \alpha \vdash q : q$ (by (1),(2),(3), and rule T-Wkn) (4)
Case T-α holds. (by (4)) (5)

Case T-B,T-q: These cases hold trivially.

□

Lemma A.5. If $\Gamma \vdash_D P$, then $\Gamma\{q/\alpha\} \vdash_D P\{q/\alpha\}$.

Proof. By rule induction on $\Gamma \vdash_D P$.

Case T-0: $\dfrac{\mathsf{end}(\Gamma)}{\Gamma \vdash_D \mathbf{0}}$. Since $\mathsf{end}(\Gamma)$, then $\Gamma\{q/\alpha\} = \Gamma$ (Figure 4.8). Also $\mathbf{0}\{q/\alpha\} = \mathbf{0}$.

Case T-⊕: $$\frac{\Gamma_\oplus \vdash c : \oplus\boldsymbol{\rho}{:}\mathsf{m}(\vec{T}).S \qquad \Gamma_r \vdash \boldsymbol{\rho} : \boldsymbol{\rho} \qquad (\Gamma_i \vdash V_i : T_i)_{i\in 1..n} \qquad \Gamma_0 + c : S \vdash_D P'}{\Gamma = \Gamma_0 \cdot \Gamma_\oplus \cdot \Gamma_r (\cdot \Gamma_i)_{i\in 1..n} \vdash_D P = c[\boldsymbol{\rho}] \oplus \mathsf{m}\langle (V_i)_{i\in 1..n}\rangle . P'}$$

First observe that by the definition of role substitution on contexts Figure 4.8, substitution distributes over context splits. Then observe that by the ind. hyp.:

$$(\Gamma_0 + c : S)\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash_D P'\{\boldsymbol{q}/\boldsymbol{\alpha}\}$$

Furthermore, by lemma A.4:

$$\begin{gathered}\Gamma_\oplus\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash (c\{\boldsymbol{q}/\boldsymbol{\alpha}\}) : (\oplus\boldsymbol{\rho}{:}\mathsf{m}(\vec{T}).S\{\boldsymbol{q}/\boldsymbol{\alpha}\}) \\ \Gamma_r\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash (\boldsymbol{\rho}\{\boldsymbol{q}/\boldsymbol{\alpha}\}) : (\boldsymbol{\rho}\{\boldsymbol{q}/\boldsymbol{\alpha}\}) \\ \forall i \in 1..n : \Gamma_i\{\boldsymbol{q}/\boldsymbol{\alpha}\} \vdash (V_i\{\boldsymbol{q}/\boldsymbol{\alpha}\}) : (T_i\{\boldsymbol{q}/\boldsymbol{\alpha}\})\end{gathered}$$

Then, using T-⊕, the case holds.

All other cases follow the same steps, relying on the inductive hypothesis and lemma A.4. □

Lemma A.6. $T \leqslant T'$ implies $T\{\boldsymbol{q}/\boldsymbol{\alpha}\} \leqslant T'\{\boldsymbol{q}/\boldsymbol{\alpha}\}$.

Proof. By coinduction on $T \leqslant T'$ (Definition 4.5).

Case 1: $$\frac{}{T \leqslant T}$$

The reflexive case immediately holds since the substitution occurs on the same type on both sides.

Case 2: $$\frac{(\vec{T}_i \leqslant \vec{T}'_i)_{i\in I} \qquad (S_i \leqslant S'_i)_{i\in I}}{\&_{i\in I}\boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T}_i).S_i \leqslant \&_{i\in I\cup J}\boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T}'_i).S'_i}$$

From the coinductive hypothesis, it follows that $\forall i \in I : \vec{T}_i\{\boldsymbol{q}/\boldsymbol{\alpha}\} \leqslant \vec{T}'_i\{\boldsymbol{q}/\boldsymbol{\alpha}\}$ and $S_i\{\boldsymbol{q}/\boldsymbol{\alpha}\} \leqslant S'_i\{\boldsymbol{q}/\boldsymbol{\alpha}\}$. Furthermore, if $\boldsymbol{\rho} = \boldsymbol{\alpha}$, then the substitution occurs on both sides.

Other cases follow the same reasoning and hold directly from the coinductive hypothesis and since any role substitution made on the type subject affects both sides equally. □

Lemma A.7. If $\Gamma + \boldsymbol{\alpha} : \boldsymbol{\alpha} \vdash_D P$ and $\boldsymbol{\alpha} \notin \mathsf{frv}(P)$, then $\Gamma \vdash_D P$

Proof. By rule induction on $\Gamma + \boldsymbol{\alpha} : \boldsymbol{\alpha} \vdash_D P$, observing that the role is never used if it

does not appear free. □

Lemma A.8. If $\Gamma + x : T \vdash_D P$ and $\Gamma' \vdash V : T'$ where $T' \leqslant T$, then $\Gamma + \Gamma' \vdash_D P\{V/x\}$.

Proof. By rule induction on the typing derivation. □

A.1.3 Subtyping

Lemma A.9. $T \leqslant T'$ implies $T\{\boldsymbol{q}/\boldsymbol{\alpha}\} \leqslant T'\{\boldsymbol{q}/\boldsymbol{\alpha}\}$.

Proof. By coinduction on $T \leqslant T'$ (Definition 4.5).

Case 1: $\dfrac{}{T \leqslant T}$

The reflexive case immediately holds since the substitution occurs on the same type on both sides.

Case 2: $\dfrac{(\vec{T_i} \leqslant \vec{T_i'})_{i \in I} \qquad (S_i \leqslant S_i')_{i \in I}}{\&_{i \in I} \boldsymbol{\rho}:\mathsf{m}_i(\vec{T_i}).S_i \leqslant \&_{i \in I \cup J} \boldsymbol{\rho}:\mathsf{m}_i(\vec{T_i'}).S_i'}$

From the coinductive hypothesis, it follows that $\forall i \in I : \vec{T_i}\{\boldsymbol{q}/\boldsymbol{\alpha}\} \leqslant \vec{T_i'}\{\boldsymbol{q}/\boldsymbol{\alpha}\}$ and $S_i\{\boldsymbol{q}/\boldsymbol{\alpha}\} \leqslant S_i'\{\boldsymbol{q}/\boldsymbol{\alpha}\}$. Furthermore, if $\boldsymbol{\rho} = \boldsymbol{\alpha}$, then the substitution occurs on both sides.

Other cases follow the same reasoning and hold directly from the coinductive hypothesis and since any role substitution made on the type subject affects both sides equally. □

Lemma A.10. Assume $\Gamma \vdash V : T$ and $\Gamma' \leqslant \Gamma$. Then, $\Gamma' \vdash V : T$.

Proof. By rule induction on the typing derivation. □

Lemma A.11. Assume $\Gamma \vdash_D P$ and $\Gamma' \leqslant \Gamma$. Then, $\Gamma' \vdash_D P$.

Proof. By rule induction on the typing derivation, and using lemma A.10 for values. □

A.1.4 Congruence

Lemma A.12. Assume $\Gamma \vdash_D P$ and $P \equiv P'$. Then, $\exists \Gamma'$ s.t. $\Gamma \equiv \Gamma'$ and $\Gamma' \vdash P'$.

Proof. By rule induction on $P \equiv P'$, observing that for each rule, either the process remains well typed under the same context, or the context has a matching congruence rule. □

A.2 Subject Reduction

Lemma A.13. For all contexts Γ_1, Γ_2:

1. if $\mathsf{safe}_!(\Gamma_1 \cdot \Gamma_2)$, then $\mathsf{safe}_!(\Gamma_1)$;
2. if $\mathsf{safe}_!(\Gamma_1)$ and $\Gamma_1 \leqslant \Gamma_2$, then $\mathsf{safe}_!(\Gamma_2)$;
3. if $\mathsf{safe}_!(\Gamma_1)$ and $\exists \Gamma_2' : \Gamma_1 \leqslant \Gamma_2 \rightarrow \Gamma_2'$, then $\exists \Gamma_1' : \Gamma_1 \rightarrow \Gamma_1' \leqslant \Gamma_2'$.

Proof. *(1)* First observe that by the definition of frv (Figure 4.8), if $\mathsf{frv}(\Gamma_1 \cdot \Gamma_2) = \emptyset$ then $\mathsf{frv}(\Gamma_1) = \emptyset$. Hence condition φ-$\boldsymbol{\rho}$ is preserved through context splits. The preservation of the remaining conditions is proved by contradiction. Assume Γ_1 is *not* safe, then there must exist some Γ' such that $\Gamma_1 \rightarrow^* \Gamma''$ and $\Gamma' = \mathsf{unf}^*(\Gamma'')$ and Γ' violates φ-S. But by rule Γ-Cong_2, $\Gamma_2 \cdot \Gamma_1 \rightarrow^* \Gamma_2 + \Gamma''$ and after some possible applications of rule Γ-μ, $\Gamma_2 + \Gamma'$ violates φ-S.
Hence, $\Gamma_2 + \Gamma'$ is not safe. By Table 4.1, the proof splits into two similar cases; the first case is demonstrated below.

$$\exists s, \boldsymbol{p}, \boldsymbol{q}, \mathsf{m}, \mathsf{m}', \vec{T}, \vec{T}' : \Gamma' \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge\ \Gamma' \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}'(\vec{T}')} \wedge\ \mathsf{m}, \mathsf{m}' \in \mathsf{L} \wedge \neg \left(\mathsf{all} \left(\Gamma' \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \right) \right).$$

By the definition of $+$ (Figure 4.5), context addition only (at-most) increases the number of transitions and does not *decrease* or *replace* transitions.

$$\therefore\ \Gamma_2 + \Gamma' \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})} \wedge\ \Gamma_2 + \Gamma' \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}'(\vec{T}')} \wedge\ \mathsf{m}, \mathsf{m}' \in \mathsf{L} \wedge \neg \left(\mathsf{all} \left(\Gamma_2 + \Gamma' \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}} \right) \right).$$

This implies that $\Gamma_2 + \Gamma'$ is not safe, meaning $\Gamma_2 + \Gamma''$ is not safe via φ-μ, and thus $\Gamma_2 + \Gamma_1$ is not safe by φ-$\rightarrow$. By lemma A.1 and commutativity of $\cdot$, $\Gamma_1 \cdot \Gamma_2$ is not safe. This contradicts the thesis.
The other case is when the safety violation occurs through a replicated label, i.e., $\mathsf{m}, \mathsf{m}' \in \mathsf{R}$. The case follows similar reasoning to the above. □

(2) By coinduction on $\mathsf{safe}_!(\Gamma_1)$. Observe that by $\mathsf{safe}_!(\Gamma_1)$, Γ_1 is safe (as in Table 4.1).

Thus, if $\Gamma_1 \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}(\vec{T})}$ and $\Gamma_1 \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}'(\vec{T}')}$ then $\Gamma_1 \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}}$. Assume communication to be enabled on Γ_1, and without loss of generality, assume the above output and input actions to also be enabled. By rule Γ-$\oplus$ and rule Γ-&, the shape of Γ_1 can be inferred to be:

$$\Gamma_1 = \Gamma_1' \cdot s[\boldsymbol{p}] : S^{\oplus} \cdot s[\boldsymbol{q}] : S^{\&}$$
$$S^{\oplus} = \oplus_{i\in I} \boldsymbol{r_i}:\mathsf{m}_i(\vec{T}_i).S_i \quad \text{for some } I$$
$$\exists I' \subseteq I \text{ such that } \forall i \in I' : \boldsymbol{r_i} = \boldsymbol{q}$$
$$S^{\&} = \&_{j\in J} \boldsymbol{p}:\mathsf{m}'_j(\vec{T}'_j).S'_j \text{ or } S^{\&} = !\&_{j\in J} \boldsymbol{p}:\mathsf{m}'_j(\vec{T}'_j).S'_j \text{ or } S^{\&} = !\&_{j\in J} \boldsymbol{\alpha}:\mathsf{m}'_j(\vec{T}'_j).S'_j$$

By φ-S there are no unexpected messages in $S^{\oplus}$. Consider now a $S^{\oplus\prime} \geqslant S^{\oplus}$ and $S^{\&\prime} \geqslant S^{\&}$. By the definition of $\leqslant$ (Definition 4.5), $S^{\oplus\prime}$ has (at most) less internal choices, and $S^{\&\prime}$ has (at most) more external choices. Therefore, there are still no unexpected messages in $S^{\oplus\prime}$. Hence, given $\mathsf{safe}_!(\Gamma_1)$ and a $\Gamma_2 \geqslant \Gamma_1$, it follows that φ-S holds for Γ_2. It also holds that $\mathsf{frv}(S^{\oplus\prime}) = \mathsf{frv}(S^{\oplus}) = \emptyset$, by $\mathsf{safe}_!(\Gamma_1)$ and since $S^{\oplus\prime}$ has (at most) less choices than $S^{\oplus}$; and $\mathsf{frv}(S^{\&\prime}) = \mathsf{frv}(S^{\&}) = \emptyset$, by $\mathsf{safe}_!(\Gamma_1)$ and since role variables in any new branches are in binding position (by Figure 4.8). Furthermore, by Rule Γ-μ, no new transitions are introduced via recursive binders, hence φ-μ must also hold for Γ_2. Lastly, by the coinductive hypothesis, $\mathsf{safe}_!$ holds for contexts containing the continuation types of elements in Γ_1 and Γ_2—these contexts are uniquely contained within the context reduction relation; therefore φ-$\rightarrow$ holds for Γ_2. □

(3) By case analysis. Assume $\Gamma_2 \rightarrow$ by rule Γ-Com_1, then:

$$\Gamma_2 = \Gamma_2'' \cdot s[\boldsymbol{p}] : S^{\oplus\prime} \cdot s[\boldsymbol{q}] : S^{\&\prime}$$
$$S^{\oplus\prime} = \oplus_{i\in I'} \boldsymbol{r_i}:\mathsf{m}''_i(\vec{T}''_i).S''_i$$
$$S^{\&\prime} = \&_{j\in J'} \boldsymbol{p}:\mathsf{m}'''_j(\vec{T}'''_j).S'''_j \text{ or } S^{\&\prime} = !\&_{j\in J'} \boldsymbol{p}:\mathsf{m}'''_j(\vec{T}'''_j).S'''_j$$
$$\exists K' \subseteq I' \text{ such that } \forall i \in K' : \boldsymbol{r_i} = \boldsymbol{q}$$
$$K' \subseteq J' \text{ and } \forall i \in K' : \vec{T}''_i \leqslant \vec{T}'''_i$$

