Typed Concurrent Functional Programming with Channels, Actors, and Sessions

Simon Fowler

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2019

Abstract

The age of writing single-threaded applications is over. To develop scalable applications, developers must make use of concurrency and parallelism. Nonetheless, introducing concurrency and parallelism is difficult: naïvely implemented, concurrent code is prone to issues such as race conditions and deadlocks. Moving to the distributed setting introduces yet more issues, in particular the possibility of failure.

To cope with many of the problems of concurrent programming, language designers have proposed a class of programming languages known as *communication-centric* programming languages, which provide lightweight processes which do not share memory, but instead communicate using explicit message passing. The focus of this thesis is on *typed* communication-centric functional programming languages, using type systems to provide static guarantees about the runtime behaviour of concurrent programs. We investigate two strands of work: the relationship between typed channel- and actor-based languages, and the integration of asynchrony, exception handling, and session types in a functional programming language.

In the first strand, we investigate two particular subclasses of communication-centric languages: channel-based languages such as Go, and languages based on the actor model, such as Erlang. We distil the essence of the languages into two concurrent λ-calculi: λ_{ch} for simply-typed channels, and λ_{act} for simply-typed actors, and provide type- and semantics-preserving translations between them. In doing so, we clear up confusion between the two models, give theoretical foundations for recent implementations of type-parameterised actors, and also provide a theoretical grounding for frameworks which emulate actors in channel-based languages. Furthermore, by extending the core calculi, we note that actor synchronisation drastically simplifies the translation from channels into actors, and show that Erlang's selective receive mechanism can be implemented without specialised constructs.

In the second strand, we integrate session types, asynchrony, and exception handling in a functional programming language. Session types are a behavioural type system for communication channel endpoints, allowing conformance to protocols to be checked statically. We provide the first integration of exception handling and asynchronous session types in a core functional language, *Exceptional GV*, and prove that it satisfies preservation, global progress, and that it is confluent and terminating. We demonstrate the practical applicability of the approach by extending the Links tierless web programming language with exception handling, in turn providing the first implementation of exception handling in the presence of session types in a functional language. As a result, we show the first application of session types to web programming, providing examples including a two-factor authentication workflow and a chat application.

Lay Summary

Current mainstream programming languages are ill-suited to writing concurrent applications, as they require developers to reason about how multiple *threads* (concurrent executions of a piece of code) interact when using shared resources. A canonical example of an error in concurrent code is when considering a bank account. Say the bank account initially stands at £100, and thread 1 tries to deposit £60 and thread 2 tries to withdraw £50. We would expect the final result to be £110. However, if both threads read £100 at the same time, and then perform the operation, then the result could be either £160 (if the owner of the account is lucky and thread 1 writes last, overwriting the withdrawal), or £50 (if the owner of the account is unlucky and thread 2 writes last, overwriting the deposit).

The above example details a *race condition*. Race conditions can be fixed by allowing only one thread to access a shared resource at a time when the resource is being updated, which is accomplished through a mechanism called *locks*. However, locks have been shown to be difficult to compose and reason about. In turn, this can lead to *deadlocks*, where a process waits on a lock which will never be released.

We instead consider *message passing* concurrency, where lightweight processes do not share resources with each other directly, but instead communicate explicitly using message passing. Moving to the message passing paradigm is not a silver bullet, however: developers must take care not to send messages which cannot be handled by a remote process, and to not wait for a message which will never arrive. In short, for message passing applications to function as intended, developers must respect *informally-specified* protocols.

In this thesis, we apply the notion of a *data type* to concurrent, message-passing, functional programming languages. The use of data types in programming languages gives lightweight guarantees about program correctness before a program is run: as an example, trying to multiply 5 (of type Int) by true (of type Bool) will result in a compile-time error.

In the first part of the thesis, we investigate two classes of typed concurrent programming languages: channel-based languages and actor-based languages, and distil them to core mathematical models. We show that the channel-based language can emulate the actor-based language, and vice-versa. In doing so, we clear up confusion between the two models, and explicitly detail the difficulties with adding types to actors.

In the second part of the thesis, we investigate the integration of *session types* and exception handling. Session types are a more powerful type system and allow *protocols* to be encoded in types. However, most current designs and implementations of languages with session types do not take into account failure or exception handling, which limits their practical applicability. We design a core language which smoothly integrates session types and exceptions, and show the practical applicability of this approach by implementing it in the Links web programming language.

Acknowledgements

First, I would like to thank my supervisor, Sam Lindley. I have been immensely fortunate to have worked with Sam. Sam has been there every step of the way, discussing all of my ideas with enthusiasm, and the view to making my time as a PhD student a valuable and enjoyable experience. He has gone above and beyond in shaping my development as a researcher.

I am grateful to Philip Wadler, my second supervisor, for always showing interest in my work, and finding time to provide detailed feedback. Phil's insights have greatly simplified ideas and formalisms, and his lessons on writing and presenting will last me a lifetime.

Thanks also to the inhabitants of Office 5.28, in particular Brian Campbell, Garrett Morris, and James McKinna, who have each contributed to my understanding of programming languages, and more generally have made my time at Edinburgh just that bit more fun. James Cheney has provided me with many opportunities throughout my PhD, and beyond.

Thanks to all of the other Edinburgh-PL students past and present: Danel Ahman, Frank Emrich, Stefan Fehrenbach, Weili Fu, Daniel Hillerström, Rudi Horn, Wen Kokke, Craig McLaughlin, Shayan Najd, Jack Williams, Thomas Wright, and Jakub Zalewski for all of the discussions and cameraderie throughout. You all made the office a fun place to be. Thanks also to Office 5.32's advice-giver-in-chief Alyssa Alcorn, especially for the teaching and viva tips.

My time as a PhD student has been greatly enriched by being a member of the Centre for Doctoral Training in Pervasive Parallelism: I have met many wonderful people and learnt many new topics directly as a result of the CDT.

My journey into research started as an undergraduate at the University of St Andrews. I am grateful to the University of St Andrews Functional Programming group, especially Philip Hölzenspies and Edwin Brad(y), who supervised my two research internships and introduced me to the wider programming languages research community. I often wonder how different my life would have been had I not met Philip at the coffee machine that one afternoon.

I am very grateful to my examiners, Peter Thiemann and Ian Stark, for agreeing to examine this thesis. As well as making numerous helpful comments, they strove to make the viva a stimulating and friendly experience. I am also grateful to all anonymous reviewers who have reviewed the submissions stemming from this thesis. Special thanks to a PLDI'18 reviewer who went far beyond the call of duty and whose suggestions led to significant simplifications.

My parents, Clare and Andy Fowler; my brother, Peter Fowler (and our cat, Tommy Fowler) have been an endless source of support and encouragement. I am truly lucky to have such a loving family. Finally, I would like to thank Maria McParland (and our dog, Tara McParland) for showing me that there is so much more to life than just concurrent λ-calculi.

My studies were supported by the EPSRC Centre for Doctoral Training in Pervasive Parallelism (EP/L01503X/1). COST Action IC1201 (BETTY) supported my attendance at the BETTY Summer Schools in 2014 and 2016.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapters 4–6 describe an extended version of material previously published in the proceedings of ECOOP'17, co-authored with Sam Lindley and Philip Wadler.

- Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In Peter Müller, editor, *ECOOP*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:28, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.ECOOP.2017.11

Chapters 9 and 10 describe an extended version of material to appear in Proceedings of the ACM on Programming Languages (PACMPL) issue POPL, co-authored with Sam Lindley, J. Garrett Morris, and Sára Decova.

- Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL): 28:1–28:29, Jan 2019. doi: 10.1145/3290341

I declare that I was primary author on both publications.

(*Simon Fowler*)

Table of Contents

Chapter 1

Introduction

Although the age of single-threaded applications is long over, writing concurrent code remains a challenge. Writers of sequential code have it comparatively easily: while they must concern themselves with the correctness and efficiency of algorithms, and the memory-safety of applications, developers need not consider how different threads of programs interact, or the manner in which threads communicate.

One way of writing concurrent code involves *shared memory*, where different threads have access to shared state. Safe access to shared memory requires synchronisation, normally in the form of locks which ensure the memory is only written to when a thread has exclusive access. Shared memory approaches are common in practice, being the preferred method of co-ordination in languages such as Java, C++, and C. Unfortunately, lock-based approaches are difficult to debug and reason about, in particular because locks are inherently non-compositional [115]. While much research concentrates on the safety of shared memory concurrency, we take a different approach.

Communication-centric programming languages. In this thesis, we instead turn our attention to a second approach, where threads have *share-nothing* semantics, but instead communicate explicitly via message passing. To this end, languages such as Go [61], Concurrent ML [186], and Erlang [11] form a class of languages which are *communication-centric*, providing language support for lightweight threads and message passing between them.

Nonetheless, communication-centric programming languages are not a silver bullet. Moving to communication-centric setting still requires reasoning about whether a particular process is able to handle a given message; whether a message will ever receive a response; whether cycles in a communication topology will give rise to deadlocks at runtime; and whether programs conform to communication protocols.

Type systems for communication. The notion of a data type in static type systems provides lightweight static guarantees that a program is well-behaved: as Milner's slogan goes, "well-typed programs don't go wrong". Indeed, static type systems provide early feedback on application errors, instead of allowing errors to manifest themselves at runtime, and also help developers better structure their code.

It is natural to consider whether the notion of a data type can be used to help reason about concurrent code. Indeed, channels in Go are parameterised by types: for example, a channel which can send and receive integers can be given the type:

$$\mathsf{Chan}(\mathsf{Int})$$

On the other hand, the notion of a *mailbox* (incoming message queue) in actor-based languages such as Erlang remains stubbornly untyped, allowing any type of message to be sent and received. In turn, messages may never be handled, and thus introduce memory leaks.

Taking the idea of types for channels further, *session types*, as introduced by Honda [94] and later expanded upon by Honda et al. [97], allow channel endpoints to be given more expressive types which capture the *sequencing* and *direction* of messages. As an example, consider extending our channel of integers to an endpoint which sends two integers and receives a boolean:

$$\mathsf{Chan}(!\mathsf{Int}.!\mathsf{Int}.?\mathsf{Bool}.\mathsf{End})$$

Such a type conveys more information than a simply-typed channel, allowing *protocols* to be encoded within a type. The other participant in the session would have a channel endpoint with the *dual* session type:

$$\mathsf{Chan}(?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End})$$

Where the one participant sends, the other receives, and vice-versa. Duality ensures the compatibility of two endpoints. In turn, this more precise type provides the additional static guarantee that an implementation correctly implements the protocol.

Concurrent λ-calculi. In this thesis, we restrict our attention to *functional* programming languages: programming languages where functions are first-class, and which prefer immutable variable bindings to mutable state. Functional programming languages have seen increasing popularity in industry: languages such as Scala [159] have widespread adoption; Haskell [172] has been used to excellent effect in industry [74]; and functional features such as anonymous- and higher-order functions have even made their way to "mainstream" languages such as Java [148] and C++ [199].

Functional programming languages are an ideal candidate for study as they can be reasoned about in the context of the λ-calculus: a core calculus based around function abstraction and

application. *Simply-typed* λ-calculi have a well-behaved core, providing not only preservation (well-typed terms reduce to well-typed terms) and progress (all well-typed programs are either a value or can reduce further) guarantees, but also strong properties such as determinism and termination.

Concurrent λ-calculi, in the style of Niehren et al. [156], describe the concurrent behaviour of functional programs by adding terms for communication and concurrency to the language, and adding a language of *configurations* to describe the concurrent behaviour. We make the (admittedly opinionated) decision to use concurrent λ-calculi throughout this thesis, as opposed to process calculi such as the π-calculus or CCS. The reasons for this are twofold. First, we are interested in the design of programming languages, and there is often not a direct connection between a feature in a process calculus and a corresponding feature in a programming language: what may be specified in a model of the *state* of a system may not necessarily reflect the static *term* evaluated to arrive at such a state. Second, the simply-typed λ-calculus has strong correctness properties, whereas typed process calculi do not have such a well-behaved core. For example, the π-calculus with a simple type system guarantees type preservation, but more sophisticated type systems are required in order to guarantee progress or termination [118]. We can therefore extend the λ-calculus modularly, knowing which properties remain with each extension.

1.1 Research Challenges

We investigate two research questions:

What is the relationship between typed channel- and actor-based programming, and why have typed actor mailboxes seen limited uptake?

Languages such as Go allow processes to communicate over typed channels, whereas actor-based languages associate a *mailbox* with each process, encouraging asynchronous, point-to-point messaging. Whereas typed channels are widespread, typed mailboxes have received more limited attention. The two models are closely related, but there remains confusion between the two. What is the precise relation between the two models in the typed setting, and can this help explain why typed actor mailboxes have seen less adoption?

How can session types be adapted to support exceptions in a functional language where communication is asynchronous?