Via subtyping, the shape of Γ_1 can be inferred (since $\Gamma_1 \leqslant \Gamma_2$).

$$\Gamma_1 = \Gamma_1'' \cdot s[\boldsymbol{p}] : S^{\oplus} \cdot s[\boldsymbol{q}] : S^{\&}$$
$$S^{\oplus} = \oplus_{i \in I} \boldsymbol{r_i}{:}\mathsf{m}_i(\vec{T}_i).S_i$$
$$S^{\&} = \&_{j \in J} \boldsymbol{p}{:}\mathsf{m}'_j(\vec{T}'_j).S'_j \text{ or } S^{\&} = !\&_{j \in J} \boldsymbol{p}{:}\mathsf{m}'_j(\vec{T}'_j).S'_j$$
$$\exists K \subseteq I \text{ such that } \forall i \in K : \boldsymbol{r_i} = \boldsymbol{q}$$
$$K \subseteq J \text{ and } \forall i \in K : \vec{T}_i \leqslant \vec{T}'_i$$

By the definition of $\leqslant$, it follows that $K \supseteq K'$ and therefore $K' \subseteq J$. Hence, since $\mathsf{safe}_!(\Gamma_1)$, $\forall i \in K' : \vec{T}''_i \leqslant \vec{T}'_i$; therefore Γ_1 can reduce, at the very least, by the transitions enabled on the supertype. That is, $\exists \Gamma'_1 : \Gamma_1 \rightarrow \Gamma'_1$. Furthermore, from Definition 4.5, type continuations preserve subtyping. Therefore $\Gamma'_1 \leqslant \Gamma'_2$.
The second case is when $\Gamma_2 \rightarrow$ by rule Γ-Com$_2$. The proof case is similar to the above, just with $S^{\&}$ and $S^{\&\prime}$ being universal receives (observe that by Definition 4.5, this has no impact on the set inclusions obtained from subtyping). □

Theorem A.14 (Subject Reduction). If $\Gamma \vdash_D P$ with $\mathsf{safe}_!(\Gamma)$ and $P \rightarrow_D P'$, then $\exists \Gamma'$ s.t. $\Gamma \rightarrow^* \Gamma'$ and $\Gamma' \vdash_D P'$ with $\mathsf{safe}_!(\Gamma')$.

Proof. By induction on the derivation of $P \rightarrow_D P'$.

The assumptions are: **(A1)** $\Gamma \vdash_D P$; **(A2)** $\mathsf{safe}_!(\Gamma)$; and **(A3)** $P \rightarrow_D P'$.

Case R-C:

Let $P_{\oplus} = s[\boldsymbol{p}][\boldsymbol{q}] \oplus \mathsf{m}_k\langle \vec{d} \rangle \,.\, Q$	(definition)	(1)
Let $P_{\&} = s[\boldsymbol{q}][\boldsymbol{p}] \,\&_{i \in I}\, \mathsf{m}_i(\vec{b}_i) \,.\, Q'_i$	(definition)	(2)
$P = P_{\oplus} \mid P_{\&}$	(by R-C, (A3))	(3)
$P' = Q \mid Q'_k\{\vec{d}/\vec{b}_k\}$	(by R-C, (A3))	(4)
Let $n = \lvert\vec{d}\rvert$	(definition)	(5)

Without loss of generality, consider $\vec{d} = \vec{\boldsymbol{r}}, \vec{d}'$ where $\vec{d}'$ does not contain role values.

$\Gamma = \Gamma^{\oplus} \cdot \Gamma^{\&} \vdash_D P$	(by (3), (A1), T-$\lvert$)	(6)
$\Gamma^{\oplus} = \Gamma^{\oplus}_0 \cdot \Gamma^{\oplus}_c \cdot \Gamma^{\oplus}_r \,(\cdot \Gamma^{\oplus}_{d_i})_{i \in 1..n} \vdash_D P_{\oplus}$	(by (6), (3), T-$\oplus$)	(7)
$\Gamma^{\oplus}_c \vdash s[\boldsymbol{p}] : \oplus \boldsymbol{q}{:}\mathsf{m}_k(\vec{\boldsymbol{r}}, \vec{T}).S$	(by (7), T-$\oplus$)	(8)
$\Gamma^{\oplus}_r \vdash \boldsymbol{q} : \boldsymbol{q}$	(by (7), T-$\oplus$)	(9)
$\forall i \in 1..\lvert\vec{\boldsymbol{r}}\rvert : \Gamma^o_{d_i} \vdash d_i : \boldsymbol{r_i}$ and $\forall j \in \lvert\vec{\boldsymbol{r}}\rvert..\lvert\vec{T}\rvert : \Gamma^o_{d_j} \vdash d_j : T_j$	(by (7), T-$\oplus$)	(10)
$\Gamma^{\oplus}_0 + s[\boldsymbol{p}] : S \vdash_D Q$	(by (7), T-$\oplus$)	(11)

$\Gamma^{\&} = \Gamma^{\&}_0 \cdot \Gamma^{\&}_c \cdot \Gamma^{\&}_r \vdash_D P_{\&}$ (by (6), (3), T-&) (12)

$\Gamma^{\&}_c \vdash s[q] : \&_{i\in I} p{:}m'_i(\vec{\alpha_i}, \vec{T'_i}).S'_i$ (by (12), T-&) (13)

$\Gamma^{\&}_r \vdash p : p$ (by (12), T-&) (14)

$\forall i \in I : \Gamma^{\&}_0 + s[q] : S'_i + \overrightarrow{\alpha_i : \alpha_i} + \overrightarrow{x_i : T'_i} \vdash_D Q'_i$ (by (12), T-&) (15)

$\Gamma^{\oplus}_c \xrightarrow{s:p\oplus q:\mathsf{m}_k(\vec{r},\vec{T})} \Gamma^{\oplus\prime}_c$ (by (8), T-Wkn, T-Sub, Definition 4.5) (16)

$\Gamma^{\&}_c \xrightarrow{s:q\&p:\mathsf{m}'_j(\vec{\alpha}_j,\vec{T'_j})} \Gamma^{\&\prime}_{c_j}$ for $j \in I$ (by (13), T-Wkn, T-Sub, Definition 4.5) (17)

$\Gamma^{\oplus}_c \cdot \Gamma^{\&}_c \xrightarrow{s:p,q:\mathsf{m}_k} \Gamma^{\oplus\prime}_c + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\}$ (by (16), (17), (A2), lemma A.13, Γ-Com$_1$) (18)

$\Gamma^{\oplus\prime}_c \vdash s[p] : S$ (by (8), (16), Γ-$\oplus$, T-$\oplus$) (19)

$\Gamma^{\&\prime}_{c_k} \vdash s[q] : S'_k$ (by (13), (17), Γ-&, T-&) (20)

$\Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\} \vdash s[q] : S'_k\{\vec{r}/\vec{\alpha}_k\}$ (by (20), lemma A.4) (21)

$\vec{T} \leqslant \vec{T'_k}$ (by (16), (17), (18), Γ-Com$_1$) (22)

$\Gamma^{\&}_0 + \Gamma^{\&\prime}_{c_k} + \overrightarrow{\alpha_k : \alpha_k} + \overrightarrow{x_k : T'_k} \vdash_D Q'_k$ (by (15), (20), T-Wkn, T-Sub) (23)

$\Gamma^{\&}_0 + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\} + \overrightarrow{\alpha_k : \alpha_k} + \overrightarrow{x_k : T'_k} \vdash_D Q'_k\{\vec{r}/\vec{\alpha}_k\}$ (by (21), (23), lemma A.5) (24)

$\Gamma^{\&}_0 + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\} + \overrightarrow{x_k : T'_k} \vdash_D Q'_k\{\vec{r}/\vec{\alpha}_k\}$ (by (24), lemma A.7) (25)

$\Gamma^{\&}_0 + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\}(+\Gamma^{\oplus}_{d_i})_{i\in 1..n} \vdash_D Q'_k\{\vec{d}/\vec{b}_k\}$ (by (10), (25), (22), lemma A.8) (26)

$\Gamma^{\oplus}_0 + \Gamma^{\oplus\prime}_c \vdash_D Q$ (by (11), (19), T-Wkn, T-Sub) (27)

$(\Gamma^{\&}_0 + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\}(+\Gamma^{\oplus}_{d_i})_{i\in 1..n}) \cdot (\Gamma^{\oplus}_0 + \Gamma^{\oplus\prime}_c) \vdash_D P'$ (by (26), (27), T-|) (28)

$\Gamma^{\&}_0 + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\}(+\Gamma^{\oplus}_{d_i})_{i\in 1..n} + \Gamma^{\oplus}_0 + \Gamma^{\oplus\prime}_c \vdash_D P'$ (by (28), lemma A.1) (29)

$\Gamma \to \Gamma'$
$= (\Gamma^{\oplus\prime}_c + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\}) + (\Gamma^{\&}_0 \cdot \Gamma^{\&}_r \cdot \Gamma^{\oplus}_0 \cdot \Gamma^{\oplus}_r (\cdot \Gamma^{\oplus}_{d_i})_{i\in 1..n})$ (by (6), (18), Γ-Cong$_1$) (30)

$= \Gamma^{\oplus\prime}_c + \Gamma^{\&\prime}_{c_k}\{\vec{r}/\vec{\alpha}_k\} + \Gamma^{\&}_0 + \Gamma^{\&}_r + \Gamma^{\oplus}_0 + \Gamma^{\oplus}_r (+\Gamma^{\oplus}_{d_i})_{i\in 1..n}$ (by (30), lemma A.1) (31)

$\mathsf{end}(\Gamma^{\oplus}_r + \Gamma^{\&}_r)$ (by (9), (14), T-Wkn, T-q) (32)

$\Gamma' \vdash_D P'$ (by (29), (31), (32), lemma A.2) (33)

$\mathsf{safe}_!(\Gamma')$ (by (A2), (30), φ-$\to$) (34)

Case R-C holds. (by (30), (33), (34)) (35)

Case R-!C$_1$:

Let $R = !s[q][p]\ \&_{i\in I}\ \mathsf{m}_i(\vec{b}_i)\ .\ Q'_i$ (definition) (1)

Let $P_{\oplus} = s[p][q] \oplus \mathsf{m}_k\langle\vec{d}\rangle\ .\ Q$ (definition) (2)

Let $n = |\vec{d}|$ (definition) (3)

$P = P_{\oplus} \mid R$ (by R-!C$_1$ and (A3)) (4)

$P' = Q \mid R \mid Q'_k\{\vec{d}/\vec{b}_k\}$ (by R-!C$_1$ and (A3)) (5)

Without loss of generality, consider $\vec{d} = \vec{r}, \vec{d'}$ where $\vec{d'}$ does not contain role values.

$\Gamma = \Gamma^{\oplus} \cdot \Gamma^{!} \vdash_D P$ (by (4), (A1), and T-|) (6)

$\Gamma^{\oplus} = \Gamma^{\oplus}_0 \cdot \Gamma^{\oplus}_c \cdot \Gamma^{\oplus}_r \, (\cdot \Gamma^{\oplus}_{d_i})_{i \in 1..n} \vdash_D P_{\oplus}$ (by (6), (4), T-$\oplus$) (7)

$\Gamma^{\oplus}_c \vdash s[\boldsymbol{p}] : \oplus \boldsymbol{q}{:}\mathsf{m}_k(\vec{\boldsymbol{r}}, \vec{T}).S$ (by (7), T-$\oplus$) (8)

$\Gamma^{\oplus}_r \vdash \boldsymbol{q} : \boldsymbol{q}$ (by (7), T-$\oplus$) (9)

$\forall i \in 1..|\vec{\boldsymbol{r}}| : \Gamma^{\oplus}_{d_i} \vdash d_i : \boldsymbol{r_i}$ and $\forall j \in |\vec{\boldsymbol{r}}|..|\vec{T}| : \Gamma^{\oplus}_{d_j} \vdash d_j : T_j$ (by (7), T-$\oplus$) (10)

$\Gamma^{\oplus}_0 + s[\boldsymbol{p}] : S \vdash_D Q$ (by (7), T-$\oplus$) (11)

$\Gamma^{!} = \Gamma^{!}_0 \cdot \Gamma^{!}_c \vdash_D R$ (by (6), (4), T-!) (12)

$\Gamma^{!}_c \vdash s[\boldsymbol{q}] : !\&_{i \in I} \boldsymbol{p}{:}\mathsf{m}_i(\vec{\boldsymbol{\alpha}}_i, \vec{T'_i}).S'_i$ (by (12), T-!) (13)

$\mathsf{end}(\Gamma^{!}_0)$ (by (12), T-!) (14)

$\forall i \in I : \Gamma^{!}_0 + s[\boldsymbol{q}] : S'_i + \overrightarrow{\boldsymbol{\alpha_i} : \boldsymbol{\alpha'_i}} + \overrightarrow{x_i : T'_i} \hookleftarrow \boldsymbol{p} \vdash_D Q'_i$ (by (12), T-!) (15)

$\Gamma^{\oplus}_c \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}_k(\vec{\boldsymbol{r}},\vec{T})} \Gamma^{\oplus\prime}_c$ (by (8), T-Wkn, T-Sub, Definition 4.5) (16)

$\Gamma^{!}_c \xrightarrow{s:\boldsymbol{q}\&\boldsymbol{p}:\mathsf{m}'_j(\vec{\boldsymbol{\alpha}}_j,\vec{T'_j})} \Gamma^{!\prime}_{c_j}$ for $j \in I$ (by (13), T-Wkn, T-Sub, Definition 4.5) (17)

$\Gamma^{\oplus}_c \cdot \Gamma^{!}_c \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}_k} \Gamma^{\oplus\prime}_c + \Gamma^{!\prime}_{c_k}\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}}\}$ (by (16), (17), (A2), lemma A.13, Γ-Com$_1$) (18)

$\Gamma^{\oplus\prime}_c \vdash s[\boldsymbol{p}] : S$ (by (8), (16), Γ-$\oplus$, T-$\oplus$) (19)

$\Gamma^{!\prime}_{c_k} = \Gamma^{!}_c \cdot \Gamma_k$ such that: (by (13), (17), Γ-!) (20)

$\Gamma_k \vdash s[\boldsymbol{q}] : S'_k$ (by (20), (13), (17), Γ-!, T-!) (21)

$\Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\} \vdash s[\boldsymbol{q}] : S'_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\}$ (by (21), lemma A.4) (22)

$\vec{T} \leqslant \vec{T'_k}$ (by (16), (17), (18), Γ-Com$_1$) (23)

$\Gamma^{\oplus}_0 + \Gamma^{\oplus\prime}_c \vdash_D Q$ (by (11), (19)) (24)

$\Gamma^{!}_0 + \Gamma^{!}_c \vdash_D R$ (by (12)) (25)

$\Gamma^{!}_0 + \Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\} + \overrightarrow{\boldsymbol{\alpha}_i : \boldsymbol{\alpha'_i}} + \overrightarrow{x_i : T'_i} \vdash_D Q'_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\}$ (by (15), (22), lemma A.5) (26)

$\Gamma^{!}_0 + \Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\} + \overrightarrow{x_i : T'_i} \vdash_D Q'_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\}$ (by (26), lemma A.7) (27)

$\Gamma^{!}_0 + \Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\}(+\Gamma^{\oplus}_{d_i})_{i \in 1..n} \vdash_D Q'_k\{\vec{d}/\vec{b}_k\}$ (by (10), (15), (23), lemma A.8) (28)

$\Gamma^{\oplus}_0 + \Gamma^{\oplus\prime}_c + \Gamma^{!}_0 + \Gamma^{!}_c + \Gamma^{!}_0 + \Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\}(+\Gamma^{\oplus}_{d_i})_{i \in 1..n} \vdash_D P'$ (by (24), (25), (28), T-|) (29)

By (12) and (14), $\Gamma^{!}_0$ is end-typed. Assume without loss of generality that $\Gamma^{!}_0$ does not contain any session typed entities, since any end-typed channels could be grouped with a different split of the context instead. Then, it holds that $\Gamma^{!}_0 = \Gamma^{!}_0 \cdot \Gamma^{!}_0$

$\Gamma \to \Gamma'$

$= (\Gamma^{\oplus\prime}_c + (\Gamma^{!}_c \cdot \Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\})) + (\Gamma^{!}_0 \cdot \Gamma^{!}_0 \cdot \Gamma^{\oplus}_0 \cdot \Gamma^{\oplus}_r \, (\cdot \Gamma^{\oplus}_{d_i})_{i \in 1..n})$ (by (18), Γ-Cong$_1$) (30)

$= \Gamma^{\oplus}_0 + \Gamma^{\oplus\prime}_c + \Gamma^{!}_0 + \Gamma^{!}_c + \Gamma^{!}_0 + \Gamma_k\{\vec{\boldsymbol{r}}/\vec{\boldsymbol{\alpha}_k}\}(+\Gamma^{\oplus}_{d_i})_{i \in 1..n} + \Gamma^{\oplus}_r$ (by (30), lemma A.1) (31)

$\mathsf{end}(\Gamma^{\oplus}_r)$ (by (9), T-Wkn, T-$\boldsymbol{q}$) (32)

$\Gamma' \vdash_D P'$ (by (29), (31), (32), lemma A.2) (33)

$\mathsf{safe}_!(\Gamma')$ (by (A2), (30), φ-$\rightarrow$) (34)

Case R-!C$_1$ holds. (by (30), (33), (34)) (35)

Case R-!C$_2$:

Let $R = !s[q][\delta]\ \&_{i\in I}\ \mathsf{m}_i(\vec{b}_i)\ .\ Q'_i$ (definition) (1)

Let $P_\oplus = s[p][q] \oplus \mathsf{m}_k\langle\vec{d}\rangle\ .\ Q$ (definition) (2)

Let $n = |\vec{d}|$ (definition) (3)

$P = P_\oplus \mid R$ (by R-!C$_2$ and (A3)) (4)

$P' = Q \mid R \mid Q'_k\{\vec{d}/\vec{b}_k\}\{p/\delta\}$ (by R-!C$_2$ and (A3)) (5)

Without loss of generality, consider $\vec{d} = \vec{r}, \vec{d}'$ where $\vec{d}'$ does not contain role values.

$\Gamma = \Gamma^\oplus \cdot \Gamma^! \vdash_D P$ (by (4), (A1), and T-$|$) (6)

$\Gamma^\oplus = \Gamma^\oplus_0 \cdot \Gamma^\oplus_c \cdot \Gamma^\oplus_r \cdot (\Gamma^\oplus_{d_i})_{i\in 1..n} \vdash_D P_\oplus$ (by (6), (4), T-$\oplus$) (7)

$\Gamma^\oplus_c \vdash s[p] : \oplus q{:}\mathsf{m}_k(\vec{r},\vec{T}).S$ (by (7), T-$\oplus$) (8)

$\Gamma^\oplus_r \vdash q : q$ (by (7), T-$\oplus$) (9)

$\forall i \in 1..|\vec{r}| : \Gamma^\oplus_{d_i} \vdash d_i : r_i$ and $\forall j \in |\vec{r}|..|\vec{T}| : \Gamma^\oplus_{d_j} \vdash d_j : T_j$ (by (7), T-$\oplus$) (10)

$\Gamma^\oplus_0 + s[p] : S \vdash_D Q$ (by (7), T-$\oplus$) (11)

$\Gamma^! = \Gamma^!_0 \cdot \Gamma^!_c \vdash_D R$ (by (6), (4), T-!) (12)

$\Gamma^!_c \vdash s[q] : !\&_{i\in I}\delta{:}\mathsf{m}_i(\vec{\alpha}_i,\vec{T}'_i).S'_i$ (by (12), T-!) (13)

$\mathsf{end}(\Gamma^!_0)$ (by (12), T-!) (14)

$\forall i \in I : \Gamma^!_0 + s[q] : S'_i + \overrightarrow{\alpha_i : \alpha_i} + \overrightarrow{x_i : T'_i} \hookleftarrow \delta \vdash_D Q'_i$ (by (12), T-!) (15)

$\Gamma^\oplus_c \xrightarrow{s:p\oplus q:\mathsf{m}_k(\vec{r},\vec{T})} \Gamma^{\oplus\prime}_c$ (by (8), T-Wkn, T-Sub, Definition 4.5) (16)

$\Gamma^!_c \xrightarrow{s:q\&\delta:\mathsf{m}'_j(\vec{\alpha}_j,\vec{T}'_j)} \Gamma^{!\prime}_{c_j}$ for $j \in I$ (by (13), T-Wkn, T-Sub, Definition 4.5) (17)

$\Gamma^\oplus_c \cdot \Gamma^!_c \xrightarrow{s:p,q:\mathsf{m}_k} \Gamma^{\oplus\prime}_c + \Gamma^{!\prime}_{c_k}\{\vec{r},p/\vec{\alpha},\delta\}$ (by (16), (17), (A2), lemma A.13, Γ-Com$_1$) (18)

$\Gamma^{\oplus\prime}_c \vdash s[p] : S$ (by (8), (16), Γ-$\oplus$, T-$\oplus$) (19)

$\Gamma^{!\prime}_{c_k} = \Gamma^!_c \cdot \Gamma_k$ such that: (by (13), (17), Γ-!) (20)

$\Gamma_k \vdash s[q] : S'_k$ (by (20), (13), (17), Γ-!, T-!) (21)

$\Gamma_k\{\vec{r}/\vec{\alpha}_k\}\{p/\delta\} \vdash s[q] : S'_k\{\vec{r}/\vec{\alpha}_k\}\{p/\delta\}$ (by (21), lemma A.4) (22)

$\vec{T} \leqslant \vec{T}'_k$ (by (16), (17), (18), Γ-Com$_1$) (23)

$\Gamma^\oplus_0 + \Gamma^{\oplus\prime}_c \vdash_D Q$ (by (11), (19)) (24)

$\Gamma^!_0 + \Gamma^!_c \vdash_D R$ (by (12)) (25)

$\Gamma^!_0 + \Gamma_k\{\vec{r}/\vec{\alpha}_k\}\{p/\delta\} + \overrightarrow{\alpha_i : \alpha_i} + \overrightarrow{x_i : T'_i} \vdash_D Q'_k\{\vec{r}/\vec{\alpha}_k\}\{p/\delta\}$ (by (15), (22), lemma A.5) (26)

$\Gamma^!_0 + \Gamma_k\{\vec{r}/\vec{\alpha}_k\}\{p/\delta\} + \overrightarrow{x_i : T'_i} \vdash_D Q'_k\{\vec{r}/\vec{\alpha}_k\}\{p/\delta\}$ (by (26), lemma A.7) (27)

$\Gamma_0^! + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}(+\Gamma_{d_i}^{\oplus})_{i\in 1..n} \vdash_D Q'_k\{\vec{d}/\vec{b_k}\}$ ((10), (15), (23), lemma A.8) (28)

$\Gamma_0^{\oplus} + \Gamma_c^{\oplus\prime} + \Gamma_0^! + \Gamma_c^! + \Gamma_0^! + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}(+\Gamma_{d_i}^{\oplus})_{i\in 1..n} \vdash_D P'$
(by (24), (25), (28), T-|) (29)

By (12) and (14), $\Gamma_0^!$ is end-typed. Assume without loss of generality that $\Gamma_0^!$ does not contain any session typed entities, since any end-typed channels could be grouped with a different split of the context instead. Then, it holds that $\Gamma_0^! = \Gamma_0^! \cdot \Gamma_0^!$

$\Gamma \to \Gamma' = (\Gamma_c^{\oplus\prime} + (\Gamma_c^! \cdot \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\})) + (\Gamma_0^! \cdot \Gamma_0^! \cdot \Gamma_0^{\oplus} \cdot \Gamma_r^{\oplus}(\cdot\Gamma_{d_i}^{\oplus})_{i\in 1..n})$
(by (18), Γ-Cong$_1$) (30)

$= \Gamma_0^{\oplus} + \Gamma_c^{\oplus\prime} + \Gamma_0^! + \Gamma_c^! + \Gamma_0^! + \Gamma_k\{\vec{r}/\vec{\alpha_k}\}\{p/\delta\}(+\Gamma_{d_i}^{\oplus})_{i\in 1..n} + \Gamma_r^{\oplus}$
(by (30), lemma A.1) (31)

$\mathsf{end}(\Gamma_r^{\oplus})$ (by (9), T-Wkn, T-q) (32)

$\Gamma' \vdash_D P'$ (by (29), (31), (32), lemma A.2) (33)

$\mathsf{safe}_!(\Gamma')$ (by (A2), (30), φ-$\to$) (34)

Case R-!C$_2$ holds. (by (30), (33), (34)) (35)

Remaining cases are straight forward. □

A.3 Session Fidelity

Lemma A.15. Assume $\Gamma \vdash_D P$ with $\mathsf{one\text{-}role}_!(P)$. Then $P \equiv \big|_{p\in I} P_p$ and $\Gamma = \Gamma_0(\cdot\Gamma'_p, s[p] : U_p)_{p\in I'}$, where:

1. $\forall p \in I' : \Gamma'_p, s[p] : U_p \vdash_D P_p$ and $c : U \notin \Gamma'_p$;
2. $\forall q \in I \setminus I' : P_q \equiv \mathbf{0}$;
3. $c : U \notin \Gamma_0$.

Proof. First observe that by Definition 4.24(2), $P \equiv \big|_{p\in I} P_p$ where $\forall p \in I : P_p \equiv \mathbf{0}$ or P_p only plays role p. Therefore, by rule T-|, $\Gamma = (\cdot\Gamma_p)_{p\in I}$ where $\forall p \in I : \Gamma_p \vdash_D P_p$. For each split of the context, there are two cases to consider.

If $P_p \equiv \mathbf{0}$, then by rule T-0, $\mathsf{end}(\Gamma_p)$.

Otherwise, P_p only plays role p. Since $\Gamma_p \vdash_D P_p$, then Γ_p serves as witness for Definition 4.24(1). Hence, it can be concluded that $\Gamma_p = \Gamma''_p, s[p] : U_p$ with $U_p \not\leqslant \mathbf{end}$ and $\mathsf{end}(\Gamma''_p)$.

Therefore, $\Gamma = (\cdot\Gamma''_p, s[p] : U_p)_{p\in I''}(\cdot\Gamma_q)_{q\in I\setminus I''}$ where $\forall p \in I'' : \Gamma''_p, s[p] : U_p \vdash_D P_p$ with

$\mathsf{end}(\Gamma''_p)$ and $\forall q \in I \setminus I'' : \Gamma_q \vdash_D P_q \equiv \mathbf{0}$ with $\mathsf{end}(\Gamma_q)$.

Without loss of generality, assume that all end-typed channels are grouped in $I \setminus I''$. Hence, $\forall p \in I'' : \Gamma''_p, s[p] : U_p \vdash_D P_p$ with $c : U \notin \Gamma''_p$ and $\forall q \in I \setminus I'' : \Gamma_q \vdash_D P_q \equiv \mathbf{0}$ with $\mathsf{end}(\Gamma_q)$ and either $\Gamma_q = \Gamma''_q, s[q] : \mathbf{end}$ or $c : U \notin \Gamma''_q$.

Thus, $(\cdot \Gamma_q)_{q \in I \setminus I''} = \Gamma_0 (\cdot \Gamma_q)_{q \in I'''}$ where $c : U \notin \Gamma_0$ and $I''' \subseteq I \setminus I''$.