Session types allow conformance to a protocol to be checked at compile time. To safely integrate session types and programming languages, one needs to ensure that session endpoints are used exactly once, which is typically ensured through

the use of a linear type system. However, linearity is too strong an assumption for realistic programs, which may involve exceptions or disconnection. What are the language constructs required to neatly integrate session types, asynchrony, and exceptions in a functional language?

1.2 Contributions

This thesis makes contributions in two strands of work: type-parameterised channels and actors, and session-typed functional programming languages with exceptions.

Type-parameterised Actors and Channels Part II investigates the relation between type-parameterised channels and actors, by defining two calculi and showing type- and semantics-preserving translations between them. Additionally, we show that the calculi may be modularly extended.

The contributions of Part II are as follows:

1. A calculus λ_{ch} with typed asynchronous channels, and a calculus λ_{act} with type-parameterised actors, based on the λ-calculus extended with communication primitives specialised to each model. We give a type system and operational semantics for each calculus, and precisely characterise the notion of progress that each calculus enjoys (Chapter 4).

2. A simple translation from λ_{act} into λ_{ch}, and a more involved translation from λ_{ch} into λ_{act} with proofs that both translations are type- and semantics-preserving. While the former translation is straightforward, it is *global*, in the sense of Felleisen [66]. While the latter is more involved, it is in fact *local*. Our initial translation from λ_{ch} to λ_{act} requires each channel in the system to have the same type, exemplifying the *type pollution* problem identified by He et al. [86] (Chapter 5).

3. An extension of λ_{act} to support synchronous calls, showing how this can alleviate type pollution and simplify the translation from λ_{ch} into λ_{act} (Chapter 6, §6.2).

4. An extension of λ_{act} to support Erlang-style selective receive, a translation from λ_{act} with selective receive into plain λ_{act}, and correctness proofs for the translations (Chapter 6, §6.3).

Session-Typed Functional Programming Languages with Exceptions In Part III, we provide the first formal integration of asynchronous session types with exceptions in a linear functional language, and show the first implementation integrating session types and

exceptions in a functional programming language by extending the Links functional web programming language. The contributions of Part III are as follows:

1. *Exceptional GV* (Chapter 9), a linear λ calculus extended with asynchronous session-typed channels and exception handling. We prove that the core calculus enjoys preservation, progress, a strong form of confluence called the *diamond property*, and termination.

2. Extensions to EGV supporting exception payloads, unrestricted types, and a more flexible mechanism of session initiation known as *access points* (Chapter 9, §9.4).

3. The design and implementation of an extension of the Links web programming language to support tierless web applications which can communicate using session-typed channels (Chapter 10, §10.5).

4. Client and server backends for Links implementing session typing with exception handling, drawing on connections with effect handlers [178] (Chapter 10, §10.5.2).

5. Example applications using the infrastructure. In particular, we show a two-factor authentication workflow and outline the implementation of a chat server (Chapter 10, §10.4).

1.3 Thesis Structure

The remainder of the thesis proceeds as follows.

Part I introduces the relevant background material. Chapter 2 introduces formal models of concurrency and session types, and surveys the literature. Chapter 3 introduces a core session-typed linear λ-calculus, Synchronous GV (SGV), which is adapted from the GV session-typed functional language originally described by Wadler [213], and originally inspired by the work of Gay and Vasconcelos [77]. Our presentation is inspired directly by the incarnation of GV described by Lindley and Morris [132].

Part II investigates the relationship between typed channels and actors. Chapter 4 informally introduces the two models, distils them down to core calculi, and proves preservation and progress properties. Chapter 5 makes the connection between the two models more explicit by translating λ_{act} into λ_{ch}, and λ_{ch} into λ_{act}. Chapter 6 describes extensions to the core calculi. Chapter 7 concludes and describes related work.

Part III describes the integration of session types, exceptions, and tierless web programming. Chapter 8 introduces Asynchronous GV (AGV), an extension of SGV with asynchronous communication primitives without violating any of SGV's strong metatheory. Chapter 9 introduces Exceptional GV (EGV), an extension of AGV to integrate asynchronous session

types and exception handling, which proves crucial to handling disconnection in the distributed setting. Chapter 10 describes the implementation of distributed session types in Links, in particular describing the implementation of EGV's exception handling construct using a minimal translation to effect handlers [176].

Part IV concludes. Chapter 11 reprises the contributions, and describes directions for future work.

Part I

Background

Chapter 2

Background

In this chapter, we examine the background material required for the remainder of the thesis, and survey the literature on concurrent programming and session types. We begin by introducing formal models of concurrency. We then focus our attention on programming languages, in particular discussing languages integrating communication channels, and languages based on the actor model [87]. Next, we focus further on *session types*, a behavioural type system for communication channel endpoints allowing protocols to be encoded as types for channel endpoints, and enabling conformance to protocols to be checked statically by type checking. We concentrate in particular on the integration of session types and functional programming languages. Finally, we describe *multiparty* session types, which describe a top-down characterisation of protocols allowing more than two participants.

2.1 Process Calculi

Modelling concurrency formally has a host of advantages. Formal modelling allows us to be precise about the concurrent behaviour of a system, and better understand issues such as races and deadlocks which may arise.

Process calculi capture the interplay between communication and concurrency. In particular, processes evaluate in parallel, and reduction rules describe how communication affects the evolution of parallel processes.

Hoare [92] was amongst the first to describe concurrency formally, introducing *communicating sequential processes* or CSP. Processes in CSP communicate by synchronous rendezvous over shared names called *channels*. Milner [142] describes CCS, another algebraic approach to synchronous channel-based communication, and describes its behavioural theory.

A large portion of the literature on session types, and indeed the concurrent portion of the λ-calculi described in the remainder of this thesis, is based on calculi inspired by Milner's π-calculus [144]. The π-calculus makes a major twist to calculi such as CSP and CCS in that it

permits *mobility*—that is, the communication of channel names themselves. The π-calculus is Turing-complete, and can simulate the λ-calculus [143].

2.1.1 Model Checking

Process calculi provide a useful abstraction for *model checking*. Model checkers such as SPIN [93] exhaustively model the state space of a concurrent program, allowing a developer to describe properties written in a specification logic such as linear temporal logic (LTL) and check the properties via a translation [207] to Büchi automata—a generalisation of nondeterministic finite state automata to accept infinite words.

Model checking is a powerful technique, allowing the verification of detailed temporal properties, and has seen widespread use in industry (see, e.g., [41, 80, 194]). Indeed, model checking is a fascinating area of study in its own right. Model checking is however subject to tradeoffs: model checking can only verify properties of a *model* of a system, as opposed to a system's actual implementation. Additionally, model checking is subject to the potential pitfall of *state explosion*: the unfortunate property that the state space of an application grows exponentially with the number of parallel processes, although much progress has been made in ameliorating this problem [42].

In this thesis we choose to focus on more lightweight, type-based verification approaches which provide direct feedback on the implementation of an application. Of course, there is nothing stopping both approaches being used together.

2.2 Communication-centric Programming Languages

Instead of adding concurrency as an afterthought, *communication-centric* programming languages put communication and concurrency at the heart of their design. Communication-centric programming languages such as Erlang [11] and Go [61] provide lightweight threads which are scheduled by a runtime system, and which communicate through the use of message passing.

In this section we discuss two particular types of communication-centric programming languages: languages which communicate over channels, and actor-based languages.

2.2.1 Communication Channels

Process calculi are based around the notion of a *channel*. A channel is a name shared between two processes, allowing them to communicate. Communication may be either *synchronous*, requiring both communicating parties to rendezvous, or *asynchronous*, allowing non-blocking sends.

Several programming languages and libraries have embraced the idea of communication channels. Concurrent ML (CML) [186] is an extension of the ML programming language [145] with first class synchronous channels. CML introduces the notion of a first-class *event* which allows *synchronisation* on a channel to be treated as a first-class concept. A key concept in CML is the idea of *choice*, where a process can choose to nondeterministically synchronise on one of many channels.

The Go programming language [61] is a programming language with lightweight threads (known as "goroutines") and simply-typed channels. Go has found much use in backend systems, for example the Kubernetes container management system [185].

Channels need not be added as first-class entities to languages themselves; as an example, the `core.async` [88] provides a library implementation of asynchronous channels for the Clojure programming language.

2.2.2 The Actor Model

Actor-based languages such as Erlang and Elixir have seen widespread adoption in industry, since asynchronous communication and non-mobile mailboxes are particularly suited to distribution.

Actor-based languages have a formal grounding in the actor model of concurrency. The actor model was originally developed as a formalism for artificial intelligence by Hewitt et al. [87]. In this setting, an actor is an entity which, upon receiving a message, can perform three operations:

1. Spawn a finite set of new actors
2. Send a finite set of messages to other actors
3. Change its behaviour the next time a message is received

Clinger [44] provides a power domain semantics for actor languages. Agha [5] first demonstrates the use of the actor model as a formalism for concurrency and distribution, in particular describing two actor languages, SAL and *Act*. Agha et al. [6] develop the theory of actors in the functional setting by extending the λ-calculus with three constructs: letactor, which creates a new actor; send, which sends a message; and become, which changes the actor's behaviour. The authors provide an operational semantics, prove safety properties, and develop a behavioural theory.

Actors and Reliability. Erlang [11] was originally developed as a real-time programming language for designing scalable and reliable distributed telecoms applications. Erlang provides lightweight, addressable processes with an incoming message queue. A particular strength of

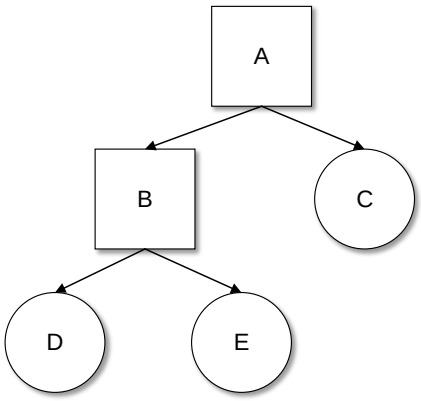


Figure 2.1: Supervision Hierarchy

Erlang is its 'let it crash' ideology [10, 33]: processes are arranged in *supervision hierarchies*, where *supervisor* processes are notified when a worker process crashes, allowing the crashed process to be restarted.

Instead of trying to handle exceptions, processes adopt a 'fail-fast' methodology, where they crash upon encountering unrecoverable errors. Figure 2.1 shows an example supervision hierarchy. Consider the case where process D encounters an unhandled error. In this case, the process will crash, and a notification will be sent to supervisor process B. Process B may then choose to restart D, and also may restart E if the two processes are tightly coupled.

The notion of a supervision hierarchy has also been adopted in other frameworks, such as the Akka [217] framework for Scala.

Variations on a Theme. Many languages refer to themselves as being based on the actor model, but all interpret the term 'actor' in slightly different ways. De Koster et al. [56] describe a taxonomy of actor systems.

Classic Actor Model

The 'classic' actor model is based on the original work of Hewitt et al. [87] and Agha [5], incorporating *behaviours* which govern how an actor responds to an incoming message. The classic actor model uses the spawn, send, and become primitives. Perhaps the most popular implementation is Scala's Akka framework.

Process-based Actors

A process-based actor runs a piece of code from beginning to end, and instead of a become primitive, incorporates a receive construct to retrieve a message from its mailbox. Implementations are typically functional, and examples include Erlang [11] and Elixir [203]. Process-based actors have also been investigated in Scala, without the need for explicit runtime support [84].

Active Objects

Active Objects [128] combine object-oriented programming with the actor model. An active object has an interface, and calling a method on an active object results in a request message being sent to the object. Requests are processed sequentially in the active object's event loop, and results of methods are returned using futures [55]. Examples include Pony [43], ABS [114], and JCoBox [193].

Communicating Event Loops

Communicating Event Loops, as pioneered by the E language [141] have similarities with active objects, but group multiple objects together in *vats* consisting of an object heap, a stack, and an incoming message queue.

Due to their close association with functional programming, we focus primarily on process-based actors.

2.3 Session Types

So far, we have considered only *simple* type systems for channels, where a channel has a single type which does not change throughout its evaluation. For example, a channel in Go might have the type

$$\mathsf{Chan}(\mathsf{Int})$$

meaning that the channel may be used to send and receive integers.

Session types [94, 97] are types for protocols, and describe both the shape and order of messages.

2.3.1 Session Types by Example

We illustrate session types with a basic example of two-factor authentication, as often used for logging on to banking applications. A user inputs their credentials. If the login attempt is from a known device, then the user is authenticated and may proceed to perform privileged actions. If the login attempt is from an unrecognised device, then the user is sent a challenge code. They enter the challenge code into a hardware key which yields a response code. If the user responds with the correct response code, then they are authenticated.