Let $I' = (I \setminus I'') \cup I'''$, then $\Gamma = \Gamma_0 (\cdot \Gamma'_p, s[p] : U_p)_{p \in I'}$. □

Lemma A.16. Assume $\Gamma_0, s[p] : U_p \vdash_D P_p$ with $c : U \notin \Gamma_0$ and either P_p only plays role p or $P_p \equiv \mathbf{0}$. Then:

1. if $S_1 | \cdots | S_n \leqslant U_p$, then $P_p \equiv P_1 | \cdots | P_n$ and $\forall i \in 1..n : \Gamma_0, s[p] : S_i \vdash_D P_i$ and P_i only plays role p;
2. if $!\&_{i \in I} \rho : \mathsf{m}_i(\vec{T_i}).S_i \leqslant U_p$, then either:
   (a) $P_p \equiv !s[p][\rho]\&_{i \in I'} \mathsf{m}_i(\vec{b_i}).P'_i$ and $I \subseteq I'$; or
   (b) $P_p \equiv \mathrm{X}\langle \vec{V} \rangle$ and $D(\mathrm{X}) = (\vec{b'}, \vec{T'}, Q)$ and $Q\{\vec{V}/\vec{b'}\} \equiv !s[p][\rho]\&_{i \in I'} \mathsf{m}_i(\vec{b_i}).P'_i$ and $I \subseteq I'$;
3. if $\&_{i \in I} q : \mathsf{m}_i(\vec{T_i}).S_i \leqslant U_p$, then either:
   (a) $P_p \equiv s[p][q]\&_{i \in I'} \mathsf{m}_i(\vec{b_i}).P'_i$ and $I \subseteq I'$; or
   (b) $P_p \equiv \mathrm{X}\langle \vec{V} \rangle$ and $D(\mathrm{X}) = (\vec{b'}, \vec{T'}, Q)$ and $Q\{\vec{V}/\vec{b'}\} \equiv s[p][q]\&_{i \in I'} \mathsf{m}_i(\vec{b_i}).P'_i$ and $I \subseteq I'$;
4. if $\oplus_{i \in I} q_i : \mathsf{m}_i(\vec{T_i}).S_i \leqslant U_p$, then either:
   (a) $P_p \equiv \sum_{i \in I'} s[p][q_i] \oplus \mathsf{m}_i\langle \vec{V_i} \rangle.P'_i$ and $I \supseteq I'$; or
   (b) $P_p \equiv \mathrm{X}\langle \vec{V} \rangle$ and $D(\mathrm{X}) = (\vec{b'}, \vec{T'}, Q)$ and $Q\{\vec{V}/\vec{b'}\} \equiv \sum_{i \in I'} s[p][q_i] \oplus \mathsf{m}_i\langle \vec{V_i} \rangle.P'_i$ and $I \supseteq I'$;
5. if $\mathbf{end} \leqslant U_p$, then $P_p \equiv \mathbf{0}$.

Proof. By case analysis on the 5 possible implications.

Case 1:

Follows from the definition of context splits (Figure 4.5), observing that since $c : U \notin \Gamma_0$ then $\Gamma_0, s[p] : S_1 | \cdots | S_n = \Gamma_0, s[p] : S_1 \cdot \ldots \cdot \Gamma_0, s[p] : S_n$.

Furthermore, by rule T-|, $\forall i \in 1..n : \Gamma_0, s[p] : S_i \vdash_D P_i$.

Lastly, from the hypothesis it follows that D is guarded, $\mathsf{fv}(P) = \emptyset$ and $U_{\boldsymbol{p}} \not\leqslant \mathbf{end}$, thus $\forall i \in 1..n : \mathsf{fv}(P_i) = \emptyset$ and $S_i \not\leqslant \mathbf{end}$. (Notice that any **end** in the parallel type can be removed via type congruence, and also reflected in the process.) Therefore, by Definition 4.24, $\forall i \in 1..n : P_i$ only plays role $\boldsymbol{p}$, by $\Gamma_0, s[\boldsymbol{p}] : S_i$.

Case 2:

From the hypothesis, and by subtyping and narrowing, $\Gamma_0, s[\boldsymbol{p}] : !\&_{i\in I}\boldsymbol{\rho}{:}m_i(\vec{T_i}).S_i \vdash_D P_{\boldsymbol{p}}$. By inversion of typing rules, there are two possible rules that could be applied.

Rule T-!:

$\Gamma_0, s[\boldsymbol{p}] : !\&_{i\in I}\boldsymbol{\rho}{:}\mathsf{m}_i(\vec{T_i}).S_i \vdash_D !s[\boldsymbol{p}][\boldsymbol{\rho}]\&_{i\in I}\mathsf{m}_i(\vec{b_i}).P_i'$ (by T-!) (1)

$\Gamma_0, s[\boldsymbol{p}] : U_{\boldsymbol{p}} \vdash_D !s[\boldsymbol{p}][\boldsymbol{\rho}]\&_{i\in I'}\mathsf{m}_i(\vec{b_i}).P_i'$ where $I \subseteq I'$ (by (1) and Definition 4.5) (2)

$P_{\boldsymbol{p}} \equiv !s[\boldsymbol{p}][\boldsymbol{\rho}]\&_{i\in I'}\mathsf{m}_i(\vec{b_i}).P_i'$ (by (2) and subject congruence (lemma A.12)) (3)

Rule T-X:

By inversion of T-X, $P_{\boldsymbol{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $\exists k \in 1..|\vec{V}| : V_k = s[\boldsymbol{p}]$ and $s[\boldsymbol{p}] : U_{\boldsymbol{p}} \vdash V_k : T_k'$. Without loss of generality, assume $k = 1$, then $D(\mathrm{X}) = ((x, \vec{b}), (T_k, \vec{T'}), Q)$.

Assuming that $\Gamma_0, s[\boldsymbol{p}] : U_{\boldsymbol{p}} \vdash_D P_{\boldsymbol{p}}$ is derived from a well-typed program, then it follows that $x : T_k', \{\overrightarrow{b : T'}\} \vdash_D Q$ by rule T-$\mathscr{P}$. Furthermore, by Definition 4.23, it follows that Q cannot be typed by a further application of rule T-X, and thus must instead be typed by rule T-!. Following the same reasoning as outlined in the previous subcase, it must be that $Q \equiv !x[\boldsymbol{\rho}]\&_{i\in I'}\mathsf{m}_i(\vec{b_i}).P_i'$ and $I \subseteq I'$

Case 3, 4: Similar to Case 2.

Case 5:

Since $\mathbf{end} \leqslant U_{\boldsymbol{p}}$ and $c : U \notin \Gamma_0$, then $\mathsf{end}(\Gamma_0, s[\boldsymbol{p}] : U_{\boldsymbol{p}})$. By inversion of the typing rules, the only possible shape for $P_{\boldsymbol{p}}$ is $P_{\boldsymbol{p}} \equiv \mathbf{0}$ via rule T-**0**. □

Theorem A.17 (Session Fidelity). Assume $\Gamma \vdash_D P$ with $\mathsf{fid}_!(\Gamma)$ and $\mathsf{one\text{-}role}_!(P)$. Then, $\Gamma \to$ implies $\exists \Gamma', P'$ s.t.:

(i) $\Gamma \to \Gamma'$;

(ii) $P \to_D^* P'$;

(iii) $\Gamma' \vdash_D P'$ with $\mathsf{fid}_!(\Gamma')$; and

(iv) $\mathsf{one\text{-}role}(P')$.

Proof. By induction on the derivation of $\Gamma \to$. There are two possible cases by which Γ can be enabled with a reduction: by rule Γ-Com$_1$ or by rule Γ-Com$_2$. The following elaborates on the latter case (the other case is similar).

The assumptions are: **(A1)** $\Gamma \vdash_D P$; **(A2)** $\mathsf{fid}_!(\Gamma)$; **(A3)** $\mathsf{one\text{-}role}_!(P)$; and **(A4)** $\Gamma\!\rightarrow$.

The proof begins by inferring some structure on both the types and processes, utilizing lemma A.15.

$P \equiv \big|_{p\in I} P_p$ and $\Gamma = \Gamma_0(\cdot\, \Gamma'_p, s[p] : U_p)_{p\in I'}$ (by (A1), (A3), and lemma A.15) (1)

$\forall p \in I' : \Gamma'_p, s[p] : U_p \vdash_D P_p$ and $c : U \notin \Gamma'_p$ (by (A1), (A3), and lemma A.15) (2)

$\forall q \in I \setminus I' : P_q \equiv \mathbf{0}$ (by (A1), (A3), and lemma A.15) (3)

$c : U \notin \Gamma_0$ (by (A1), (A3), and lemma A.15) (4)

Since the context is known to be enabled with a reduction (by assumption), the actions emitted by the context can be inferred from rule Γ-Com$_2$. Specifically, $\exists p, q \in I'$ such that:

$\Gamma'_p, s[p] : U_p \xrightarrow{s:p\oplus q:\mathsf{m}(\vec{r},\vec{T}_1)} \Gamma'_p, s[p] : U'_p$ (by (1), (A4) and Γ-Com$_2$) (5)

$\Gamma'_q, s[q] : U_q \xrightarrow{s:q\&\alpha':\mathsf{m}(\vec{\alpha},\vec{T}_2)} \Gamma'_q, s[q] : U'_q$ (by (1), (A4) and Γ-Com$_2$) (6)

$\vec{T}_1 \leqslant \vec{T}_2$ (by (1), (A4) and Γ-Com$_2$) (7)

$(\Gamma'_p, s[p] : U_p) \cdot (\Gamma'_q, s[q] : U_q) \xrightarrow{s:p,q:\mathsf{m}} (\Gamma'_p, s[p] : U'_p) + (\Gamma'_q, s[q] : U'_q\{\vec{r}/\vec{\alpha}\}\{p/\alpha'\})$
(by (5), (6), (7), and Γ-Com$_2$) (8)

From these actions, the shapes of the types can be outlined, following from the type semantics in Figure 4.7.

$\oplus_{j\in J} q_j{:}\mathsf{m}_j(\vec{T}_j).S_j \leqslant U_p \leqslant q \oplus \mathsf{m}(\vec{r},\vec{T}_1).S$ (by (5), Γ-$\oplus$) (9)

$\exists l \in J : q_l = q \wedge \mathsf{m}_l = \mathsf{m} \wedge \vec{T}_l \geqslant \vec{T}_1$ (by (9), Definition 4.5) (10)

$!\&_{k\in K} \alpha'{:}\mathsf{m}'_k(\vec{\delta}_k,\vec{T}'_k).S'_k \leqslant U_q$ and $\exists l \in K : \mathsf{m}'_l = \mathsf{m}$ (by (6), Γ-!) (11)

$\vec{T}'_l \leqslant \vec{T}_2$ (by (11), (6), Definition 4.5) (12)

The shapes of processes can now be inferred from the types, using (2), (9), lemma A.16 for P_p, and (2), (11), lemma A.16 for P_q. Note that by the session inversion lemma, there are two possible shapes for each process; thus a total of four possible permutations to consider, resulting in 4 subcases.

Case $\oplus$!: Consider:

$P_p \equiv \sum_{j\in J'} s[p][q_j] \oplus \mathsf{m}_j\langle\vec{V}_j\rangle.P'_j$ and $J \supseteq J'$ (by case assumption) (13)

$P_q \equiv !s[q][\alpha'] \&_{k\in K'} \mathsf{m}'_k(\vec{b}_k).P''_k$ and $K \subseteq K'$ (by case assumption) (14)

This is where the novel fidelitous property comes into play. As it stands, there is no guarantee that $\exists x \in J' : q_x = q$, as the implementation is a subset of the type and could hence implement a part of the selection that does not include the role considered for communication in (8). However, by (8), it is guaranteed that at least one (type-level) communication is possible. Then by (A2) and table 4.1, it follows that communication should be eventually possible for every path in J, and thus also J' (since $J' \subseteq J$ by (13)).

Therefore, it can be assumed without loss of generality that the nondeterministic choice in $P_{\boldsymbol{p}}$ reduces to a path x in J' such that $\boldsymbol{q_x} = \boldsymbol{q}$.

$P_{\boldsymbol{p}} \to P'_{\boldsymbol{p}} \equiv s[\boldsymbol{p}][\boldsymbol{q}] \oplus \mathsf{m}_x\langle \vec{V}_x \rangle.P'_x$ and $x \in J'$ (explanation above) (15)

$\exists y \in K : \mathsf{m}_y = \mathsf{m}_x$ (by (15), (A2), Definition 4.22) (16)

$y \in K'$ (by (16) and (14)) (17)

$P'_{\boldsymbol{p}} \mid P_{\boldsymbol{q}} \to P'_x \mid P_{\boldsymbol{q}} \mid P''_y\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{V}_x/\vec{b}_y\}$ (by (14), (15), (17), R-!C_2) (18)

$\Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U_{\boldsymbol{p}} \xrightarrow{s:\boldsymbol{p}\oplus\boldsymbol{q}:\mathsf{m}_x(\vec{\boldsymbol{r'}},\vec{T}_x)} \Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U'_{\boldsymbol{p}}$ (by (2), (15), Γ-$\oplus$) (19)

$(\Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U_{\boldsymbol{p}}) \cdot (\Gamma'_{\boldsymbol{q}}, s[\boldsymbol{q}] : U_{\boldsymbol{q}}) \xrightarrow{s:\boldsymbol{p},\boldsymbol{q}:\mathsf{m}_x} (\Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U'_{\boldsymbol{p}}) + (\Gamma'_{\boldsymbol{q}}, s[\boldsymbol{q}] : U'_{\boldsymbol{q}}\{\vec{\boldsymbol{r'}}/\vec{\boldsymbol{\delta}}\}\{\boldsymbol{p}/\boldsymbol{\alpha'}\})$
(by (19), (8), (A2)) (20)

$\Gamma'_{\boldsymbol{q}}, s[\boldsymbol{q}] : U_{\boldsymbol{q}} \xrightarrow{s:\boldsymbol{q}\oplus\boldsymbol{\alpha'}:\mathsf{m}_x(\vec{\boldsymbol{\delta}},\vec{T}'_x)} \Gamma'_{\boldsymbol{q}}, s[\boldsymbol{p}] : U'_{\boldsymbol{q}}$ (by (20), (19), Γ-Com$_2$) (21)

$\vec{T}_x \leqslant \vec{T}'_x$ (by (21), (20), (19), Γ-Com$_2$) (22)

The subcase is concluded by observing that the reduced contexts type the reduced processes, and that the reduced processes still adhere to the one-role$_!$ requirements.

$\Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U'_{\boldsymbol{p}} \vdash_D P'_x$ (by (2), (19), T-$\oplus$) (23)

$\Gamma'_{\boldsymbol{q}}, s[\boldsymbol{p}] : U'_{\boldsymbol{q}}\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{\boldsymbol{r'}}/\vec{\boldsymbol{\delta}}\} \vdash_D P''_y\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{V}_x/\vec{b}_y\}$
(by (2), (21), (22), (23), T-!, and both substitution lemmas) (24)

Note that for the above, all the types for payloads are present in $\Gamma'_{\boldsymbol{q}}$ since by the session fidelity assumptions, payloads can only contain non-session-typed entities; these are copied through context splits (Figure 4.5).