A session type specifies the communication behaviour of one endpoint of a communication channel participating in a dialogue (or *session*) with the other endpoint of the channel. Fig. 2.2 shows the session types of two channel endpoints connecting a client and a server. Fig. 2.2a shows the session type for the server which first receives (?) a pair of a username and password from the client. Next, the server selects ($\oplus$) whether to authenticate the client, issue a challenge, or reject the credentials. If the server decides to issue a challenge, then it sends (!) the challenge

$$\begin{aligned}
&\mathsf{TwoFactorServer} \triangleq \\
&\quad ?(\mathsf{Username}, \mathsf{Password}).\oplus\{ \\
&\qquad \mathsf{Authenticated} : \mathsf{ServerBody}, \\
&\qquad \mathsf{Challenge} : !\mathsf{ChallengeKey}.?\mathsf{Response}. \\
&\qquad\quad \oplus\{\mathsf{Authenticated} : \mathsf{ServerBody}, \\
&\qquad\qquad \mathsf{AccessDenied} : \mathsf{End}\}, \\
&\qquad \mathsf{AccessDenied} : \mathsf{End}\}
\end{aligned}$$

(a) Server Session Type

$$\begin{aligned}
&\mathsf{TwoFactorClient} \triangleq \\
&\quad !(\mathsf{Username}, \mathsf{Password}).\&\{ \\
&\qquad \mathsf{Authenticated} : \mathsf{ClientBody}, \\
&\qquad \mathsf{Challenge} : ?\mathsf{ChallengeKey}.!\mathsf{Response}. \\
&\qquad\quad \&\{\mathsf{Authenticated} : \mathsf{ClientBody}, \\
&\qquad\qquad \mathsf{AccessDenied} : \mathsf{End}\}, \\
&\qquad \mathsf{AccessDenied} : \mathsf{End}\}
\end{aligned}$$

(b) Client Session Type

Figure 2.2: Two-factor Authentication Session Types

string, awaits the response, and either authenticates or rejects the client. The ServerBody type abstracts over the remainder of the interactions, for example making a deposit or withdrawal.

The client implements the *dual* session type, shown in Fig. 2.2b. Whenever the server receives a value, the client sends a value, and vice versa. Whenever the server makes a selection, the client offers a choice (&), and vice versa. This *duality* between client and server ensures that each communication is matched by the other party. We denote duality with an overbar; thus we could define $\mathsf{TwoFactorClient} = \overline{\mathsf{TwoFactorServer}}$ or $\mathsf{TwoFactorServer} = \overline{\mathsf{TwoFactorClient}}$.

Implementing Two-factor Authentication. Let us suppose we have constructs for sending and receiving along, and for closing, an endpoint.

send $M\,N : S$	where M has type A, and N is an endpoint with session type $!A.S$
receive $M : (A \times S)$	where M is an endpoint with session type $?A.S$
close $M : \mathbf{1}$	where M is an endpoint with session type End

Let us also suppose we have constructs for selecting and offering a choice:

select $\ell_j\,M : S_j$	where M is an endpoint with session type $\oplus\{\ell_i : S_i\}_{i\in I}$, and $j \in I$
offer $M\,\{\ell_i(x_i) \mapsto N_i\}_{i\in I} : A$	where M is an endpoint with session type $\&\{\ell_i \mapsto S_i\}_{i\in I}$, each x_i binds an endpoint with session type S_i, and each N_i has type A

Let us write $\mathbf{1}$ for the unit type. We can now write a client implementation:

$$
\begin{array}{l}
\mathsf{twoFactorClient} : (\mathsf{Username} \times \mathsf{Password} \times \mathsf{TwoFactorClient}) \multimap \mathbf{1} \\
\mathsf{twoFactorClient}(\mathit{username}, \mathit{password}, s) \triangleq \\
\quad \textbf{let}\, s = \textbf{send}\, (\mathit{username}, \mathit{password})\, s\, \textbf{in} \\
\quad \textbf{offer}\, s\, \{ \\
\qquad \mathsf{Authenticated}(s) \mapsto \mathsf{clientBody}(s) \\
\qquad \mathsf{Challenge}(s) \mapsto \\
\qquad\quad \textbf{let}\, (\mathit{key}, s) = \textbf{receive}\, s\, \textbf{in} \\
\qquad\quad \textbf{let}\, s = \textbf{send}\, \mathsf{generateResponse}(\mathit{key})\, s\, \textbf{in} \\
\qquad\quad \textbf{offer}\, s\, \{ \\
\qquad\qquad \mathsf{Authenticated}(s) \mapsto \mathsf{clientBody}(s) \\
\qquad\qquad \mathsf{AccessDenied}(s) \mapsto \mathsf{loginFailed}(s) \\
\qquad\quad \} \\
\qquad \mathsf{AccessDenied}(s) \mapsto \mathsf{loginFailed}(s) \\
\quad \}
\end{array}
$$

The twoFactorClient function takes a username, password, and an endpoint s of type TwoFactorClient as its arguments. It sends the username and password along the endpoint, before offering three branches depending on whether the server authenticates the user, sends a two-factor challenge, or rejects the authentication attempt.

In the case that the server authenticates the user, then the program progresses to the main application (denoted here by $\mathsf{clientBody}(s)$). If the server sends a challenge, the client receives the challenge key, and sends the response, calculated by generateResponse. It then offers two branches based on whether the challenge response was successful. If the login attempt fails, then the client evaluates loginFailed, which abstracts over notifying the user of the login failure, and closing the channel.

We can implement a simple server as follows:

$$
\begin{array}{l}
\mathsf{twoFactorServer} : \mathsf{TwoFactorServer} \multimap \mathbf{1} \\
\mathsf{twoFactorServer}(s) \triangleq \\
\quad \textbf{let}\, ((\mathit{username}, \mathit{password}), s) = \textbf{receive}\, s\, \textbf{in} \\
\quad \textbf{if}\, \mathsf{checkDetails}(\mathit{username}, \mathit{password})\, \textbf{then} \\
\qquad \textbf{let}\, s = \textbf{select}\, \mathsf{Authenticated}\, s\, \textbf{in}\, \mathsf{serverBody}(s) \\
\quad \textbf{else} \\
\qquad \textbf{let}\, s = \textbf{select}\, \mathsf{AccessDenied}\, s\, \textbf{in}\, \textbf{close}\, s
\end{array}
$$

The twoFactorServer function takes an endpoint of type TwoFactorServer and receives a username and password, which are checked using the checkDetails function. If the check passes, then the server authenticates the client and proceeds to the application body (denoted

here by serverBody(s)); if not, then the server notifies the client by selecting the AccessDenied branch. Note that this particular server implementation opts never to send a challenge request: whereas the client must implement code for all possible branches, the session type does not oblige the server to select a particular branch.

2.3.2 Linear Type Systems

Simply providing constructs for sending and receiving values, and for selecting and offering choices, is not quite enough to safely implement session types. Consider the following client:

$$
\begin{aligned}
&\mathsf{wrongClient} : \mathsf{TwoFactorClient} \multimap \mathbf{1} \\
&\mathsf{wrongClient}(s) \triangleq \\
&\quad \mathbf{let}\, t = \mathbf{send}\,(\texttt{"Alice"}, \texttt{"hunter2"})\, s\, \mathbf{in} \\
&\quad \mathbf{let}\, t = \mathbf{send}\,(\texttt{"Bob"}, \texttt{"letmein"})\, s\, \mathbf{in}\ \ldots
\end{aligned}
$$

Reuse of s allows a (username, password) pair to be sent along the same endpoint twice, violating the fundamental property of *session fidelity*, which states that in a well-typed program the communication taking place over an endpoint matches its session type.

Linear logic [79] is a logic particularly suited for reasoning about *resources* which may neither be duplicated nor discarded. A line of work started by Lafont [127] applies the ideas from linear logic to programming languages, allowing in particular finer-grained control over resource management. Lafont [127] describes an abstract machine based on linear logic, along with the syntax and typing rules for a linear λ-calculus. The work of Wadler [212] takes the idea of a linear λ calculus further, motivating linear types to allow non-destructive updates of arrays in purely-functional languages, and also showcasing how an unrestricted 'of-course!' modality can enable practical programming.

In order to maintain session fidelity and ensure that all communication actions in a session type occur, session type systems require a similar approach: each endpoint must be used *exactly once*.

2.3.3 Session Types and Process Calculi

Session types were originally introduced in terms of *session calculi*: typed process calculi loosely based on the π-calculus, but including specialised language constructs and typing rules. Such calculi include the original incarnations by Honda [94] and Honda et al. [97], and the revisited calculus by Yoshida and Vasconcelos [218], which fixes problems caused by delegation (sending endpoints in session messages) in the original work. Vasconcelos [209] provides a tutorial introduction to session calculi, providing a core calculus incorporating both linear and unrestricted names.

Session calculi may in fact be expressed in terms of a more canonical base calculus: the linear π-calculus [120]. Kobayashi [118] was first to present a translation from a session calculus into the linear π-calculus; Kobayashi's translation received renewed attention due to the work of Dardha et al. [54], who reformulated the translation in terms of the base calculus of Vasconcelos [209], and proved that the translation was type- and semantics-preserving. They additionally argued for the robustness of the encoding, showing that it accounts for subtyping, recursion, and polymorphism. Key to the translation is the use of continuation-passing style.

2.3.4 Correspondence with Linear Logic

Propositions as types. The Curry-Howard correspondence [103] describes a close correspondence between propositional logic and the simply-typed λ-calculus: propositions correspond to types; proofs correspond to programs; and proof simplification corresponds to evaluation of programs. The relation with logic ensures that the simply-typed λ-calculus enjoys a strong metatheory: in particular, preservation, progress, confluence, and termination. Wadler [214] provides an accessible and comprehensive introduction.

Proofs as processes. A natural question to ask is whether a similar logical correspondence exists for *concurrent* programming. Girard [79] introduces *linear logic*, a substructural logic without the structural logical rules of contraction and weakening. Consequently, propositions may be used *exactly once*, making linear logic useful for reasoning about systems with resources, for example file handles.

Much as the Curry-Howard correspondence provides a solid logical basis for functional programming languages, Abramsky [1] and Bellin and Scott [16] seek to establish linear logic as a basis for concurrent programming languages, by presenting translations from linear logic into the π-calculus. One of the key insights of both works is the interpretation of the logical 'cut' rule as a combination of a name restriction and parallel composition, ensuring a tree-structured and hence acyclic communication topology.

Propositions as sessions. The question of a logical basis for concurrent programming received renewed attention with the landmark paper by Caires and Pfenning [27], providing a logical basis for session-typed programming in the setting of *intuitionistic* linear logic; Caires and Pfenning introduce πDILL, a typed process calculus whose types are precisely the propositions of dual intuitionistic linear logic [14], interpreting propositions as *session types*; proofs as processes; and cut elimination as communication.

Wadler [213] connects the lines of work on session-typed functional programming languages and logically-based process calculi by introducing *CP*, a process calculus based on *classical* linear logic; and *GV*, a session-typed linear λ-calculus. We describe this work further

in §2.3.5.

Pérez et al. [170] describe a theory of linear logical relations for session types. Whereas the primary properties of session-typed process calculi are type preservation and (in some cases) progress and can be proven syntactically in the style of Wright and Felleisen [216], linear logical relations can be used to reason about semantic properties including termination, even in the presence of unrestricted types or shared channels.

Caires and Pérez [26] describe a logically-grounded calculus incorporating session-typed concurrency, nondeterminism, and control operators such as exceptions. Their work is related to the work in Chapter 9, and we discuss it in more detail there.

Deadlock-free cyclic processes. Kobayashi [116, 117, 119] and Kobayashi et al. [121] describe type-based approaches to ensuring lock- and deadlock-freedom in process calculi. These approaches are based on a type system including *priorities* which rule out deadlocking interactions. Padovani [162] integrates priority-based type systems with session type systems; duality (hence compatibility of communication actions) in session types allows a stronger progress property than deadlock-freedom to be proven for typeable session-typed processes. Padovani [163] later investigates priorities in the setting of the linear π-calculus.

Dardha and Gay [53] provide a logical basis for priority-based type systems with session types, who extend CP by replacing the cut rule with two rules, *cycle* and *mix*. Without the use of priorities, the cycle and mix rules immediately introduce the possibility of deadlock. With priorities, however, it becomes possible to eliminate the cycle rule (which is analogous to cut elimination), and thus the work expands the type system of CP to support deadlock-free cyclic processes.

2.3.5 Session Types and Functional Programming Languages

Core functional languages. Session types have traditionally been studied in the setting of typed process calculi. Vasconcelos et al. [208, 210] examine session types in the context of a functional programming language, using a separate *channel environment* to keep track of the state of each session endpoint.

By moving to the setting of a linear functional language, Gay and Vasconcelos [77] provide substantial simplifications, in particular not requiring a separate channel environment or reasoning about channel aliasing. In particular, they provide the first characterisation of asynchronous communication and subtyping in the functional setting, and prove that the size of a buffer is bounded by the session type of its associated endpoint.

Wadler [213] introduces *CP*, a session-typed process calculus based on Classical Linear Logic. Wadler also introduces *GV*, a minimal session-typed linear functional language based on the ideas of Gay and Vasconcelos [77], and presents a type-preserving translation from GV

into CP. Wadler's GV presents the first *deadlock-free* session-typed linear functional language, in particular inherited from CP due to its logical correspondence. More specifically, deadlock-freedom arises due to the interpretation of the logical cut rule in CP as a *combination* of name restriction *and* parallel composition, which ensures by construction the acyclicity of configurations.

Wadler does not present a semantics for GV, instead relying on the translation to CP and the subsequent use of CP's cut reduction semantics. Lindley and Morris [132] describe another variation of GV, and provide it with a small-step operational semantics. This language, also known as GV, forms the basis of the languages we extend throughout this thesis; we provide a detailed introduction in Chapter 3. Lindley and Morris provide a semantics-preserving translation from GV into CP as well as being the first to show a semantics-preserving translation from CP to GV, and show extensions including channel replication and unrestricted types.

Lindley and Morris [133] introduce μCP and μGV, extending CP and GV with structural recursion based on catamorphisms, translating between them, and thus provide a logical basis for recursive session types. A key outcome is a solid semantic grounding for the definition of *duality* in the presence of delegation and recursion, which had proven to be problematic previously [19, 52]. Additionally, they introduce a core version of GV *without* concurrency, showing that concurrency may be simulated through the use of a CPS translation, thereby providing a new proof technique for showing strong normalisation in session-typed functional languages which include unrestricted types.

Lindley and Morris [135] investigate the problem of *practical* programming languages based on GV, introducing FST (System F with Session Types), which extends GV with row typing and polymorphism, and an integration of linear and unrestricted types using subkinding (as pioneered by Mazurak et al. [137]). FST forms the basis of session typing in the Links [46] programming language, the first fully-fledged implementation of session typing in a general-purpose functional programming language. We extend Links' concurrent implementation of session-typed concurrency to a distributed, web-based implementation in Chapter 10.

Lindley and Morris [136] discuss an *asynchronous* semantics for FST and prove its correctness properties. We distil the semantics into an extension of GV in Chapter 8.

Extending πDILL, Toninho et al. [205] describe a language integrating (unrestricted) functional variables and (necessarily linear) session channels, through the use of a *linear contextual monad*. The monadic approach stratifies the language into linear and intuitionistic fragments; in this thesis we tend to work with purely linear calculi for simplicity.

The session typing systems we have described up until now can be described by a regular language. However, such formulations of session types are not sufficient to describe protocols which involve the serialisation of recursive data types, for example, which require session types to be *context-free*. Thiemann and Vasconcelos [202] introduce an account of context-free

session types for an extension of a GV-like core functional language. Key to the formulation is the introduction of a monoidal sequencing operator instead of prefixed send- and receive session types, and the use of polymorphic recursion. A technical challenge is showing that type equivalence is decidable, which the authors prove by developing an LTS semantics for their type system and encoding the system in the context-free BPA [18] calculus, where process equivalence is known to be decidable [40].

Padovani [165] builds upon the work of Thiemann and Vasconcelos [202] by being the first to implement context-free session types, in the setting of the FuSe [164] OCaml implementation of binary sessions. Padovani reformulates context-free sessions to make use of *resumptions* to implement the sequencing required by context-free session types. To safely implement resumptions, session types are ascribed with *identities*. An advantage of Padovani's approach is that type equivalence is no longer needed in the type system of terms, but only to reason about the metatheory, at the cost of additional syntactic markers.

Gradual typing [197] allows the coexistence of statically- and dynamically-typed code, with the goal of gradually adding static typing to the dynamically-typed fragment. A key development of gradual typing is the notion of *blame*, which indicates the part of the program at fault when a runtime type error is found. Wadler and Findler [215] introduce the *blame calculus*, which allows the authors to prove that if a type error arises, then it is due to a failure in the less-typed part of the program. Much like it is desirable to gradually migrate from dynamically-typed to statically-typed code, it is desirable to gradually migrate from code without session types, to code which uses session types. A particular challenge is the gradual handling of linearity. Igarashi et al. [109] propose an extension of GV which supports gradual session types, and prove a blame theorem.

Functional Embeddings. Functional programming languages are useful host languages for *embedding* session types, allowing the benefits of session types to be enjoyed in more mainstream languages. The core challenge in implementing session types is enforcing linear (or even affine) endpoint usage.

Perhaps the most popular language used for embedding session types is Haskell [172], primarily on account of its many type system extensions allowing linearity to be emulated.

Neubauer and Thiemann [149] provide the first such embedding of a session-typed core calculus into Haskell. Pucella and Tov [182] provide an alternative approach, encoding duality using typeclasses with functional dependencies; allowing communication along multiple channels through an explicitly-managed stack; and are the first to use a *parameterised monad* [12] (a monad with pre- and post-conditions) to enforce linearity. By reasoning about their core calculus, the authors prove that even though their implementation makes use of unsafe operations in Haskell, the embedding remains safe. Independently, Sackman and Eisenbach [188]

also implement session types in Haskell using a parameterised monad to allow management of multiple channels.

Lindley and Morris [134] break from the tradition of encoding session types using a parameterised monad, making use of Polakow's embedding of a linear λ-calculus in Haskell [179]. Their approach embeds GV in Haskell, allowing *first-class* channel endpoints as opposed to needing to explicitly manipulate a stack in a parameterised monad, and allow interpretation in multiple underlying monads, including IO and the continuation monad.

Orchard and Yoshida [160] describe type and semantics-preserving translations between a session process calculus FPCF: a variant of Plotkin's PCF [177] extended with a type-and-effect system, and a session calculus. As a consequence, the authors show that it is possible to embed session types through the use of a graded monad.

Orchard and Yoshida [161] provide a comprehensive survey of implementations of session types in Haskell between 2004 and 2017, as well as a comparison of the features supported by each.

Recent work [20] adds first-class linear types to Haskell. Given first-class linearity, it would be interesting to see a more first-class session typing library with first-class channel endpoints, without the additional artefacts of embedding linearity.

Session types have also been implemented in the OCaml programming language [129]. Imai et al. [110] implement session types in OCaml, again using a parameterised monad, and *polarities*, which view a session type from the point of the view of a client or a server. The authors additionally make use of *lenses* to more cleanly manipulate the stack of session channels. Padovani [164] describes FuSe, a tiny library implementation of session types in OCaml. A key design decision is to enforce linearity at *run-time*: although runtime checking of linearity results in fewer static guarantees, it results in a particularly clean library design and implementation, in particular allowing first-class manipulation of channel endpoints.

Scalas and Yoshida [190] also make use of dynamically-checked linearity, as well as the CPS transformation from session-typed process calculi into the linear π-calculus advocated by Dardha et al. [54], to provide a library implementation of session types in Scala. They demonstrate that the CPS translation allows *delegation*—the ability to send channel endpoints as part of a session—in the distributed setting without the need for distributed algorithms [106].

Why a first-class implementation? There are many embeddings of session types in mainstream functional languages. In this thesis, we later consider an extension of the Links programming language with session types and distribution.

A natural question to ask, therefore, is: "why go to the effort of modifying a language to incorporate session types as first-class entities, when you can embed them in a mainstream language?"

The biggest advantage is that we have the power of a native linear type system, and thus do not have to worry about the artefacts arising from encoding linearity, or sacrifice guarantees by checking linearity at runtime. The language is also closer to the core calculi we describe, allowing us to draw side-by-side comparisons between theory and practice. We may additionally add other first-class constructs to the language which require runtime support (for example, exceptions, as we discuss in Chapter 9), which could cause difficulty using an embedding. Finally, we have more control over the error messages that are displayed in the case of an incorrect program, whereas errors in an EDSL may leak details of the embedding.

2.4 Multiparty Session Types

Up until now, we have concentrated on *binary* session types, where a session involves precisely two participants. *Multiparty* session types were introduced by Honda et al. [98, 100] and generalise binary session types by allowing a top-down description of the interactions between multiple participants in a protocol.

The canonical example of a multiparty session type is that of the *two-buyer* protocol, which is intended to be representative of financial protocols. The two-buyer protocol describes the scenario in Figure 2.3:

1. Buyer 1 requests the price of an item from the seller
2. The seller sends the price of the item to Buyer 1 and Buyer 2
3. Buyer 1 sends Buyer 2 the amount that Buyer 2 should pay
4. Buyer 2 can choose to:
    - Accept the offer, at which point it sends a delivery address to the seller and receives a delivery date
    - Reject the offer, and await another offer from Buyer 1
    - End the protocol

The formal presentation of multiparty session types was later substantially simplified by Bettini et al. [22] and Coppo et al. [49]. They consider 'role-indexed channels', which eliminate the need for causality analysis on channels partaking in a multiparty session, and introduce a communication construct which unifies sending a value and making a choice. Additionally, the authors provide the first notion of *global progress* for multiparty sessions, allowing deadlock-freedom and progress in the presence of multiple multiparty sessions; the original work only guaranteed deadlock-freedom for a *single* multiparty session. Global progress is ensured through the use of an additional *interaction typing system*.

Yoshida et al. [219] describe a theory of *parameterised* multiparty session types, which extend global types to parameterise participants by indices. This extension of global types makes it possible to describe protocols from the domain of high-performance computing, such

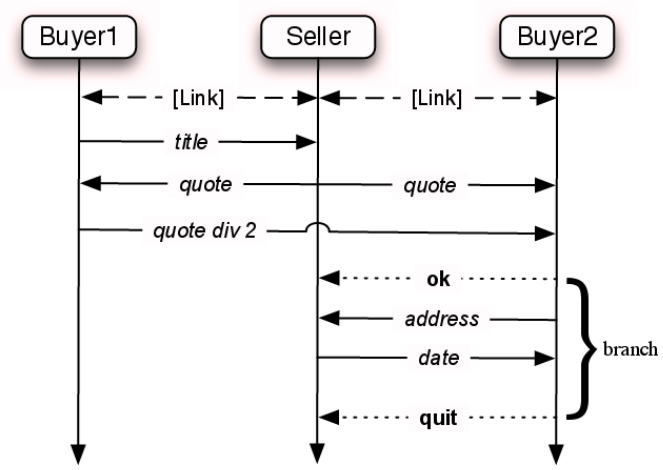


Figure 2.3: Two-buyer protocol

as mesh topologies.

More recent work by Scalas and Yoshida [191] identifies the severe limitations on the expressiveness of previous multiparty session typing systems, in particular due to the conservative requirement of *consistency*. Scalas and Yoshida propose a more general, expressive, and yet simpler system based on generic type systems for the π-calculus [108]. The authors define *safety* protocols such as liveness, and show how protocol conformance can be checked via a translation into formulae which can be checked by the mCRL2 [82] model checker. The system therefore allows more expressive protocols to be described and *checked against implementations*, which is a significant avantage over model checking on its own.

2.4.1 Implementations of Multiparty Session Types

2.4.1.1 The Scribble Protocol Description Language

Scribble [99, 220] is a language-independent protocol description language, based on the theory of multiparty session types. Developers write a global protocol, specifying the interactions between each participant in the system in a top-down manner.

The Scribble tool firstly verifies that the global protocol is well-formed and thus describes a safe protocol; originally verification was achieved using fairly conservative syntactic checks, however recent work [105] takes a more semantic approach through the use of 1-bounded model checking, and is thus less restrictive.

The two-buyer protocol can be described in Scribble as follows:

```
global protocol TwoBuyer(role Buyer1, role Buyer2, role Seller) {
    title(String) from Buyer1 to Seller;
    price(Currency) from Seller to Buyer1, Buyer2;
    rec loop {
        share(Currency) from Buyer1 to Buyer2;
        choice at Buyer2 {
            accept() from Buyer2 to Buyer1;
            deliveryAddress(String) from Buyer2 to Seller;
            deliveryDate(Date) from Seller to Buyer2;
        } or {
            reject() from Buyer2 to Buyer1;
            continue loop;
        } or {
            quit() from Buyer2 to Buyer1, Seller;
        }
    }
}
```

Next, the Scribble tool *projects* the global protocol into *local types*, describing the protocol from the viewpoint of each participant. As an example, the two-buyer protocol projected at Buyer 1 is as follows:

```
local protocol TwoBuyer at Buyer1(role Buyer1, role Buyer2, role Seller)
    {
    title(String) to Seller;
    price(Currency) from Seller;
    rec loop {
        share(Currency) to Buyer2;
        choice at Buyer2 {
            accept(String) from Buyer2;
        } or {
            reject() from Buyer2;
            continue loop;
        } or {
            quit() from Buyer2;
        }
    }
}
```

After local projections have been generated, they may be used to verify conformance to a protocol either statically, or via runtime verification techniques.

2.4.1.2 Static Checking

One of the earliest implementations of statically-checked multiparty session types is Multiparty Session C [154], which implements multiparty session types in C via a lightweight runtime system and a compiler plugin. Multiparty Session C concentrates on bringing the benefits of multiparty session types to the domain of high-performance computing. Later work by Ng et al. [155] uses Scribble to generate MPI backbone code, reducing the amount of boilerplate a

developer of HPC applications must write, and guaranteeing deadlock-freedom.