$P \to^* P' \mid P'_x \mid P_{\boldsymbol{q}} \mid P''_y\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{V}_x/\vec{b}_y\}$ with one-role$_!(P')$ (by (A3), (15), (18)) (25)

one-role$_!(P'_x \mid P_{\boldsymbol{q}} \mid P''_y\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{V}_x/\vec{b}_y\})$ (by (1), (23), (24), Definition 4.24) (26)

$\Gamma \to \Gamma'$ (by (20)) (27)

$P \to^* P' \mid P'_x \mid P_{\boldsymbol{q}} \mid P''_y\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{V}_x/\vec{b}_y\}$ (by (25)) (28)

$\Gamma' \vdash_D P' \mid P'_x \mid P_{\boldsymbol{q}} \mid P''_y\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{V}_x/\vec{b}_y\}$ (by (2), (20), (23), (24), T-|) (29)

fid$_!(\Gamma')$ (by (A2), (27), φ-$\to$) (30)

one-role$_!(P' \mid P'_x \mid P_{\boldsymbol{q}} \mid P''_y\{\boldsymbol{p}/\boldsymbol{\alpha'}\}\{\vec{V}_x/\vec{b}_y\})$ (by (25), (26)) (31)

Case $\oplus!$ holds. (by (25), (27)–(31)) (32)

The remaining subcases all follow similar reasoning, except with the additional step of allowing for process reduction via rule R-X before communication becomes available.

□

Appendix B

Further Definitions and Proofs for MAGπ and MAGπ!

The following two sections assume a syntax for MAGπ! without the fidelity restrictions.

B.1 Definitions

The full definition for compatibility in MAGπ! is given in Definition B.1, and an *end*-typed context is formalised in Definition B.2. Context operations for MAGπ! are similar to those of MAGπ, just extended to handle role singletons, and are defined in Figure B.1. Using these, the typing rules for MAGπ! are listed in Figure B.2.

Definition B.1 (Compatible). The compatibility relation $\sqsubseteq$ is co-inductively defined on types by the following inference rules:

$$\frac{(\vec{T}_i \sqsubseteq \vec{T}'_i)_{i\in I} \qquad (S_i \sqsubseteq S'_i)_{i\in I}}{\&_{i\in I}\boldsymbol{\rho_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq \&_{i\in I}\boldsymbol{\rho_i}{:}\mathsf{m}_i(\vec{T}'_i).S'_i} \qquad \frac{(\vec{T}_i \sqsupseteq \vec{T}'_i)_{i\in I} \qquad (S_i \sqsubseteq S'_i)_{i\in I}}{\oplus_{i\in I\cup J}\boldsymbol{\rho_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq \oplus_{i\in I}\boldsymbol{\rho_i}{:}\mathsf{m}_i(\vec{T}'_i).S'_i}$$

$$\frac{(\vec{T}_i \sqsubseteq \vec{T}'_i)_{i\in I} \qquad (S_i \sqsubseteq S'_i)_{i\in I}}{!\boldsymbol{\alpha}\&_{i\in I}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq !\boldsymbol{\alpha}\&_{i\in I}\mathsf{m}_i(\vec{T}'_i).S'_i} \qquad \frac{S_1^{\&} \sqsubseteq S_2^{\&}}{!S_1^{\&} \sqsubseteq !S_2^{\&}} \qquad \frac{U \sqsubseteq U' \qquad S \sqsubseteq S'}{U \,|\, S \sqsubseteq U' \,|\, S'}$$

$$\frac{S_1^{\&} \sqsubseteq S_2^{\&} \qquad S'_1 \sqsubseteq S'_2}{S_1^{\&}, \textcircled{\scriptsize t}.S'_1 \sqsubseteq S_2^{\&}, \textcircled{\scriptsize t}.S'_2} \qquad \frac{S\{\mu \mathrm{t}.S/\mathrm{t}\} \sqsubseteq S'}{\mu \mathrm{t}.S \sqsubseteq S'} \qquad \frac{S \sqsubseteq S'\{\mu \mathrm{t}.S'/\mathrm{t}\}}{S \sqsubseteq \mu \mathrm{t}.S'} \qquad \frac{}{T \sqsubseteq T}$$

Context splitting $\boxed{\Gamma = \Gamma_1 \cdot \Gamma_2}$

$$\frac{}{\emptyset = \emptyset \cdot \emptyset} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, x : B = (\Gamma_1, x : B) \cdot (\Gamma_2, x : B)} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, \alpha : \alpha = \Gamma_1, \alpha : \alpha \cdot \Gamma_2, \alpha : \alpha}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = (\Gamma_1, c : U) \cdot \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U = \Gamma_1 \cdot (\Gamma_2, c : U)} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, c : U_1 | U_2 = \Gamma_1, c : U_1 \cdot \Gamma_2, c : U_2}$$

$$\frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, s : \widetilde{M} = (\Gamma_1, s : \widetilde{M}) \cdot \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \cdot \Gamma_2}{\Gamma, s : \widetilde{M} = \Gamma_1 \cdot (\Gamma_2, s : \widetilde{M})}$$

Context addition $\boxed{\Gamma_1 + \Gamma_2 = \Gamma}$

$$\frac{}{\Gamma + \emptyset = \Gamma} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{(\Gamma_1, x : B) + (\Gamma_2, x : B) = \Gamma, x : B} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma}{(\Gamma_1, s : \widetilde{M}_1) + (\Gamma_2, s : \widetilde{M}_2) = \Gamma, s : \widetilde{M}_1, \widetilde{M}_2}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma}{\Gamma_1, \alpha : \alpha + \Gamma_2, \alpha : \alpha = \Gamma, \alpha : \alpha} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad s \notin \mathsf{dom}(\Gamma_2)}{(\Gamma_1, s : \widetilde{M}) + \Gamma_2 = \Gamma, s : \widetilde{M}} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad s \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + (\Gamma_2, s : \widetilde{M}) = \Gamma, s : \widetilde{M}}$$

$$\frac{\Gamma_1 + \Gamma_2 = \Gamma \quad a \notin \mathsf{dom}(\Gamma_2)}{(\Gamma_1, a : T) + \Gamma_2 = \Gamma, a : T} \qquad \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad a \notin \mathsf{dom}(\Gamma_1)}{\Gamma_1 + (\Gamma_2, a : T) = \Gamma, a : T}$$

Buffer Extraction and Role Insertion $\boxed{\Gamma = \Gamma_1 ; \Gamma_2}$ $\boxed{\Gamma \hookleftarrow \rho}$

$$\frac{\Gamma = \Gamma_1, \Gamma_2 \quad \Gamma_1 = \{a_i : T_i\}_{i \in I} \quad \Gamma_2 = \{s_j : \widetilde{M}_j\}_{j \in J}}{\Gamma = \Gamma_1 ; \Gamma_2} \qquad \Gamma \hookleftarrow q = \Gamma \qquad \Gamma \hookleftarrow \alpha = \Gamma + \alpha : \alpha$$

Figure B.1: Type context operations.

Definition B.2 (End-typed environment). A context is *end-typed*, written $\mathsf{end}(\Gamma)$, iff its channels have type **end**, and it is void of buffer types.

$$\frac{}{\mathsf{end}(\emptyset)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(c : \mathbf{end}, \Gamma)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(x : B, \Gamma)} \qquad \frac{\mathsf{end}(\Gamma)}{\mathsf{end}(\alpha : \alpha, \Gamma)}$$

Typing Rules for Values and Buffers $\boxed{\Gamma \vdash V : T}$ $\boxed{\Gamma \vdash \mathcal{B}}$

$$\text{T-Sub}\ \frac{T \sqsubseteq T'}{c : T \vdash c : T'} \qquad \text{T-Wkn}\ \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad \Gamma_1 \vdash V : T \quad \mathsf{end}(\Gamma_2)}{\Gamma \vdash V : T} \qquad \text{T-B}\ \frac{v \in B}{\emptyset \vdash v : B}$$

$$\text{T-Wkn}_2\ \frac{\Gamma_1 + \Gamma_2 = \Gamma \quad \Gamma_1 \vdash \mathcal{B} \quad \mathsf{gc}(\Gamma_2)}{\Gamma \vdash \mathcal{B}} \qquad \text{T-Empty}_1\ \frac{\Gamma \vdash \mathcal{B}}{\Gamma \vdash \mathcal{B}, s : \emptyset}$$

$$\text{T-Empty}_2\ \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash \emptyset} \qquad \text{T-Msg}\ \frac{(\Gamma_i \vdash d_i : T_i)_{i \in 1..n} \quad \Gamma\,;\, \Gamma' + s : \widetilde{M} \vdash \mathcal{B}, s : \widetilde{\mathcal{M}}}{\Gamma\,(\cdot\,\Gamma_i)_{i \in 1..n}\,;\, \Gamma' \cdot s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}((T_i)_{i \in 1..n})), \widetilde{M} \vdash \mathcal{B}, s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle (d_i)_{i \in 1..n} \rangle), \widetilde{\mathcal{M}}}$$

Typing Rules for Programs and Processes $\boxed{\vdash \mathscr{P}}$ $\boxed{\Gamma \vdash_D P :: \mathcal{B}}$ $\boxed{\Gamma \vdash_D P}$

$$\text{T-}\mathscr{P}\ \frac{\emptyset \vdash_D P :: \mathcal{B} \quad \forall i \in I : \{\overrightarrow{b_i : T_i}\} \vdash_D Q_i}{\vdash (P :: \mathcal{B}, D = \{\mathrm{X}_i \mapsto (\vec{b}_i, \vec{T}_i, Q_i)\}_{i \in I})} \qquad \text{T-}\mathcal{B}\ \frac{\Gamma \vdash_D P \quad \Gamma_d\,;\, \Gamma_s \vdash \mathcal{B}}{\Gamma \cdot \Gamma_d\,;\, \Gamma_s \vdash_D P :: \mathcal{B}}$$

$$\text{T-}|\ \frac{\Gamma_1 \vdash_D P_1 \quad \Gamma_2 \vdash_D P_2}{\Gamma_1 \cdot \Gamma_2 \vdash_D P_1 | P_2} \qquad \text{T-}\mathbf{0}\ \frac{\mathsf{end}(\Gamma)}{\Gamma \vdash_D \mathbf{0}}$$

$$\text{T-}\nu\ \frac{\varphi(\mathtt{assoc}_s(\Psi)) \quad s \notin \Gamma \quad \varphi \text{ is a } \mathfrak{R}\text{-}\textit{safety} \text{ property} \quad \Gamma + \mathtt{assoc}_s(\Psi) \vdash_D P}{\Gamma \vdash_D (\nu s^{\mathfrak{R}} : \Psi)\ P} \qquad \text{T-}\oplus\ \frac{\Gamma_\oplus \vdash c : \oplus \boldsymbol{\rho} : \mathsf{m}(\vec{T}).S \quad (\Gamma_i \vdash V_i : T_i)_{i \in 1..n} \quad \Gamma + c : S \vdash_D P}{\Gamma \cdot \Gamma_\oplus(\cdot\,\Gamma_i)_{i \in 1..n} \vdash_D c \oplus [\boldsymbol{\rho}]\,\mathsf{m}\langle \vec{V} \rangle.P}$$

$$\text{T-X}\ \frac{\mathsf{end}(\Gamma_0) \quad D(\mathrm{X}) = (\vec{x}, \vec{T}, P) \quad \forall i \in 1..n : \Gamma_i \vdash V_i : T_i}{\Gamma_0(\cdot\,\Gamma_i)_{i \in 1..n} \vdash_D \mathrm{X}\langle (V_i)_{i \in 1..n} \rangle}$$

$$\text{T-}\&\ \frac{\Gamma_\& \vdash c : \&_{i \in I} \boldsymbol{\rho_i} : \mathsf{m}_i(\vec{T}_i).S_i \left[, \textcircled{\scriptsize t}.S'\right] \quad (\Gamma + c : S_i + \overrightarrow{b_i : T_i} \vdash_D P_i)_{i \in I} \quad \left[\Gamma + c : S' \vdash_D Q\right]}{\Gamma \cdot \Gamma_\& \vdash_D c \,\&_{i \in I}\, [\boldsymbol{\rho_i}]\,\mathsf{m}_i(\vec{b}_i).P_i \left[, \textcircled{\scriptsize t}.Q\right]}$$

$$\text{T-}!\ \frac{\Gamma_! \vdash c : !\&_{i \in I} \boldsymbol{\rho} : \mathsf{m}_i(\vec{T}_i).S_i \quad \mathsf{end}(\Gamma) \quad (\Gamma + c : S_i + \overrightarrow{b_i : T_i} \leftrightarrow \boldsymbol{\rho} \vdash_D P_i)_{i \in I}}{\Gamma \cdot \Gamma_! \vdash_D !c[\boldsymbol{\rho}]\ \&_{i \in I}\, \mathsf{m}_i(\vec{b}_i)\ .\ P_i} \qquad \text{T-}+\ \frac{(\Gamma \vdash_D P_i^{\oplus})_{i \in I}}{\Gamma \vdash_D \sum_{i \in I} P_i^{\oplus}}$$

Figure B.2: Typing rules.

B.2 Subject Reduction

Theorem B.3 (Subject Reduction). If $P :: \mathcal{B} \to_{D;\mathcal{R}} P' :: \mathcal{B}'$ and $\Gamma \vdash_D P :: \mathcal{B}$ with $\mathcal{R}$-safe$_!(\Gamma)$, then $\exists \Gamma'$ s.t. $\Gamma \to^* \Gamma'$ and $\Gamma' \vdash_D P' :: \mathcal{B}'$ with $\mathcal{R}$-safe$_!(\Gamma')$.

Proof. By induction on the derivation of $P :: \mathcal{B} \to_{D;\mathcal{R}} P' :: \mathcal{B}'$. The assumptions are: **(A1)** $\Gamma \vdash_D P :: \mathcal{B}$; **(A2)** $\mathcal{R}$-safe$_!(\Gamma)$; and **(A3)** $P :: \mathcal{B} \to_{D;\mathcal{R}} P' :: \mathcal{B}'$. The proof follows similar steps to the previous subject reduction proofs. First, the shape of $P :: \mathcal{B}$ and $P' :: \mathcal{B}'$ are inferred from (A3) and the reduction rule in question. Then, by inversion of typing rules and (A1), the shape of the types can be determined. Lastly, (A2) may possibly be used if the context needs to reduce via a communication action, and the reduced process is shown

to be typed under a possibly reduced context.