Typestate [198] is a behavioural typing discipline from the field of object-oriented programming. Typestate governs which methods are available on an object, and may change as the object evaluates. Kouzapas et al. [125] describe the design and implementation of two tools, Mungo and StMungo, which leverage typestate to support static checking of conformance to multiparty session types in the Java programming language. StMungo (or "Scribble-to-Mungo") translates from Scribble local protocols into typestate specifications, and Mungo checks to see whether Java objects correctly follow their typestate.

Hu and Yoshida [104] describe an approach called *endpoint API generation*, where local types guide the generation of *state channel* objects for each role. State channels guide a developer in following the protocol through the use of an object-oriented call-chaining API. Linearity is enforced dynamically through a simple run-time check, ensuring that each state channel object is used only once.

Type providers [171] allow statically-typed access to unstructured and untyped external data sources such as CSV files and SQL schemas via compile-time metaprogramming. Neykova et al. [153] leverage the work on endpoint API generation to define a *session type provider*, extending type providers to the domain of communication-centric software and introducing *interaction refinements*: predicates on message payloads which are enforced by use of an SMT solver.

Scalas et al. [192] extend the continuation-passing translation from binary session types into the linear π-calculus introduced by Kobayashi [118] and later extended by Dardha et al. [54] to the multiparty setting. Their approach lends itself to an implementation of multiparty session types in Scala following previous work on `lchannels` [190], in particular being the first work to support distributed delegation in the multiparty setting.

2.4.1.3 Runtime Monitoring

An alternative approach to checking conformance to protocols statically is to verify conformance at run-time. Deniélou and Yoshida [60] describe deep connections between multiparty session types and communicating finite-state automata [25], identifying a class of communicating finite-state automata called *multiparty session automata*, which enjoy safety properties such as deadlock-freedom.

Multiparty session automata can be used as *monitors* to dynamically enforce compliance with a session at run-time. Chen et al. [35] and Bocchi et al. [23] describe the theory of run-time monitoring of communication against session types. The formalism consists of an unmonitored semantics; a labelled transition system semantics of monitors; and a monitored semantics where actions are predicated on labels emitted by monitor reduction. The key results are of safety and transparency: safety means that the processes behave in accordance with the global

specification, and transparency means that a monitored network behaves exactly the same as an equivalent unmonitored network conforming to the specification.

SPY [152] is the first implementation of multiparty session types in a dynamically-checked programming language, implementing a Python API for session programming where communication safety is guaranteed through runtime monitors generated from Scribble specifications. Demangeon et al. [59] extend this work with an *interruptible* construct, allowing blocks to be *interrupted* by incoming messages.

Neykova and Yoshida [150] are first to integrate multiparty session types and the actor model via dynamic monitoring. In the conceptual framework proposed by the authors, each actor is an entity which may take part in multiple sessions, and where a message received in one session may trigger a message to be sent in another session. The conceptual framework is implemented in Python, and communication between actors is mediated via monitors derived from Scribble specifications. Subsequent work [71] implements an extended version of Neykova & Yoshida's conceptual framework in Erlang, motivating the use of subsessions [58] to allow parts of a protocol to be repeated with new participants. Neykova and Yoshida [151] investigate failure recovery strategies in Erlang, using information gained from protocols to compute and revert to safe states when a failure occurs.

2.4.2 Correspondence with Linear Logic

Carbone et al. [31] provide the first logical basis for multiparty session types, introducing a process calculus *MCP* with a strong correspondence to linear logic. The key insight is to generalise the binary cut rule with a 'coherence cut' rule allowing composition of multiple processes taking part in a session, so long as all local types are *coherent*. Coherence takes a bottom-up approach to ensuring that local types are compatible, as opposed to the more common top-down notion of projection.

Carbone et al. [30] build on this work and strengthen its connection to linear logic. In particular, the authors introduce a calculus of *governed classical processes* (GCP) and a reformulation of MCP; dispense with the notion of roles in favour of variables; and show that via the use of an *arbiter* process, it is possible to translate MCP and GCP into binary CP.

Toninho and Yoshida [204] provide another perspective on the connection between multiparty session types and classical linear logic from the perspective of *interconnection networks* [2], which describe the connections between participants in a multiparty session. The authors show that the interconnection networks allowed by multiparty sessions are strictly more expressive than the interconnection networks in classical linear logic, but show that this expressive power can be regained through the use of a controlled multicut rule.

Chapter 3

Synchronous GV

3.1 Introduction

In this chapter we describe Synchronous GV (SGV), a linear λ-calculus with session types and synchronous communication. Synchronous GV is a minimal core language which we extend modularly throughout the thesis.

As we discussed in Chapter 2, Wadler [213] introduces a process calculus *CP* with a strong correspondence with linear logic, and a core functional language *GV*, and shows a translation from GV into CP. GV is inspired by the linear functional language described by Gay and Vasconcelos [77], which has obtained the name LAST (for Linear Asynchronous Session Types) in the literature [135]. LAST is a fairly fully-fledged language which includes asynchrony, subtyping, machinery for proving the boundedness of buffers, recursion, a mix of linear and unrestricted types, and a flexible mechanism for initiating sessions known as *access points*.

The flexibility of LAST comes at a cost. LAST programs preserve typing under reduction, but the notion of progress enjoyed by LAST does not preclude deadlock. Additionally, access points admit nondeterminism, and the use of a fixpoint combinator admits non-terminating programs. In contrast, the logical grounding of GV endows it with a strong metatheory, encompassing preservation, deadlock-freedom, global progress, determinism, and termination. Naturally, this makes GV programs less expressive, but it is possible to modularly extend GV with various features, knowing which properties remain.

SGV is a variant of GV based primarily on the incarnation of GV described by Lindley and Morris [132]; we can reuse most of the definitions and proof techniques directly. We make several modifications, in particular omitting features required only to aid the translations to- and from CP such as the link construct, which is required in order to simulate the axiom rule, and the weak explicit substitutions [130] required for CP to simulate β-reduction in GV.

Of course, making the above changes is a tradeoff between simplicity of the language and the correspondence with logic. By making these modifications, we lose the ability to translate

SGV into CP and vice-versa. On the other hand, we obtain an easily-extensible core language which retains the strong metatheory stemming from GV's logical foundations.

Furthermore, we make several stylistic changes: we separate syntactic classes for runtime names and variables as they are fundamentally different entities—an observation which will become crucial in Chapter 9—and describe communication and concurrency constructs as language primitives instead of constants.

3.2 Synchronous GV

In this section, we describe the syntax and typing rules for SGV terms.

3.2.1 Syntax and Typing Rules for Terms

Synchronous GV extends a linear λ-calculus with linear tensor products and linear sums, with session types and constructs for communication and concurrency. Figure 3.1 describes the syntax of SGV types and terms.

Types and Session Types. Types are ranged over by A, B and include the unit type $\mathbf{1}$, the linear function type $A \multimap B$, the linear sum type $A + B$, the linear product type $A \times B$, and session types S. Note that for simplicity, we do not consider intuitionistic terms here; the calculus could well be extended with an unrestricted exponential modality following Wadler [212]; we show a concrete extension of GV with unrestricted types in Chapter 9, 9.4.

Session types are types for channel endpoints. Type $!A.S$ can be read 'send a value of type A and continue as S', and dually type $?A.S$ can be read 'receive a value of type A and continue as S'. As opposed to the single End type discussed in Chapter 2, we have two types signifying the end of a session. Type $\mathsf{End}_!$ is the unit of sending, interpreted as the type of an endpoint of a child process where no more messages are to be sent or received; and $\mathsf{End}_?$ is the unit of receiving, interpreted as the type of endpoint held by a parent process where the session is finished and the endpoint may be closed. We revisit the variation with a single self-dual End type in §3.4.1.

Terms. Terms include variables, and the standard introduction and elimination forms for the unit value, products, and sums.

SGV includes four additional primitives for communication and concurrency. The $\mathbf{fork}\, M$ construct spawns term M as a new thread, and creates a session channel with which to communicate with the spawned process. The $\mathbf{send}\, M\, N$ construct sends term M along session endpoint N. The $\mathbf{receive}\, M$ construct receives a value and updated channel endpoint from endpoint M. Finally, $\mathbf{wait}\, M$ synchronises with the channel endpoint M connected to a fully-evaluated child process, which terminates the child process and discards the channel.

$$
\begin{array}{lll}
\text{Types} & A,B,C & ::= \mathbf{1} \mid A \multimap B \mid A + B \mid A \times B \mid S \\
\text{Session Types} & S & ::= !A.S \mid ?A.S \mid \mathsf{End}_! \mid \mathsf{End}_? \\
\\
\text{Variables} & x,y,z & \\
\text{Terms} & L,M,N & ::= x \mid \lambda x.M \mid M\,N \\
 & & \quad \mid () \mid \mathbf{let}\,() = M\,\mathbf{in}\,N \\
 & & \quad \mid (M,N) \mid \mathbf{let}\,(x,y) = M\,\mathbf{in}\,N \\
 & & \quad \mid \mathbf{inl}\,M \mid \mathbf{inr}\,M \mid \mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \\
 & & \quad \mid \mathbf{fork}\,M \mid \mathbf{send}\,M\,N \mid \mathbf{receive}\,M \mid \mathbf{wait}\,M \\
\\
\text{Type Environments} & \Gamma & ::= \cdot \mid \Gamma, x : A
\end{array}
$$

Figure 3.1: Syntax of Synchronous GV Types and Terms

Variables and Typing Environments. We follow Barendregt's variable convention [15] (working up to renaming of bound variables and channel names), and we treat typing environments as unordered. We write Γ_1, Γ_2 to mean the disjoint union of environments Γ_1 and Γ_2.

Typing Rules for Terms Figure 3.2 describes the typing rules for terms. The typing judgement $\Gamma \vdash M : A$ can be read as "under typing context Γ, term M has type A".

Duality on session types ensures compatibility of communication actions within a session: where one participant in a session sends a value of some type A, then the other participant in the session will receive a value of type A, and vice versa.

The type system is *linear*, requiring that each variable is used *exactly once*. Session type systems are necessarily linear, so as to avoid using an endpoint twice and violating session fidelity, as described in Chapter 2, and to avoid an endpoint being discarded and thus allowing incomplete implementations of protocols.

Linearity is enforced by the combination of two techniques. Firstly, rule T-Var requires the typing context to be empty apart from the variable being typed, and rule T-Unit requires an empty typing context; these restrictions ensure that each variable must be used *at least* once. Secondly, in rules containing multiple subterms (for example, T-Pair), the typing context is split into two disjoint contexts, meaning that each variable may be used *at most* once. Together, these techniques mean that variables must be used *precisely* once.

Apart from the modifications to ensure linearity, typing rules for variables, abstraction and application, unit and unit elimination, pair construction and elimination, and sum injections and elimination are all standard from the simply-typed λ-calculus.

Typing Rules for Terms $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Var}\ \frac{}{x:A \vdash x:A} \qquad \textsc{T-Abs}\ \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M : A \multimap B} \qquad \textsc{T-App}\ \frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M\,N : B}$$

$$\textsc{T-Unit}\ \frac{}{\cdot \vdash () : \mathbf{1}} \qquad \textsc{T-LetUnit}\ \frac{\Gamma_1 \vdash M : \mathbf{1} \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\,() = M\,\mathbf{in}\,N : A}$$

$$\textsc{T-Pair}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash (M, N) : A \times B} \qquad \textsc{T-LetPair}\ \frac{\Gamma_1 \vdash M : A \times B \quad \Gamma_2, x:A, y:B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\,(x,y) = M\,\mathbf{in}\,N : C}$$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}\,M : A + B} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}\,M : A + B} \qquad \textsc{T-Case}\ \frac{\Gamma_1 \vdash L : A + B \quad \Gamma_2, x:A \vdash M : C \quad \Gamma_2, y:B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} : C}$$

$$\textsc{T-Fork}\ \frac{\Gamma \vdash M : S \multimap \mathsf{End}_!}{\Gamma \vdash \mathbf{fork}\,M : \overline{S}} \qquad \textsc{T-Send}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : !A.S}{\Gamma_1, \Gamma_2 \vdash \mathbf{send}\,M\,N : S} \qquad \textsc{T-Recv}\ \frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{receive}\,M : (A \times S)}$$

$$\textsc{T-Wait}\ \frac{\Gamma \vdash M : \mathsf{End}_?}{\Gamma \vdash \mathbf{wait}\,M : \mathbf{1}}$$

Duality $\boxed{\overline{S}}$

$$\overline{!A.S} = ?A.\overline{S} \qquad \overline{?A.S} = !A.\overline{S} \qquad \overline{\mathsf{End}_!} = \mathsf{End}_? \qquad \overline{\mathsf{End}_?} = \mathsf{End}_!$$

Figure 3.2: Typing Rules for Synchronous GV Types and Terms

Given a function M with type $S \multimap \mathsf{End}_!$, rule T-Fork types term $\mathbf{fork}\, M$ as having type $\overline{S}$; note that the endpoints have dual session types and thus communication with the forked process is compatible.