Case R-⏲:

$P :: \mathcal{B} = s[\boldsymbol{q}]\&_{i\in I}[\boldsymbol{p_i}]\mathsf{m}_i(\vec{b_i})\ .\ Q_i,\ ⏲.\ Q' :: \mathcal{B}$ (by R-⏲ and **(A3)**) (1)

$P' :: \mathcal{B}' = Q' :: \mathcal{B}$ (by R-⏲ and **(A3)**) (2)

$\Gamma = \Gamma_P \cdot \Gamma_d; \Gamma_s \vdash_D P :: \mathcal{B}$ (by (1), **(A1)**, T-$\mathcal{B}$) (3)

$\Gamma_P = \Gamma_0 \cdot \Gamma_\& \vdash_D s[\boldsymbol{q}]\&_{i\in I}[\boldsymbol{p_i}]\mathsf{m}_i(\vec{b_i})\ .\ Q_i,\ ⏲.\ Q'$ (by (3), T-$\mathcal{B}$) (4)

$\Gamma_d; \Gamma_s \vdash \mathcal{B}$ (by (3), T-$\mathcal{B}$) (5)

$\Gamma_\& \vdash s[\boldsymbol{q}] : \&_{i\in I}\boldsymbol{p_i}{:}\mathsf{m}_i(\vec{T_i}).S_i,\ ⏲.S'$ (by (4), T-&) (6)

$\Gamma_0 + s[\boldsymbol{q}] : S' \vdash_D Q'$ (by (4), (6), T-&) (7)

$\Gamma_\& = \Gamma'_\&, s[\boldsymbol{q}] : S^\&$ and $\mathsf{end}(\Gamma'_\&)$ (by (6) and T-Wkn) (8)

Without loss of generality, assume any end-typed channels in Γ_P are split into Γ_0.

$\Gamma_P \to \Gamma_0 + s[\boldsymbol{q}] : S'$ (by (4), (6), (8), Γ-⏲) (9)

$\Gamma \to \Gamma' = \Gamma_0 + s[\boldsymbol{q}] : S' + \Gamma_d; \Gamma_s$ (by (3), (9), Γ-Cong$_2$) (10)

$\Gamma' \vdash_D P' :: \mathcal{B}$ (by (5), (7), T-$\mathcal{B}$) (11)

$\mathcal{R}$-safe$_!(\Gamma')$ (by **(A2)**, (10), and φ-$\to$) (12)

Case R-Drop:

$P :: \mathcal{B} = P :: \mathcal{B}'', s : \widetilde{\mathcal{M}}, \mathcal{M}'$ (by R-Drop and **(A3)**) (1)

$P' :: \mathcal{B}' = P :: \mathcal{B}'', s : \widetilde{\mathcal{M}}$ (by R-Drop and **(A3)**) (2)

$\Gamma = \Gamma_P \cdot \Gamma_d; \Gamma_s \vdash_D P :: \mathcal{B}$ (by (1), **(A1)**, T-$\mathcal{B}$) (3)

$\Gamma_P \vdash_D P$ (by (1), **(A1)**, T-$\mathcal{B}$) (4)

$\Gamma_d; \Gamma_s \vdash \mathcal{B}'', s : \widetilde{\mathcal{M}}, \mathcal{M}'$ (by (1), (3), T-$\mathcal{B}$) (5)

$\Gamma_0 (\cdot \Gamma'_{d_i})_{i\in 1..n}\ ;\ \Gamma'_s \cdot s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}((T_i)_{i\in 1..n})), \widetilde{M} \vdash \mathcal{B}'', s : \widetilde{\mathcal{M}}, \mathcal{M}'$ (by (5), T-Msg) (6)

where $\Gamma_d; \Gamma_s = \Gamma_0 (\cdot \Gamma'_{d_i})_{i\in 1..n}\ ;\ \Gamma'_s \cdot s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}((T_i)_{i\in 1..n})), \widetilde{M}$ (cont. of (6)) (7)

and $\mathcal{M}' = (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle (d_i)_{i\in 1..n}\rangle)$ (cont. of (7)) (8)

$\Gamma_0; \Gamma'_s + s : \widetilde{M} \vdash \mathcal{B}'', s : \widetilde{\mathcal{M}}$ (by (6) and T-Msg) (9)

$\forall i \in 1..n : \Gamma'_{d_i} \vdash d_i : T_i$ (by (6) and T-Msg) (10)

$\mathsf{gc}((\Sigma_{i\in 1..n}\Gamma'_{d_i})\ ;\ s : (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}((T_i)_{i\in 1..n})))$ (by (10) and Definition 5.7) (11)

$\Gamma_d\ ;\ \Gamma_s \vdash \mathcal{B}'$ (by (7), (9), (11) and T-Wkn$_2$) (12)

$\Gamma \vdash P' :: \mathcal{B}'$ (by (12), (4) and T-$\mathcal{B}$) (13)

Case R-Snd:

$P :: \mathcal{B} = s[\boldsymbol{p}] \oplus [\boldsymbol{q}]\,\mathsf{m}\langle \vec{d} \rangle\ .\ Q :: \mathcal{B}$ (by R-Snd and **(A3)**) (1)

$P' :: \mathcal{B}' = Q :: \mathcal{B} \leftarrow \langle s, (\boldsymbol{p} \triangleright \boldsymbol{q} : \mathsf{m}\langle \vec{d} \rangle)\rangle$ (by R-Snd and **(A3)**) (2)

$\Gamma = \Gamma_P \cdot \Gamma_d; \Gamma_s \vdash P :: \mathcal{B}$ (by (1), (A1) and T-$\mathcal{B}$) (3)
$\Gamma_P = \Gamma_0 \cdot \Gamma_\oplus(\cdot \Gamma_i)_{i \in 1..n} \vdash s[p] \oplus [q]\, \mathsf{m}\langle \vec{d} \rangle \,.\, Q$ (by (1), (3) and T-$\oplus$) (4)
$\Gamma_d; \Gamma_s \vdash \mathcal{B}$ (by (1), (3) and T-$\mathcal{B}$) (5)

Via the type semantics of Figure 6.4, rule Γ-Enq, the output message can be enqueued into the type buffer. This along with the payload types $(\cdot \Gamma_i)_{i \in 1..n}$ type the updated buffer $\mathcal{B}'$ in the process continuation via T-Msg. By T-$\oplus$, the remaining context parts type the process continuation Q. Lastly, safety is preserved in the context reduction by (A2) and φ-$\rightarrow$.

The receive case follows similar reasoning. Further details can be found in the technical report [70, Appendix A, Thm. 1] □

B.3 Session Fidelity

Lemma B.4. Assume $\Gamma \vdash_D P :: \mathcal{B}$ with $\mathsf{one\text{-}role}_!(P)$. Then $P \equiv \big|_{p \in I} P_p$ and $\Gamma = \Gamma_0(\cdot \Gamma'_p, s[p] : U_p)_{p \in I'} \cdot \Gamma_d \,;\, \Gamma_s$, where:

1. $\forall p \in I' : \Gamma'_p, s[p] : U_p \vdash_D P_p$ and $c : U \notin \Gamma'_p$;
2. $\forall q \in I \setminus I' : P_q \equiv \mathbf{0}$;
3. $c : U \notin \Gamma_0$;
4. $\Gamma_d \,;\, \Gamma_s \vdash \mathcal{B}$.

Proof. By rule T-$\mathcal{B}$, item 4. holds. The proof of the remaining items is identical to lemma A.15. □

Lemma B.5. Assume $\Gamma_0, s[p] : U_p \vdash_D P_p$ with $c : U \notin \Gamma_0$ and either P_p only plays role p or $P_p \equiv \mathbf{0}$. Then:

1. if $S_1 | \cdots | S_n \sqsubseteq U_p$, then $P_p \equiv P_1 | \cdots | P_n$ and $\forall i \in 1..n : \Gamma_0, s[p] : S_i \vdash_D P_i$ and P_i only plays role p;
2. if $!\alpha \&_{i \in I} \mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq U_p$, then either:
   (a) $P_p \equiv\, !s[p][\alpha] \&_{i \in I} \mathsf{m}_i(\vec{b}_i).P'_i$; or
   (b) $P_p \equiv \mathrm{X}\langle \vec{V} \rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv\, !s[p][\alpha] \&_{i \in I} \mathsf{m}_i(\vec{b}_i).P'_i$;
3. if $\&_{i \in I} q_i{:}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq U_p$, then either:
   (a) $P_p \equiv s[p] \&_{i \in I} [q_i]\, \mathsf{m}_i(\vec{b}_i).P'_i$; or
   (b) $P_p \equiv \mathrm{X}\langle \vec{V} \rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv s[p] \&_{i \in I} [q_i]\, \mathsf{m}_i(\vec{b}_i).P'_i$;

4. if $!\&_{i\in I}\boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq U_{\boldsymbol{p}}$, then either:
   (a) $P_{\boldsymbol{p}} \equiv !s[\boldsymbol{p}]\&_{i\in I}[\boldsymbol{q_i}]\,\mathsf{m}_i(\vec{b}_i).P'_i$; or
   (b) $P_{\boldsymbol{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv !s[\boldsymbol{p}]\&_{i\in I}[\boldsymbol{q_i}]\,\mathsf{m}_i(\vec{b}_i).P'_i$;
5. if $\&_{i\in I}\boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i, \clock.S' \sqsubseteq U_{\boldsymbol{p}}$, then either:
   (a) $P_{\boldsymbol{p}} \equiv s[\boldsymbol{p}]\&_{i\in I}[\boldsymbol{q_i}]\,\mathsf{m}_i(\vec{b}_i).P'_i, \clock.P''$; or
   (b) $P_{\boldsymbol{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv s[\boldsymbol{p}]\&_{i\in I}[\boldsymbol{q_i}]\,\mathsf{m}_i(\vec{b}_i).P'_i, \clock.P''$;
6. if $\oplus_{i\in I}\boldsymbol{q_i}{:}\mathsf{m}_i(\vec{T}_i).S_i \sqsubseteq U_{\boldsymbol{p}}$, then either:
   (a) $P_{\boldsymbol{p}} \equiv \sum_{i\in I'} s[\boldsymbol{p}] \oplus [\boldsymbol{q_i}]\,\mathsf{m}_i\langle\vec{V}_i\rangle.P'_i$ and $I \supseteq I'$; or
   (b) $P_{\boldsymbol{p}} \equiv \mathrm{X}\langle\vec{V}\rangle$ and $D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and $Q\{\vec{V}/\vec{b}'\} \equiv \sum_{i\in I'} s[\boldsymbol{p}] \oplus [\boldsymbol{q_i}]\,\mathsf{m}_i\langle\vec{V}_i\rangle.P'_i$ and $I \supseteq I'$;
7. if $\mathbf{end} \sqsubseteq U_{\boldsymbol{p}}$, then $P_{\boldsymbol{p}} \equiv \mathbf{0}$.

Proof. Similar to lemma A.16. □

Theorem B.6 (Session Fidelity). Assume $\Gamma \vdash_D P :: \mathcal{B}$ with $\mathcal{R}$-fid$_!(\Gamma)$ and one-role$_!(P)$. Then, $\Gamma\!\rightarrow$ implies either:

1. $\exists \Gamma', P', \mathcal{B}'$ s.t. *(i)* $\Gamma\!\rightarrow\Gamma'$; *(ii)* $P :: \mathcal{B} \rightarrow^*_{D;\mathcal{R}} P' :: \mathcal{B}'$; *(iii)* $\Gamma' \vdash_D P' :: \mathcal{B}'$; and *(iv)* one-role$_!(P')$; or
2. $P \not\rightarrow_{D;\mathcal{R}}$ and $\forall \Gamma'$ s.t. $\Gamma\!\rightarrow\Gamma' : \Gamma \xrightarrow{A} \Gamma' \wedge A = s{:}\boldsymbol{p}\&\boldsymbol{q}{:}\mathsf{m} \wedge \mathsf{m} \in \mathsf{R}$.

Proof. The assumptions are: **(A1)** $\Gamma \vdash_D P :: \mathcal{B}$; **(A2)** $\mathcal{R}$-fid$_!(\Gamma)$; **(A3)** one-role$_!(P)$.

The proof begins by inferring some structure on both the types and processes, utilizing lemma B.4.

$P \equiv \big\vert_{\boldsymbol{p}\in I} P_{\boldsymbol{p}}$ and $\Gamma = \Gamma_0(\cdot\,\Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U_{\boldsymbol{p}})_{\boldsymbol{p}\in I'} \cdot \Gamma_d \,;\, \Gamma_s$	((A1), (A3), lemma B.4)	(1)
$\forall \boldsymbol{p} \in I' : \Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U_{\boldsymbol{p}} \vdash_D P_{\boldsymbol{p}}$ and $c : U \notin \Gamma'_{\boldsymbol{p}}$	((A1), (A3), lemma B.4)	(2)
$\forall \boldsymbol{q} \in I \setminus I' : P_{\boldsymbol{q}} \equiv \mathbf{0}$	((A1), (A3), lemma B.4)	(3)
$c : U \notin \Gamma_0$	((A1), (A3), lemma B.4)	(4)
$\Gamma_d \,;\, \Gamma_s \vdash \mathcal{B}$	((A1), (A3), lemma B.4)	(5)

The proof now continues by induction on $\Gamma\!\rightarrow$, with a further case analysis on the last transition action enabling the context reduction. There are three possible cases, where Γ is enabled with a reduction by either Γ-Enq, Γ-Deq, or Γ-$\clock$.

Case Γ-$\clock$:

$\Gamma'_p, s[p] : U_p \xrightarrow{s:p:\textcircled{1}} \Gamma'_p, s[p] : U'_p$ (by (1), Γ-$\textcircled{1}$) (6)

From this action, the shape of the type can be outlined, since it must contain a branching type with a defined timeout branch.

$\&_{i\in I} q_i{:}\mathsf{m}_i(\vec{T}_i).S_i, \textcircled{1}.S' \mid U''_p \sqsubseteq U_p$ (by (6), Γ-$\textcircled{1}$) (7)
$\Gamma \to \Gamma' = \Gamma_0(\cdot \Gamma'_r, s[r] : U_r)_{r\in I'\setminus\{p\}} \cdot \Gamma'_p, s[p] : U'_p \cdot \Gamma_d \,;\, \Gamma_s$ (by (1), (6), Γ-Cong$_1$) (8)
$S' \mid U''_p \sqsubseteq U'_p$ (by (1), (6), Γ-Cong$_1$) (9)

Using lemma B.5, the process can be inferred to have one of two possible shapes.

Subcase $P_p \equiv s[p]\&_{i\in I}[q_i]\,\mathsf{m}_i(\vec{b}_i).P'_i, \textcircled{1}.P'' \mid P_1 \mid \cdots \mid P_n$:
$P :: \mathcal{B} \to_{D;\mathcal{R}} P' :: \mathcal{B}$ where: (by subcase hyp., R-$\textcircled{1}$) (10)
$P' \equiv P'' \mid P_1 \mid \cdots \mid P_n \mid_{r\in I\setminus\{p\}} P_r$ (cont. of (10)) (11)
$\Gamma'_p, s[p] : U'_p \vdash_D P'' \mid P_1 \mid \cdots \mid P_n$ (by (A1), (7), T-|, T-&) (12)
$\Gamma' \vdash_D P' :: \mathcal{B}$ (by (A1), (8), (11), T-|) (13)
$\mathsf{one\text{-}role}_!(\mid_{r\in I\setminus\{p\}} P_r)$ (by (A3)) (14)
$\mathsf{one\text{-}role}_!(P'' \mid P_1 \mid \cdots \mid P_n)$ ($\mathsf{one\text{-}role}_!(P_p)$ and by lemma B.5 item 1.) (15)
$\mathsf{one\text{-}role}_!(P')$ (by (14), (15)) (16)

Therefore, by (8), (10), (13), (16), the subcase holds.

Subcase $P_p \equiv \mathrm{X}\langle\vec{V}\rangle \mid P_1 \mid \cdots \mid P_n$:
$D(\mathrm{X}) = (\vec{b}', \vec{T}', Q)$ and (by lemma B.5) (17)
$Q\{\vec{V}/\vec{b}'\} \equiv s[p]\&_{i\in I}[q_i]\,\mathsf{m}_i(\vec{b}_i).P'_i, \textcircled{1}.P''$ (cont. of (17)) (18)
$P :: \mathcal{B} \to_{D;\mathcal{R}} \to_{D;\mathcal{R}} P' :: \mathcal{B}$ where: (by subcase hyp., R-X, R-$\textcircled{1}$) (19)
$P' \equiv P'' \mid P_1 \mid \cdots \mid P_n \mid_{r\in I\setminus\{p\}} P_r$ (cont. of (19)) (20)

The remainder of the subcase is similar to the previous.

Case Γ-Enq**:**
$\Gamma'_p, s[p] : U_p \xrightarrow{s:p\oplus q:\mathsf{m}} \Gamma'_p, s[p] : U'_p$ (by (1), Γ-$\textcircled{1}$) (21)

From this action, the shape of the type can be outlined, since it must contain a selection.

$\oplus_{i\in I} q_i{:}\mathsf{m}_i(\vec{T}_i).S_i \mid U''_p \sqsubseteq U_p$ (by (21), Γ-Enq) (22)
$\Gamma \to \Gamma' = \Gamma_0(\cdot \Gamma'_r, s[r] : U_r)_{r\in I'\setminus\{p\}} \cdot \Gamma'_p, s[p] : U'_p \cdot \Gamma_d \,;\, \Gamma_s$ ((1), (21), Γ-Cong$_1$) (23)
$S' \mid U''_p \sqsubseteq U'_p$ ((1), (21), Γ-Cong$_1$) (24)

Using lemma B.5, the process can be inferred to have one of two possible shapes. The case continues similar to the previous. This time, the process either matches the type reduction after two reductions (via R-+ and R-Snd), or after three reductions (via R-X, R-+ and R-Snd).

Case Γ-Deq:

$$\Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U_{\boldsymbol{p}} \xrightarrow{s:\boldsymbol{p}\&\boldsymbol{q}:\mathsf{m}} \Gamma'_{\boldsymbol{p}}, s[\boldsymbol{p}] : U'_{\boldsymbol{p}} \qquad \text{(by (1), Γ-Deq)} \quad (25)$$

From this action, the type could have 4 possible shapes. These are: *(i)* universal receive; *(ii)* replicated receive; *(iii)* reliable receive; *(iv)* receive with timeout branch.

For *(iv)*, the process can always at least mimic the timeout action, as shown in Case Γ-🕐.

For *(iii)*, by **(A2)** (φ-$\mathcal{R}$), the dequeue action is on a reliable message. Therefore, by **(A1)**, it follows that the message is also in the process buffer. The case then follows similar to the previous cases, where the process mimics the type via either R-Rcv, or R-X then R-Rcv.

For *(i)* and *(ii)*, by **(A1)**, the message enabling the dequeue action is either in the process buffer (by T-Msg), or has been dropped from the buffer (by T-Wkn$_2$). If the message is in the buffer, then the proof is similar to that of *(iii)*. If the message is not in the process buffer, then the process cannot mimic the dequeue action. By induction on all the above cases, it can be observed that the only types resulting in processes unable to mimic their actions are replicated types. Thus item *2.* of the theorem is satisfied. □

Bibliography

[1] ADAMEIT, M., PETERS, K., AND NESTMANN, U. Session types for link failures. In *Formal Techniques for Distributed Objects, Components, and Systems, FORTE* (June 2017), A. Bouajjani and A. Silva, Eds., vol. 10321 of *Lecture Notes in Computer Science*, Springer, pp. 1–16.

[2] AKKOYUNLU, E. A., EKANANDHAM, K., AND HUBER, R. V. Some constraints and tradeoffs in the design of network communications. In *Symposium on Operating System Principles, SOSP* (November 1975), J. C. Browne and J. Rodriguez-Rosell, Eds., Association for Computing Machinery, pp. 67–74.

[3] ALMEIDA, B., MORDIDO, A., THIEMANN, P., AND VASCONCELOS, V. T. Polymorphic lambda calculus with context-free session types. *Information and Computation Volume 289*, Part A (2022).

[4] ARANDA, J., GIUSTO, C. D., PALAMIDESSI, C., AND VALENCIA, F. D. On recursion, replication and scope mechanisms in process calculi. In *Formal Methods for Components and Objects, FMCO* (November 2006), F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4709 of *Lecture Notes in Computer Science*, Springer, pp. 185–206.

[5] ARANDA, J., AND VENOLIA, G. The secret life of bugs: Going past the errors and omissions in software repositories. In *Software Engineering* (2009), pp. 298–308.

[6] ARMSTRONG, J. A history of Erlang. In *History of Programming Languages, HOPL* (2007), Association for Computing Machinery, pp. 6:1–6:26.

[7] ARMSTRONG, J. Erlang. *Communications of the ACM 53*, 9 (2010), 68–75.

[8] BAIER, C., AND KATOEN, J.-P. *Principles of model checking*. MIT press, 2008.

[9] BARBER, A., AND PLOTKIN, G. *Dual intuitionistic linear logic*. University of Edinburgh, Department of Computer Science, 1996.

[10] BARTOCCI, E., FALCONE, Y., FRANCALANZA, A., AND REGER, G. Introduction to runtime verification. In *Lectures on Runtime Verification - Introductory and Advanced*

Topics, E. Bartocci and Y. Falcone, Eds., vol. 10457 of *Lecture Notes in Computer Science*. Springer, 2018, pp. 1–33.

[11] BARTOLETTI, M., SCALAS, A., TUOSTO, E., AND ZUNINO, R. Honesty by typing. *Logical Methods in Computer Science 12*, 4 (2016).

[12] BARWELL, A. D., HOU, P., YOSHIDA, N., AND ZHOU, F. Crash-stop failures in asynchronous multiparty session types. *Logical Methods in Computer Science 21*, 2 (2025).

[13] BARWELL, A. D., SCALAS, A., YOSHIDA, N., AND ZHOU, F. Generalised multiparty session types with crash-stop failures. In *Concurrency Theory, CONCUR* (September 2022), B. Klin, S. Lasota, and A. Muscholl, Eds., vol. 243 of *Leibniz International Proceedings in Informatics*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:25.

[14] BETTINI, L., COPPO, M., D'ANTONI, L., LUCA, M. D., DEZANI-CIANCAGLINI, M., AND YOSHIDA, N. Global progress in dynamically interleaved multiparty sessions. In *International Conference on Concurrency Theory, CONCUR* (August 2008), F. van Breugel and M. Chechik, Eds., vol. 5201 of *Lecture Notes in Computer Science*, Springer, pp. 418–433.

[15] BJESSE, P. What is formal verification? *SIGDA Newsletter 35*, 24 (December 2005).

[16] BOWMAN, H., AND DERRICK, J. *Formal methods for distributed processing: a survey of object-oriented approaches*. Cambridge University Press, 2001.

[17] BRAND, J. *Observations on Popular Antiquities*. 1780.

[18] BUSI, N., GABBRIELLI, M., AND ZAVATTARO, G. Replication vs. recursive definitions in channel based calculi. In *International Conference on Automata, Languages and Programming, ICALP* (June 2003), Springer, pp. 133–144.

[19] CAIRES, L., AND PÉREZ, J. A. Linearity, control effects, and behavioral types. In *European Symposium on Programming, ESOP* (April 2017), H. Yang, Ed., vol. 10201 of *Lecture Notes in Computer Science*, Springer, pp. 229–259.

[20] CAIRES, L., AND PFENNING, F. Session types as intuitionistic linear propositions. In *Concurrency Theory, CONCUR* (August 2010), P. Gastin and F. Laroussinie, Eds., vol. 6269 of *Lecture Notes in Computer Science*, Springer, pp. 222–236.

[21] CAIRES, L., PFENNING, F., AND TONINHO, B. Linear logic propositions as session types. *Mathematical Structures in Computer Science 26*, 3 (2016), 367–423.

[22] CAPECCHI, S., GIACHINO, E., AND YOSHIDA, N. Global escape in multiparty sessions. *Mathematical Structures in Computer Science 26*, 2 (2016), 156–205.

[23] CESARINI, F., AND THOMPSON, S. *Erlang programming: a concurrent approach to software development*. O'Reilly Media, Inc., 2009.

[24] CHEN, B.-S., AND YEH, R. Formal specification and verification of distributed systems. *IEEE Transactions on Software Engineering SE-9*, 6 (1983), 710–722.

[25] CHEN, T., VIERING, M., BEJLERI, A., ZIAREK, L., AND EUGSTER, P. A type theory for robust failure handling in distributed systems. In *Formal Techniques for Distributed Objects, Components, and Systems FORTE* (June 2016), E. Albert and I. Lanese, Eds., vol. 9688 of *Lecture Notes in Computer Science*, Springer, pp. 96–113.

[26] CHLIPALA, A. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

[27] CHURCH, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58*, 2 (1936), 345–363.

[28] CHURCH, A. A formulation of the simple theory of types. *Journal of Symbolic Logic 5*, 2 (1940), 56–68.

[29] COPPO, M., DEZANI-CIANCAGLINI, M., PADOVANI, L., AND YOSHIDA, N. A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming* (June 2015), M. Bernardo and E. B. Johnsen, Eds., vol. 9104 of *Lecture Notes in Computer Science*, Springer, pp. 146–178.

[30] COPPO, M., DEZANI-CIANCAGLINI, M., YOSHIDA, N., AND PADOVANI, L. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science 26*, 2 (2016), 238–302.

[31] DARDHA, O., AND GAY, S. J. A new linear logic for deadlock-free session-typed processes. In *Foundations of Software Science and Computation Structures, FOSSACS* (April 2018), C. Baier and U. D. Lago, Eds., vol. 10803 of *Lecture Notes in Computer Science*, Springer, pp. 91–109.

[32] DARDHA, O., GIACHINO, E., AND SANGIORGI, D. Session types revisited. *Information and Computation 256* (2017), 253–286.

[33] DENIÉLOU, P., YOSHIDA, N., BEJLERI, A., AND HU, R. Parameterised multiparty session types. *Logical Methods in Computer Science 8*, 4 (2012).

[34] DEZANI-CIANCAGLINI, M., MOSTROUS, D., YOSHIDA, N., AND DROSSOPOULOU, S. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming, ECOOP* (July 2006), vol. 4067 of *Lecture Notes in Computer Science*, Springer, pp. 328–352.

[35] DIGITAL EQUIPMENT CORPORATION, INTEL CORPORATION, AND XEROX CORPORATION. The Ethernet: A local area network. *Data Link Layer and Physical Layer Specifications Version 1.0* (1980).

[36] EHRHARD, T. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science 28*, 7 (2018), 995–1060.

[37] FOWLER, S., KOKKE, W., DARDHA, O., LINDLEY, S., AND MORRIS, J. G. Separating sessions smoothly. *Logical Methods in Computer Science 19*, 3 (2023).

[38] FOWLER, S., LINDLEY, S., MORRIS, J. G., AND DECOVA, S. Exceptional asynchronous session types: session types without tiers. *Principles on Programming Languages, POPL 3* (January 2019), 28:1–28:29.

[39] FRANCALANZA, A., ACETO, L., ACHILLEOS, A., ATTARD, D. P., CASSAR, I., MONICA, D. D., AND INGÓLFSDÓTTIR, A. A foundation for runtime monitoring. In *Runtime Verification, RV* (September 2017), S. K. Lahiri and G. Reger, Eds., vol. 10548 of *Lecture Notes in Computer Science*, Springer, pp. 8–29.

[40] FRANCALANZA, A., AND HENNESSY, M. A theory for observational fault tolerance. *Journal of Logic and Algebraic Programming 73*, 1-2 (September 2007), 22–50.

[41] FRANCALANZA, A., PÉREZ, J. A., AND SÁNCHEZ, C. Runtime verification for decentralised and distributed systems. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, E. Bartocci and Y. Falcone, Eds., vol. 10457 of *Lecture Notes in Computer Science*. Springer, 2018, pp. 176–210.

[42] FRANCALANZA, A., AND TABONE, G. Elixirst: A session-based type system for Elixir modules. *Journal of Logical and Algebraic Methods in Programming 135* (October 2023).