Term $\mathbf{send}\, M\, N$ has type S, given that M has type A, and N has type $!A.S$, ensuring that the session endpoint supports sending a value, and the type of value sent along the channel matches that specified by the session type. The return type is the continuation of the session. Similarly, $\mathbf{receive}\, M$ has type $(A \times S)$ if M has type $?A.S$, ensuring that the session endpoint M supports receiving a value of type A, and returns a pair of the received value and the updated endpoint. Finally, $\mathbf{wait}\, M$ is the elimination form for session endpoints of type $\mathsf{End}_?$.

Encoding Branching and Selection. Readers well-versed in the literature of session types might be surprised that Synchronous GV does not build in branching and selection, as in the two-factor authentication example, where the server chooses whether to authenticate, challenge, or deny access to a client. An important insight of the work of Dardha et al. [54] is that such constructs are redundant and can be implemented using sum types and session delegation. The intuition is as follows: to implement branching, one receives a label tagging the continuation of the session; to implement selection, one sends a label tagging the continuation of the session. The "tagging" functionality can be achieved by injections into a sum type, but what remains is the ability to use the tag to describe the continuation of the session. In fact, this functionality need not be encoded in the session type itself, since endpoints can be sent as part of a session. Therefore, the key insight is to end the current session, start a new session with the desired continuation, and send the tagged endpoint.
Formally, we may encode the types as follows:

$$S_1 \oplus S_2 \triangleq\ !(\overline{S_1} + \overline{S_2}).\mathsf{End}_? \qquad\qquad S_1 \,\&\, S_2 \triangleq\ ?(S_1 + S_2).\mathsf{End}_!$$

Let ℓ range over $\{\mathsf{inl}, \mathsf{inr}\}$. We may encode the terms as follows:

$$\begin{aligned} \mathbf{select}\, \ell\, M &\triangleq \mathbf{fork}\, (\lambda x.\mathbf{send}\, (\ell\, x)\, M) \\ \mathbf{offer}\, L\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} &\triangleq \mathbf{let}\, (x, s) = \mathbf{receive}\, L\, \mathbf{in} \\ &\quad\ \mathbf{wait}\, s; \\ &\quad\ \mathbf{case}\, x\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} \end{aligned}$$

Furthermore, as binary sums generalise to (monomorphic) variant types, this technique can be used to encode an arbitrary number of choices.

3.2.2 Runtime Syntax

So far, we have seen the *static* syntax and typing rules for Synchronous GV. To reason about the *semantics* of Synchronous GV, as well as its metatheory, we must extend the calculus with a language of *configurations* describing the state of the program.

Runtime Types	$R ::= S \mid S^\sharp$
Names	a, b, c
Terms	$M ::= \cdots \mid a$
Values	$U, V, W ::= x \mid a \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\, V \mid \mathbf{inr}\, V$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= (\nu a)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M$
Thread Flags	$\phi ::= \bullet \mid \circ$
Type Environments	$\Gamma ::= \cdots \mid \Gamma, a : R$
Evaluation Contexts	$E ::= [\,] \mid E\, M \mid V\, E$
	$\mid \mathbf{let}\,() = E\,\mathbf{in}\,M \mid \mathbf{let}\,(x, y) = E\,\mathbf{in}\,M \mid (E, V) \mid (V, E)$
	$\mid \mathbf{inl}\, E \mid \mathbf{inr}\, E \mid \mathbf{case}\, E\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}$
	$\mid \mathbf{fork}\, E \mid \mathbf{send}\, E\, M \mid \mathbf{send}\, V\, E \mid \mathbf{receive}\, E \mid \mathbf{wait}\, E$
Thread Contexts	$\mathcal{F} ::= \phi E$
Configuration Contexts	$\mathcal{G} ::= [\,] \mid (\nu a)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$

Figure 3.3: Synchronous GV Runtime Syntax

Figure 3.3 shows the runtime syntax of SGV. We introduce the runtime type of channels $S^\sharp$, which is the type of a *channel* and ascribed to a name a before it is split into two *endpoints* of type S and $\overline{S}$ over a parallel composition. Values of type $S^\sharp$ do not appear in terms. We also introduce a class of *runtime names* a which do not appear in closed source programs; instead, they are introduced by E-Fork and are bound by name restrictions. Previous work on GV [132, 133, 136] has a simpler syntax by 'punning' variables and runtime names, however we elect not to here as to do so poses problems when considering exceptions, where the distinction between static and dynamic names becomes more important.

Configurations, ranged over by $\mathcal{C}, \mathcal{D}, \mathcal{E}$, describe the runtime state of a program. Name restrictions $(\nu a)\mathcal{C}$ bind a runtime name a in configuration $\mathcal{C}$. Parallel composition $\mathcal{C} \parallel \mathcal{D}$ denotes two configurations $\mathcal{C}$ and $\mathcal{D}$ evaluating in parallel. Finally, ϕM describes a term M running as a thread; thread flags ϕ can either be $\bullet$, denoting that the thread is a *main thread* and may return a value, or $\circ$, denoting that the thread is a *child thread* and must return a finished endpoint of type $\mathsf{End}_!$.

Evaluation contexts E describe a deterministic, call-by-value evaluation strategy with left-to-right evaluation of arguments.

3.2.3 Operational Semantics

Free names. We write $\mathsf{fn}(\mathcal{C})$ and $\mathsf{fn}(M)$ for the set of free runtime names (i.e., runtime names a not occurring under a ν-binder $(\nu a)\mathcal{C}$) contained in configurations and terms respectively.

Term Reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$

$$\begin{array}{lr}
\textsc{E-Lam} & (\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\textsc{E-Unit} & \mathbf{let}\,() = ()\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M \\
\textsc{E-Pair} & \mathbf{let}\,(x,y) = (V,W)\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M\{V/x,W/y\} \\
\textsc{E-Inl} & \mathbf{case}\,\mathbf{inl}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\textsc{E-Inr} & \mathbf{case}\,\mathbf{inr}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} N\{V/y\} \\
\textsc{E-Lift} & E[M] \longrightarrow_{\mathsf{M}} E[N], \quad \text{if } M \longrightarrow_{\mathsf{M}} N
\end{array}$$

Configuration Equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad (\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

$$\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}), \quad \text{if } a \notin \mathsf{fn}(\mathcal{C})$$

Configuration Reduction $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$\begin{array}{lr}
\textsc{E-Fork} & \mathcal{F}[\mathbf{fork}\,V] \longrightarrow (\nu a)(\mathcal{F}[a] \parallel \circ V\,a) \quad (a \text{ is fresh}) \\
\textsc{E-Comm} & \mathcal{F}[\mathbf{send}\,V\,a] \parallel \mathcal{F}'[\mathbf{receive}\,a] \longrightarrow \mathcal{F}[a] \parallel \mathcal{F}'[(V,a)] \\
\textsc{E-Wait} & (\nu a)(\mathcal{F}[\mathbf{wait}\,a] \parallel \circ a) \longrightarrow \mathcal{F}[()] \\
\textsc{E-LiftM} & \phi M \longrightarrow \phi N \quad (\text{if } M \longrightarrow_{\mathsf{M}} N) \\
\textsc{E-Lift} & \mathcal{G}[\mathcal{C}] \longrightarrow \mathcal{G}[\mathcal{D}] \quad (\text{if } \mathcal{C} \longrightarrow \mathcal{D})
\end{array}$$

Figure 3.4: Reduction of Synchronous GV Terms and Configurations

Reduction on Terms. The relation $M \longrightarrow_{\mathsf{M}} N$ describes the standard β-reduction rules for the functional constructs. All are standard.

Equivalence. Configuration equivalence $\mathcal{C} \equiv \mathcal{D}$ is defined as the smallest congruence relation satisfying the equivalence axioms in Figure 3.4. The equivalence axioms describe the associativity and commutativity of parallel composition, the reordering of name restrictions, and the standard π-calculus scope extrusion rule [144, 189]. The definition of configuration equivalence as a congruence relation concretely corresponds to the addition of the following rules:

$$\frac{}{\mathcal{C} \equiv \mathcal{C}} \qquad \frac{\mathcal{D} \equiv \mathcal{C}}{\mathcal{C} \equiv \mathcal{D}} \qquad \frac{\mathcal{C} \equiv \mathcal{D} \quad \mathcal{D} \equiv \mathcal{E}}{\mathcal{C} \equiv \mathcal{E}} \qquad \frac{\mathcal{C} \equiv \mathcal{D}}{(\nu a)\mathcal{C} \equiv (\nu a)\mathcal{D}} \qquad \frac{\mathcal{C} \equiv \mathcal{D}}{\mathcal{C} \parallel \mathcal{E} \equiv \mathcal{D} \parallel \mathcal{E}}$$

Equivalence shows that while the syntax imposes a tree structure on configurations, they may be treated more like multisets.

Configuration Contexts. Configuration contexts $\mathcal{G}[-]$ allow reduction under name restrictions and parallel compositions.

Reduction on Configurations. Whereas the $\longrightarrow_{\mathsf{M}}$ relation gives a semantics to the functional fragment of the language, communication and concurrency constructs cannot reduce on their own. To give a semantics to such constructs, we introduce an evaluation relation $\longrightarrow$ on configurations.

Rule E-Fork describes the semantics of evaluating **fork** $(\lambda x.M)$ in an evaluation context: evaluating the operation creates a fresh channel name a, returns a to the calling thread, and spawns $M\{a/x\}$ as a child thread. In essence, **fork** performs *two* operations: spawning a new thread, and creating a fresh name to be used to communicate with the thread. Coupling these two operations ensures a tree-like topology on configurations and is key to ensuring deadlock-freedom in the core calculus.

Rule E-Comm describes synchronous communication between two threads over name a. Term **send** $V\ a$ reduces to a, returning the updated channel endpoint, and **receive** a reduces to a pair of the transmitted value V and the updated endpoint a.

Rule E-Wait eliminates name a when the session has completed, returning the unit value to the calling thread, and garbage-collecting the child thread.

We write $\Longrightarrow$ to mean the relation $\equiv\longrightarrow\equiv$—that is, reduction modulo equivalence.

3.3 Metatheory

Typing of Configurations. To reason about the metatheory of SGV, we require typing rules for configurations, shown in Figure 3.5. Note that configuration typing rules need not be used as part of a typechecker: configuration typing rules encode well-formedness conditions about the program state at runtime, and are needed only to reason about SGV's metatheory.

We add rule T-Name to type runtime names occurring in terms as a result of reduction.

Rule T-Nu types a name restricton $(\nu a)\mathcal{C}$ which binds name a in configuration $\mathcal{C}$; name a is ascribed type $S^\sharp$ which must be split over a parallel composition using either rule T-Connect$_1$ or T-Connect$_2$. We require both rules for parallel composition in order to preserve configuration typing under commutativity of parallel composition. To see why, consider the following derivation:

$$\frac{\Gamma_1, a : S \vdash \mathcal{C} \quad \Gamma_2, a : \overline{S} \vdash \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^\sharp \vdash^\phi \mathcal{C} \parallel \mathcal{D}}$$

Supposing that we only had rule T-Connect$_1$ available, we would not be able to commute the

Updated Term Typing Rule $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Name}\ \frac{}{a : S \vdash a : S}$$

Typing of Configurations $\boxed{\Gamma \vdash^{\phi} \mathcal{C}}$

$$\textsc{T-Nu}\ \frac{\Gamma, a : S^{\sharp} \vdash^{\phi} \mathcal{C}}{\Gamma \vdash^{\phi} (\nu a)\mathcal{C}} \qquad \textsc{T-Connect}_1\ \frac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}} \qquad \textsc{T-Connect}_2\ \frac{\Gamma_1, a : \overline{S} \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : S \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}}$$

$$\textsc{T-Thread}\ \frac{\Gamma \vdash M : \mathsf{End}_!}{\Gamma \vdash^{\circ} \circ M} \qquad \textsc{T-Main}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash^{\bullet} \bullet M}$$

Combination of Flags $\boxed{\phi_1 + \phi_2}$

$$\bullet + \circ = \bullet \qquad \circ + \bullet = \bullet \qquad \circ + \circ = \circ \qquad \bullet + \bullet \text{ undefined}$$

Reduction on Session Types and Typing Environments $\boxed{S \longrightarrow S'}$ $\boxed{\Gamma \longrightarrow \Gamma'}$

$$\frac{}{!A.S \longrightarrow S} \qquad \frac{}{?A.S \longrightarrow S} \qquad \frac{S \longrightarrow S'}{\Gamma, a : S^{\sharp} \longrightarrow \Gamma, a : S'^{\sharp}}$$

Figure 3.5: Runtime Typing for Synchronous GV

two subconfigurations while retaining typeability:

$$\frac{\Gamma_1, a : S \vdash \mathcal{D} \quad \Gamma_2, a : \overline{S} \vdash \mathcal{C}}{\Gamma_1, \Gamma_2, a : S^\sharp \vdash \mathcal{D} \parallel \mathcal{C}}$$

Rule T-Thread encodes the invariant that each child thread must return a fully-used session endpoint of type $\mathsf{End}_!$, and rule T-Main states that the main thread of the configuration may return a value of any type.

It is convenient to define the notion of a *ground configuration*. We write $\Gamma \vdash^\bullet \mathcal{C} : A$ if the derivation of $\Gamma \vdash^\bullet \mathcal{C}$ has a subderivation of the form:

$$\frac{\Gamma' \vdash M : A}{\Gamma' \vdash^\bullet \bullet M}$$

Definition 1 (Ground Configuration)**.** *We say that $\mathcal{C}$ is a* ground configuration *if there exists A such that $\cdot \vdash^\bullet \mathcal{C} : A$ and A contains no session types or function types.*

The judgement for configurations, written $\Gamma \vdash^\phi \mathcal{C}$, can be read as, "under typing environment Γ and flag ϕ, configuration $\mathcal{C}$ is well-typed".

The configuration typing rules are sufficient to rule out ill-formed and deadlocking processes. We show three examples: unmatched communication; communication mismatch; and deadlock.

Unmatched communication.

$$(\nu a)(\bullet \mathbf{send}\, 5\, a)$$

This configuration cannot reduce since name a is not split over two threads, and thus the communication action will never occur.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \vdash 5 : \mathsf{Int}} \quad a : (!\mathsf{Int}.\mathsf{End})^\sharp \vdash a : ?}{a : (!\mathsf{Int}.\mathsf{End})^\sharp \vdash \mathbf{send}\, 5\, a : ?}}{a : (!\mathsf{Int}.\mathsf{End})^\sharp \vdash^\bullet \bullet(\mathbf{send}\, 5\, a)}}{\cdot \vdash^\bullet (\nu a)(\bullet \mathbf{send}\, 5\, a)}$$

It is not possible to construct a typing derivation, since values of type $S^\sharp$ may not occur in terms.

Communication mismatch.

$$(\nu a)(\bullet \mathbf{send}\, 5\, a \parallel \circ \mathbf{send}\, 10\, a)$$

This configuration cannot reduce since uses of name a are not dual, resulting in a communication mismatch.

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash 5 : \mathsf{Int}} \quad \dfrac{}{a : !\mathsf{Int}.\mathsf{End}_? \vdash a : !\mathsf{Int}.\mathsf{End}_?}}{a : !\mathsf{Int}.\mathsf{End}_? \vdash \mathbf{send}\,5\,a : \mathsf{End}_?} \quad \dfrac{}{a : ?\mathsf{Int}.\mathsf{End}_! \not\vdash \mathbf{send}\,10\,a}}{a : !\mathsf{Int}.\mathsf{End}_? \vdash^{\bullet} \bullet\mathbf{send}\,5\,a \quad a : ?\mathsf{Int}.\mathsf{End}_! \not\vdash^{\circ} \circ\mathbf{send}\,10\,a}}{a : (!\mathsf{Int}.\mathsf{End}_?)^{\sharp} \vdash^{\bullet} \bullet\mathbf{send}\,5\,a \parallel \circ\mathbf{send}\,10\,a}}{\cdot \not\vdash^{\bullet} (\nu a)(\bullet\mathbf{send}\,5\,a \parallel \circ\mathbf{send}\,10\,a)}$$

The type system rules out the example since (by T-Connect$_1$ and T-Connect$_2$), channel type $(!\mathsf{Int}.\mathsf{End}_?)^{\sharp}$ must be split into two endpoints of dual types $!\mathsf{Int}.\mathsf{End}_?$ and $?\mathsf{Int}.\mathsf{End}_!$.

Deadlock.

$$\begin{pmatrix}\mathbf{let}\,(res,x) = \mathbf{receive}\,a\,\mathbf{in}\\ \mathbf{let}\,y = \mathbf{send}\,res\,b\,\mathbf{in}\,M\end{pmatrix} \parallel \circ \begin{pmatrix}\mathbf{let}\,(res,y) = \mathbf{receive}\,b\,\mathbf{in}\\ \mathbf{let}\,x = \mathbf{send}\,res\,x\,\mathbf{in}\,N\end{pmatrix}$$

This configuration cannot reduce since the communication is deadlocking: the first process attempts to receive along name a and then send along name b, whereas the second process attempts to receive along b and then send along name a. The type system rules out the example as there is no way to share both a and b along the single parallel composition:

$$\dfrac{\Gamma_1, a : ?A.S \not\vdash^{\bullet} \bullet\begin{pmatrix}\mathbf{let}\,(res,x) = \mathbf{receive}\,a\,\mathbf{in}\\ \mathbf{let}\,y = \mathbf{send}\,res\,b\,\mathbf{in}\,M\end{pmatrix} \quad \Gamma_2, a : !A.\overline{S}, b : (!A.T)^{\sharp} \not\vdash^{\circ} \circ\begin{pmatrix}\mathbf{let}\,(res,y) = \mathbf{receive}\,b\,\mathbf{in}\\ \mathbf{let}\,x = \mathbf{send}\,res\,x\,\mathbf{in}\,N\end{pmatrix}}{\Gamma_1, \Gamma_2, a : (?A.S)^{\sharp}, b : (!A.T)^{\sharp} \vdash^{\bullet} \bullet\begin{pmatrix}\mathbf{let}\,(res,x) = \mathbf{receive}\,a\,\mathbf{in}\\ \mathbf{let}\,y = \mathbf{send}\,res\,b\,\mathbf{in}\,M\end{pmatrix} \parallel \circ \begin{pmatrix}\mathbf{let}\,(res,y) = \mathbf{receive}\,b\,\mathbf{in}\\ \mathbf{let}\,x = \mathbf{send}\,res\,x\,\mathbf{in}\,N\end{pmatrix}}$$

Safe but untypeable configurations. The type system is conservative in that it disallows some cyclic, yet non-deadlocking terms. Consider the configuration:

$$(\nu a)(\nu b)(\bullet\begin{pmatrix}\mathbf{let}\,(res,x) = \mathbf{receive}\,a\,\mathbf{in}\\ \mathbf{let}\,y = \mathbf{send}\,res\,b\,\mathbf{in}\,M\end{pmatrix} \parallel \circ \begin{pmatrix}\mathbf{let}\,x = \mathbf{send}\,5\,a\,\mathbf{in}\\ \mathbf{let}\,(res,x) = \mathbf{receive}\,b\,\mathbf{in}\,N\end{pmatrix})$$

The first process receives along name a before sending on name b, and the second process sends along name a before receiving on name b. While the configuration is cyclic since two channels connect the two processes, it does not deadlock (assuming the interactions in M and N are also non-deadlocking). Nevertheless, the configuration is untypeable in SGV since there is no way to share both a and b over the single parallel composition. Safe cyclic processes may be typed using techniques such as channel priorities [119], and indeed Dardha and Gay [53] extend CP using channel priorities to allow safe cyclic processes. These techniques may be used in tandem with the techniques described in the remainder of this thesis, but are otherwise orthogonal.

When implementing practical languages, expressiveness may be regained through extensions such as access points (discussed further in Chapter 9) at the cost of weaker metatheoretic guarantees. In this thesis however, we concentrate on well-behaved calculi, and show that extensions such as asynchrony and exception handling do not compromise the core guarantees.

3.3.1 Overview of Metatheory

Due to its roots in linear logic, SGV enjoys a strong metatheory, providing preservation, global progress, confluence, and termination.

Preservation

Typeability of configurations is preserved by reduction. As is common in session-typed functional languages based on linear logic, typeability is not preserved by equivalence, but reduction never relies on the ill-typed use of an equivalence.

Global Progress

Due to the acyclicity of configurations, SGV is deadlock-free and does not 'get stuck'. In the case of closed configurations where the main thread contains no free names, a non-reducing configuration is equal to a value.

Determinism

Term reduction is entirely deterministic. Although the reduction relation on configurations is nondeterministic, communication actions are independent due to linearity and can therefore be performed in either order.

Termination

There are no infinite reduction sequences from a well-typed configuration. As SGV is a purely-linear calculus, this follows via an elementary argument due to linearity, since the size of a term strictly decreases under reduction.

3.3.2 Preservation and Session Fidelity

Before showing that typing is preserved under reduction, we must first state some auxiliary results.

We begin with a substitution lemma that holds for linear λ-calculi. Note that due to linearity, we must explicitly state that Γ_1, Γ_2 is defined.

Lemma 1 (Substitution)**.** *Suppose* $\Gamma_1, x : B \vdash M : A$ *and* $\Gamma_2 \vdash N : B$*, where* Γ_1, Γ_2 *is defined. Then* $\Gamma_1, \Gamma_2 \vdash M\{N/x\} : A$*.*

Proof. By induction on the derivation of $\Gamma_1, x : B \vdash M : A$. □

Next, we require some lemmas which allow us to manipulate evaluation contexts, which are adapted for linear λ-calculi and follow a similar form to those of Wright and Felleisen [216]. The first, subterm typeability, states that if a term $E[M]$ is well-typed, then there exists some subderivation showing that M is well-typed. This lemma allows us to decompose evaluation contexts.

Lemma 2 (Subterm typeability)**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : A$*, then there exists some subderivation* $\mathbf{D}'$ *of* $\mathbf{D}$ *concluding* $\Gamma_2 \vdash M : B$*, where the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* E*.*

Proof. By induction on the structure of E. □

The next lemma, *subterm replacement*, allows us to replace the term in the hole of an evaluation context.

Lemma 3 (Subterm replacement)**.** *If:*

- $\mathbf{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : A$
- $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma_2 \vdash M : B$
- *The position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* E
- $\Gamma_3 \vdash N : B$
- Γ_1, Γ_3 *is well-defined*

then $\Gamma_1, \Gamma_3 \vdash E[N] : A$*.*

Proof. By induction on the structure of E. □

So far, we have shown results for manipulating *term* contexts. *Configuration* contexts allow us to reason about reduction under name restrictions and for processes that are part of a parallel composition. The lemmas follow the same structure as the analogous lemmas for term contexts, but are complicated slightly by the fact that $(\nu a)\mathcal{G}$ binds a in $\mathcal{G}$. Nonetheless, replacement is safe when using environments related by the reduction relation. Again, both proofs follow by induction on the structure of the configuration context $\mathcal{G}$.

Lemma 4 (Subconfiguration typeability)**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma \vdash^{\phi} \mathcal{G}[\mathcal{C}]$*, then there exist* Γ', ϕ' *such that* $\mathbf{D}$ *has a subderivation* $\mathbf{D}'$ *that concludes* $\Gamma' \vdash^{\phi'} \mathcal{C}$ *and the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to the position of the hole in* $\mathcal{G}$*.*

Lemma 5 (Subconfiguration replacement)**.** *If:*

- $\mathbf{D}$ *is a derivation of* $\Gamma \vdash^{\phi} \mathcal{G}[\mathcal{C}]$

- $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding that* $\Gamma' \vdash^{\phi'} \mathcal{C}$ *for some* Γ', ϕ'
- *The position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* $\mathcal{G}$
- $\Gamma'' \vdash^{\phi'} \mathcal{D}$ *for some* Γ'' *such that* $\Gamma' \longrightarrow^? \Gamma''$

then there exist some Γ''' *such that* $\Gamma \longrightarrow^? \Gamma'''$ *and* $\Gamma''' \vdash^{\phi} \mathcal{G}[\mathcal{D}]$.

Unsurprisingly, functional reduction preserves typing.

Lemma 6 (Preservation (SGV Terms))**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M N$*, then* $\Gamma \vdash N : A$.

Proof. Standard; by induction on the derivation of $M \longrightarrow_M N$. □

We may extend this result to show reduction on configurations preserves typing.

Theorem 1 (Preservation (SGV Configurations))**.** *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then there exists some* $\Gamma \longrightarrow^? \Gamma'$ *such that* $\Gamma' \vdash^{\phi} \mathcal{D}$.

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$, making use of Lemmas 1–5. The full proof can be found in Appendix A, but we show the case for E-Comm here. As there is a choice of flags, we make the decision to prove the case where the first flag is $\bullet$ and the second is $\circ$; the proofs for other cases are similar.