[43] FRANKAU, S., SPINELLIS, D., NASSUPHIS, N., AND BURGARD, C. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming 19*, 1 (January 2009), 27–45.

[44] FREDERICK, R., CASNER, S. L., JACOBSON, V., AND SCHULZRINNE, H. RTP: A transport protocol for real-time applications. RFC 1889, January 1996.

[45] GAY, S. J., AND HOLE, M. Subtyping for session types in the pi calculus. *Acta Informatica 42*, 2-3 (2005), 191–225.

[46] GAY, S. J., AND VASCONCELOS, V. T. *Session Types*. Cambridge University Press, 2025.

[47] GIRARD, J.-Y. Linear logic. *Theoretical Computer Science 50*, 1 (1987), 1–101.

[48] GIUNTI, M., AND YOSHIDA, N. Iso-recursive multiparty sessions and their automated verification. In *European Symposium on Programming, ESOP* (May 2025), V. Vafeiadis, Ed., vol. 15694 of *Lecture Notes in Computer Science*, Springer, pp. 349–378.

[49] GROOTE, J. F., AND MOUSAVI, M. R. *Modeling and analysis of communicating systems*. MIT press, 2014.

[50] GUNAWI, H. S., DO, T., LAKSONO, A., HAO, T. M., LUKMAN, J., AND SUMINTO, R. What bugs live in the cloud. *A Study of 3000* (2014), 289–301.

[51] HARVEY, P., FOWLER, S., DARDHA, O., AND GAY, S. J. Multiparty session types for safe runtime adaptation in an actor language. In *European Conference on Object-Oriented Programming, ECOOP* (July 2021), A. Møller and M. Sridharan, Eds., vol. 194 of *Leibniz International Proceedings in Informatics*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 10:1–10:30.

[52] HINDEN, B. Reliable Data Protocol. RFC 908, July 1984.

[53] HONDA, K. Types for dyadic interaction. In *Conference on Concurrency Theory, CONCUR* (August 1993), E. Best, Ed., vol. 715 of *Lecture Notes in Computer Science*, Springer, pp. 509–523.

[54] HONDA, K., MUKHAMEDOV, A., BROWN, G., CHEN, T., AND YOSHIDA, N. Scribbling interactions with a formal foundation. In *International Conference on Distributed Computing and Internet Technology, ICDCIT* (February 2011), R. Natarajan and A. K. Ojo, Eds., vol. 6536 of *Lecture Notes in Computer Science*, Springer, pp. 55–75.

[55] HONDA, K., VASCONCELOS, V. T., AND KUBO, M. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming, ESOP* (1998), vol. 1381 of *Lecture Notes in Computer Science*, Springer, pp. 122–138.

[56] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty asynchronous session types. In *Principles of Programming Languages, POPL* (January 2008), G. C. Necula and P. Wadler, Eds., Association for Computing Machinery, pp. 273–284.

[57] HONDA, K., YOSHIDA, N., AND CARBONE, M. Multiparty asynchronous session types. *Journal of the ACM 63*, 1 (2016), 9:1–9:67.

[58] JONES, S. P. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[59] KOBAYASHI, N., PIERCE, B. C., AND TURNER, D. N. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems 21*, 5 (September 1999), 914–947.

[60] KOKKE, W., AND DARDHA, O. Deadlock-free session types in linear Haskell. In *International Symposium on Haskell* (August 2021), J. Hage, Ed., Association for Computing Machinery, pp. 1–13.

[61] KOKKE, W., MONTESI, F., AND PERESSOTTI, M. Better late than never: a fully-abstract semantics for classical processes. *Principles on Programming Languages, POPL 3* (2019), 24:1–24:29.

[62] KOKKE, W., MORRIS, J. G., AND WADLER, P. Towards races in linear logic. *Logical Methods in Computer Science 16*, 4 (2020).

[63] KOUZAPAS, D., DARDHA, O., PERERA, R., AND GAY, S. J. Typechecking protocols with Mungo and StMungo. In *International Symposium on Principles and Practice of Declarative Programming* (September 2016), J. Cheney and G. Vidal, Eds., Association for Computing Machinery, pp. 146–159.

[64] KOZEN, D. Results on the propositional μ-calculus. In *Automata, Languages and Programming* (1982), M. Nielsen and E. M. Schmidt, Eds., Springer Berlin Heidelberg, pp. 348–359.

[65] LAGAILLARDIE, N., NEYKOVA, R., AND YOSHIDA, N. Stay safe under panic: Affine Rust programming with multiparty session types. In *European Conference on Object-Oriented Programming, ECOOP* (June 2022), vol. 222 of *Leibniz International Proceedings in Informatics*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 4:1–4:29.

[66] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[67] LANGE, J., NG, N., TONINHO, B., AND YOSHIDA, N. Fencing off Go: liveness and safety for channel-based programming. In *Principles of Programming Languages, POPL* (January 2017), Association for Computing Machinery, pp. 748–761.

[68] LAPRIE, J.-C. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15 10*, 2 (1985), 124.

[69] LE BRUN, M. A., AND DARDHA, O. Magπ: Types for failure-prone communication. In *European Symposium on Programming, ESOP* (April 2023), T. Wies, Ed., vol. 13990 of *Lecture Notes in Computer Science*, Springer, pp. 363–391.

[70] LE BRUN, M. A., AND DARDHA, O. Magπ: Types for failure-prone communication. Tech. rep., 2023. https://arxiv.org/abs/2301.10827.

[71] LE BRUN, M. A., AND DARDHA, O. Magπ!: The role of replication in typing failure-prone communication. In *Formal Techniques for Distributed Objects, Components, and Systems FORTE* (June 2024), V. Castiglioni and A. Francalanza, Eds., vol. 14678 of *Lecture Notes in Computer Science*, Springer, pp. 99–117.

[72] LE BRUN, M. A., FOWLER, S., AND DARDHA, O. Multiparty session types with a bang! In *European Symposium on Programming, ESOP* (May 2025), V. Vafeiadis, Ed., vol. 15695 of *Lecture Notes in Computer Science*, Springer, pp. 125–153.

[73] LE LANN, G. Distributed systems—Towards a formal approach. In *IFIP congress* (1977), vol. 7, pp. 155–160.

[74] LOGAN, M., MERRITT, E., AND CARLSSON, R. *Erlang and OTP in Action*, 1st ed. Manning Publications Co., USA, 2010.

[75] MAJUMDAR, R., MUKUND, M., STUTZ, F., AND ZUFFEREY, D. Generalising projection in asynchronous multiparty session types. In *International Conference on Concurrency Theory, CONCUR* (August 2021), S. Haddad and D. Varacca, Eds., vol. 203 of *Leibniz International Proceedings in Informatics*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:24.

[76] MARSHALL, D., AND ORCHARD, D. Replicate, reuse, repeat: Capturing non-linear communication via session types and graded modal types. In *International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES* (April 2022), M. Carbone and R. Neykova, Eds., vol. 356 of *Electronic Proceedings in Theoretical Computer Science*, pp. 1–11.

[77] MILNER, R. *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[78] MILNER, R. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[79] MILNER, R. Functions as processes. *Mathematical Structures in Computer Science 2*, 2 (1992), 119–141.

[80] MILNER, R. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[81] MILNER, R., PARROW, J., AND WALKER, D. A calculus of mobile processes, I. *Information and computation 100*, 1 (1992), 1–40.

[82] MILNER, R., PARROW, J., AND WALKER, D. A calculus of mobile processes, II. *Information and computation 100*, 1 (1992), 41–77.

[83] MOCKAPETRIS, P. Domain names - implementation and specification. RFC 1035, November 1987.

[84] MOSTROUS, D., AND VASCONCELOS, V. T. Affine sessions. *Logical Methods in Computer Science 14*, 4 (2018).

[85] ODERSKY, M., SPOON, L., AND VENNERS, B. *Programming in Scala*. Artima Inc, 2008.

[86] ONGARO, D., AND OUSTERHOUT, J. K. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, ATC* (June 2014), G. Gibson and N. Zeldovich, Eds., USENIX Association, pp. 305–319.

[87] ORCHARD, D., AND YOSHIDA, N. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools* (2017), 219.

[88] PADOVANI, L. On the fair termination of client-server sessions. In *28th International Conference on Types for Proofs and Programs, TYPES* (June 2022), D. Kesner and P. Pédrot, Eds., vol. 269 of *Leibniz International Proceedings in Informatics*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 5:1–5:21.

[89] PETERS, K., NESTMANN, U., AND WAGNER, C. Fault-tolerant multiparty session types. In *Formal Techniques for Distributed Objects, Components, and Systems FORTE* (June 2022), M. R. Mousavi and A. Philippou, Eds., vol. 13273 of *Lecture Notes in Computer Science*, Springer, pp. 93–113.

[90] PETERS, K., AND YOSHIDA, N. Separation and encodability in mixed choice multiparty sessions. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS* (2024), Association for Computing Machinery.

[91] PIERCE, B. C. *Types and programming languages*. MIT Press, 2002.

[92] POÇAS, D., COSTA, D., MORDIDO, A., AND VASCONCELOS, V. T. System f^{μ}_{ω} with context-free session types. In *European Symposium on Programming, ESOP* (April 2023), T. Wies, Ed., vol. 13990 of *Lecture Notes in Computer Science*, Springer, pp. 392–420.

[93] POSTEL, J. User Datagram Protocol. RFC 768, Aug. 1980.

[94] POSTEL, J. Internet Control Message Protocol. RFC 777, Apr. 1981.

[95] POSTEL, J. Internet Protocol. RFC 791, September 1981.

[96] POSTEL, J. Transmission Control Protocol. RFC 793, Sept. 1981.

[97] QIAN, Z., KAVVOS, G. A., AND BIRKEDAL, L. Client-server sessions in linear logic. *Proceedings of the ACM on Programming Languages 5*, ICFP (2021), 1–31.

[98] ROCHA, P., AND CAIRES, L. Safe session-based concurrency with shared linear state. In *European Symposium on Programming, ESOP* (2023), vol. 13990 of *Lecture Notes in Computer Science*, Springer, pp. 421–450.

[99] SANGIORGI, D., AND WALKER, D. *The Pi-Calculus — a theory of mobile processes*. Cambridge University Press, 2001.

[100] SCALAS, A., DARDHA, O., HU, R., AND YOSHIDA, N. A linear decomposition of multiparty sessions for safe distributed programming. In *European Conference on Object-Oriented Programming, ECOOP* (June 2017), P. Müller, Ed., vol. 74 of *Leibniz International Proceedings in Informatics*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24:1–24:31.

[101] SCALAS, A., DARDHA, O., HU, R., AND YOSHIDA, N. A linear decomposition of multiparty sessions for safe distributed programming (artifact). *Dagstuhl Artifacts Series 3*, 2 (2017), 03:1–03:2.

[102] SCALAS, A., AND YOSHIDA, N. Less is more: multiparty session types revisited. Tech. Rep. 6, Imperial College London, 2018.

[103] SCALAS, A., AND YOSHIDA, N. Less is more: multiparty session types revisited. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 30:1–30:29.

[104] SINHA, A. Client-server computing. *Commun. ACM 35*, 7 (July 1992), 77–98.

[105] SISTLA, A. P. *Theoretical issues in the design and verification of distributed systems*. Harvard University, 1983.

[106] STUTZ, F. *Implementability of Asynchronous Communication Protocols - The Power of Choice*. PhD thesis, Kaiserslautern University of Technology, Germany, 2024.

[107] TANENBAUM, A. S. *Computer networks*. Pearson Education India, 2003.

[108] THIEMANN, P., AND VASCONCELOS, V. T. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP* (September 2016), J. Garrigue, G. Keller, and E. Sumii, Eds., Association for Computing Machinery, pp. 462–475.

[109] THOMAS, D. Programming Elixir 1.6: Functional |> Concurrent |> Pragmatic |> Fun.

[110] TONINHO, B., AND YOSHIDA, N. Interconnectability of session-based logical processes. *ACM Transactions on Programming Languages and Systems 40*, 4 (2018), 17:1–17:42.

[111] UDOMSRIRUNGRUANG, T., AND YOSHIDA, N. Top-down or bottom-up? Complexity analyses of synchronous multiparty session types. *Proceedings of the ACM on Programming Languages 9*, POPL (2025), 1040–1071.

[112] UMAR, A., AND FRASER, C. *Distributed computing: a practical synthesis of networks, client-server systems, distributed applications, and open systems*. Prentice-Hall, Inc., 1993.

[113] VASCONCELOS, V. T. Fundamentals of session types. In *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM* (June 2009), M. Bernardo, L. Padovani, and G. Zavattaro, Eds., vol. 5569 of *Lecture Notes in Computer Science*, Springer, pp. 158–186.

[114] VASCONCELOS, V. T. Fundamentals of session types. *Information and Computation 217* (2012), 52–70.

[115] VIERING, M., CHEN, T., EUGSTER, P., HU, R., AND ZIAREK, L. A typing discipline for statically verified crash failure handling in distributed systems. In *European Symposium on Programming, ESOP* (April 2018), A. Ahmed, Ed., vol. 10801 of *Lecture Notes in Computer Science*, Springer, pp. 799–826.

[116] VIERING, M., HU, R., EUGSTER, P., AND ZIAREK, L. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proceedings of the ACM on Programming Languages 5*, OOPSLA (2021), 1–30.

[117] WADLER, P. Propositions as sessions. *Journal of Functional Programming 24*, 2-3 (2014), 384–418.

[118] YOSHIDA, N., AND HOU, P. Less is more revisited: Association with global multiparty session types. In *The Practice of Formal Methods: Essays in Honour of Cliff Jones, Part II*, A. Cavalcanti and J. Baxter, Eds., vol. 14781 of *Lecture Notes in Computer Science*. Springer, 2024, pp. 268–291.

[119] YOSHIDA, N., HU, R., NEYKOVA, R., AND NG, N. The Scribble protocol language. In *Trustworthy Global Computing, TGC* (August 2013), M. Abadi and A. Lluch-Lafuente, Eds., vol. 8358 of *Lecture Notes in Computer Science*, Springer, pp. 22–41.

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

[121] ZHIVICH, M., AND CUNNINGHAM, R. K. The real cost of software errors. *IEEE Security & Privacy 7*, 2 (2009), 87–90.