Case E-Comm

$$\bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a] \quad \longrightarrow \quad \bullet E[a] \parallel \circ E'[(V, a)]$$

Assumption:

$$\frac{\dfrac{\Gamma_1, \Gamma_2, a : S \vdash E[\mathbf{send}\, V\, a] : C}{\Gamma_1, \Gamma_2, a : S \vdash^{\bullet} \bullet E[\mathbf{send}\, V\, a]} \qquad \dfrac{\Gamma_3, a : \overline{S} \vdash E'[\mathbf{receive}\, a] : \mathsf{End}_!}{\Gamma_3, a : \overline{S} \vdash^{\circ} \circ E'[\mathbf{receive}\, a]}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S^\sharp \vdash^{\bullet} \bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a]}$$

By Lemma 2 and inversion on the typing relation:

$$\frac{\Gamma_2 \vdash V : A \quad a : !A.S' \vdash a : !A.S'}{\Gamma_2, a : !A.S' \vdash \mathbf{send}\, V\, a : S'}$$

Also by Lemma 2 and inversion on the typing relation:

$$\frac{a : ?A.\overline{S'} \vdash a : ?A.\overline{S'}}{a : ?A.\overline{S'} \vdash \mathbf{receive}\, a : (A \times \overline{S'})}$$

This reasoning allows us to refine our original derivation:

$$\dfrac{\dfrac{\Gamma_1,\Gamma_2,a:!A.S' \vdash E[\mathbf{send}\,V\,a]:C}{\Gamma_1,\Gamma_2,a:!A.S' \vdash^{\bullet} \bullet E[\mathbf{send}\,V\,a]} \quad \dfrac{\Gamma_3,a:?A.\overline{S'} \vdash E'[\mathbf{receive}\,a]:\mathsf{End}_!}{\Gamma_3,a:?A.\overline{S'} \vdash^{\circ} \circ E'[\mathbf{receive}\,a]}}{\Gamma_1,\Gamma_2,\Gamma_3,a:(!A.S')^\sharp \vdash^{\bullet} \bullet E[\mathbf{send}\,V\,a] \parallel \circ E'[\mathbf{receive}\,a]}$$

By Lemma 3, $\Gamma_1,a:S' \vdash E[a]:C$, and $\Gamma_2,\Gamma_3,a:\overline{S'} \vdash E'[(V,a)]:\mathsf{End}_!$ (that Γ_2,Γ_3 is well-defined follows from the fact that the two environments are disjoint).

Recomposing:

$$\dfrac{\dfrac{\Gamma_1,a:S' \vdash E[a]:C}{\Gamma_1,a:S' \vdash^{\bullet} \bullet E[a]} \quad \dfrac{\Gamma_2,\Gamma_3,a:\overline{S'} \vdash E'[(V,a)]:\mathsf{End}_!}{\Gamma_2,\Gamma_3,a:\overline{S'} \vdash^{\circ} \circ E'[(V,a)]}}{\Gamma_1,\Gamma_2,\Gamma_3,a:S'^\sharp \vdash^{\bullet} \bullet E[a] \parallel \circ E'[(V,a)]}$$

Finally, we can show that the environment in the first derivation reduces to the environment in the final derivation:

$$\dfrac{!A.S' \longrightarrow S'}{\Gamma_1,\Gamma_2,\Gamma_3,a:(!A.S')^\sharp \longrightarrow \Gamma_1,\Gamma_2,\Gamma_3:a:S'^\sharp}$$

as required.

□

Typing and Configuration Equivalence. Preservation of configuration reduction does not quite get us the whole way, however. Note that the relation $\longrightarrow$ is *not* defined modulo equivalence. In fact, unfortunately, typeability of configurations is *not* preserved by equivalence. Consider the well-typed configuration $\Gamma \vdash^{\phi} (\nu a)(\nu b)(\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}))$, where $a \in \mathsf{fn}(\mathcal{C})$, $b \in \mathsf{fn}(\mathcal{D})$, and $a,b \in \mathsf{fn}(\mathcal{E})$. While $\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$, we have that $\Gamma \not\vdash^{\phi} (\nu a)(\nu b)((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E})$, since only a or b would be present when typing $\mathcal{E}$. It helps to consider a typing derivation:

$$\dfrac{\dfrac{\Gamma_1,a:S \vdash^{\phi_1} \mathcal{C} \quad \dfrac{\Gamma_2,b:T \vdash^{\phi_2} \mathcal{D} \quad \Gamma_3,a:\overline{S},b:\overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2,\Gamma_3,a:\overline{S},b:T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}}{\Gamma_1,\Gamma_2,\Gamma_3,a:S^\sharp,b:T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})}}{\dfrac{\Gamma_1,\Gamma_2,\Gamma_3,a:S^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\nu b)(\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}))}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash^{\phi_1+\phi_2+\phi_3} (\nu a)(\nu b)(\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}))}}$$

where $\Gamma = \Gamma_1,\Gamma_2,\Gamma_3$ and $\phi = \phi_1 + \phi_2 + \phi_3$.

However, while $\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$, it is not the case that $\Gamma \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$, since there is no way of splitting $a:S^\sharp$ and $b:T^\sharp$ over both parallel compositions. For example:

$$\dfrac{\dfrac{\dfrac{\Gamma_1,\Gamma_2,a:S,b:T^\sharp \not\vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{D} \quad \Gamma_3,a:\overline{S} \not\vdash^{\phi_3} \mathcal{E}}{\Gamma_1,\Gamma_2,\Gamma_3,a:S^\sharp,b:T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}}{\Gamma_1,\Gamma_2,\Gamma_3,a:S^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\nu b)((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E})}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash^{\phi_1+\phi_2+\phi_3} (\nu a)(\nu b)((\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E})}$$

While this is inconvenient, it poses no problems for reduction. First, note that only associativity of parallel composition breaks typeability under equivalence.

Lemma 7. *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, where the derivation of* $\mathcal{C} \equiv \mathcal{D}$ *does not contain a use of the axiom for associativity of parallel composition, then* $\Gamma \vdash^{\phi} \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$; see Appendix A. □

Second, note that we may always safely re-associate parallel composition either directly, or by firstly commuting a configuration. Returning to our example, we have that $\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv \mathcal{C} \parallel (\mathcal{E} \parallel \mathcal{D})$, and that $\Gamma \vdash^{\phi} (\nu a)(\nu b)((\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D})$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1,a:S \vdash^{\phi_1} \mathcal{C} \quad \Gamma_3,a:\overline{S},b:\overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1;\Gamma_3,a:S^\sharp,b:\overline{T} \vdash^{\phi_1+\phi_3} \mathcal{C} \parallel \mathcal{E}} \quad \Gamma_2,b:T \vdash^{\phi_2} \mathcal{D}}{\Gamma_1,\Gamma_2,\Gamma_3,a:S^\sharp,b:T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D}}}{\Gamma_1,\Gamma_2,\Gamma_3,a:S^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\nu b)((\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D})}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash^{\phi_1+\phi_2+\phi_3} (\nu a)(\nu b)((\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D})}$$

We can state this result more generally:

Lemma 8.

1. *If* $\Gamma \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$*, then either* $\Gamma \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$ *or* $\Gamma \vdash^{\phi} (\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D}$*.*
2. *If* $\Gamma \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$*, then either* $\Gamma \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$ *or* $\Gamma \vdash^{\phi} \mathcal{D} \parallel (\mathcal{C} \parallel \mathcal{E})$*.*

Proof. Direct. We show the proof of property (1); the proof of property (2) is symmetric.

By the assumption that $\Gamma \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$ we have that $\Gamma = \Gamma_1,\Gamma_2,\Gamma_3,a:S^\sharp,b:T^\sharp$ and $\phi = \phi_1+\phi_2+\phi_3$. There are 4 cases, based on whether $a,b \in \mathsf{fn}(\mathcal{D})$ or $a,b \in \mathsf{fn}(\mathcal{E})$ (it cannot be the case that $a,b \in \mathsf{fn}(\mathcal{C})$, as $\mathcal{C}$ only occurs under a single parallel composition), and the exact dualisation (i.e., whether composition happens via T-Connect$_1$ or T-Connect$_2$).

Of these, we are only interested in the cases where the sharing of the names differs, as opposed to the dualisation. Thus, we consider the following two cases, where both compositions occur using T-Connect$_1$:

1. $\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C}$, and $\Gamma_2, a : \overline{S}, b : T \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$
2. $\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C}$, and $\Gamma_2, b : T \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$

Case $a \in \mathsf{fn}(\mathcal{C}), a, b \in \mathsf{fn}(\mathcal{D}), b \in \mathsf{fn}(\mathcal{E})$

$$\dfrac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \dfrac{\Gamma_2, a : \overline{S}, b : T \vdash^{\phi_2} \mathcal{D} \quad \Gamma_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})}$$

As $\mathcal{D}$ contains both a and b, associativity does not alter the sharing of names and may be applied safely.

$$\dfrac{\dfrac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : \overline{S}, b : T \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^\sharp, b : T \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{D}} \quad \Gamma_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}$$

Case $a \in \mathsf{fn}(\mathcal{C}); b \in \mathsf{fn}(\mathcal{D}); a, b \in \mathsf{fn}(\mathcal{E})$

$$\dfrac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \dfrac{\Gamma_2, b : T \vdash^{\phi_2} \mathcal{D} \quad \Gamma_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})}$$

We may not apply associativity directly, but we may first commute $\mathcal{D}$ and $\mathcal{E}$:

$$\dfrac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \dfrac{\Gamma_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E} \quad \Gamma_2, b : T \vdash^{\phi_2} \mathcal{D}}{\Gamma_2, \Gamma_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{E} \parallel \mathcal{D}}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{E} \parallel \mathcal{D})}$$

and from here we may safely re-associate to the left:

$$\dfrac{\dfrac{\Gamma_2, a : S \vdash^{\phi_1} \mathcal{C} \quad \Gamma_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3, a : S^\sharp, b : \overline{T} \vdash^{\phi_1+\phi_2} \mathcal{D} \parallel \mathcal{E}} \quad \Gamma_3, b : T \vdash^{\phi_3} \mathcal{D}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D}}$$

□

While ill-typed configurations *may* arise as a result of the equivalence relation, this has bearing on reduction: for every reduction relying on an ill-typed use of equivalence, there is an equivalent reduction relying only on well-typed uses of equivalence.

We first formally establish the acyclicity of configurations. The configuration typing rules T-Connect$_1$ and T-Connect$_2$ require that two processes must share a single channel name in order to be composed in parallel. We define $\mathsf{fn}(\mathcal{C})$ as the set of free names contained in a configuration; for example, a name a not occurring in the scope of a name restriction $(\nu a)\mathcal{C}$. With this, we may formalise the observation that two parallel configurations have precisely one name in common.

Lemma 9. *If $\Gamma \vdash^{\phi} \mathcal{C}$ and $\mathcal{C}$ can be written $\mathcal{C} = \mathcal{G}[\mathcal{D} \parallel \mathcal{E}]$, then $\mathsf{fn}(\mathcal{D}) \cap \mathsf{fn}(\mathcal{E}) = \{a\}$ for some name a.*

Proof. By induction on the derivation of $\Gamma \vdash^{\phi} \mathcal{C}$. The only interesting cases are T-Connect$_1$ and T-Connect$_2$, which partition the environment apart from a single shared name. □

We can now formally state that the existence of ill-typed equivalence has no bearing on reduction.

Theorem 2 (Preservation modulo equivalence (SGV))**.**
If $\Gamma \vdash^{\phi} \mathcal{C}$, $\mathcal{C} \equiv \mathcal{D}$, and $\mathcal{D} \longrightarrow \mathcal{D}'$, then:

1. *There exists some $\mathcal{E}$ such that $\mathcal{D} \equiv \mathcal{E}$, and $\Gamma \vdash^{\phi} \mathcal{E}$, and $\mathcal{E} \longrightarrow \mathcal{E}'$*
2. *There exists some Γ' such that $\Gamma \longrightarrow \Gamma'$ and $\Gamma' \vdash^{\phi} \mathcal{E}'$*
3. *$\mathcal{D}' \equiv \mathcal{E}'$*

Proof. By Lemma 7, we have that all equivalence axioms except the associativity of parallel composition preserve typing. Thus, we need only consider ill-typed uses of the associativity axiom:

$$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$$

The only non-trivial reduction cases are E-Send, E-Receive, and E-Wait. The only reason for us to apply the associativity of parallel composition rule from right-to-left is to enable configurations $\mathcal{C}$ and $\mathcal{D}$ to communicate.

For any new reduction to be possible, it must be the case therefore that there exists some $a \in \mathsf{fn}(\mathcal{C})$ and $a \in \mathsf{fn}(\mathcal{D})$.

As the left-hand-side of the equivalence is well-typed, by Lemma 9, we have that $\mathcal{C}$ and $\mathcal{E}$ have no names in common, that $\mathcal{D}$ and $\mathcal{E}$ share a name, and that the right-hand-side of the equivalence must be well-typed as there is still exactly one channel connecting each of the parallel compositions.

The argument for the case of applying the rule from right-to-left is symmetric. In summary, any ill-typed use of equivalence rules is useless, as it does not enable any more reductions. □

It follows that when we write $\mathcal{C} \Longrightarrow \mathcal{D}$, we can assume that we only consider the reductions using well-typed applications of equivalence rules.

3.3.3 Global Progress

The configuration typing rules for SGV ensure an acyclic, tree-structured communication topology, and thus rule out deadlocked configurations by construction. We proceed firstly by formalising this graph-theoretic notion of deadlock-freedom, and showing that well-typed terms are deadlock free. Next, we prove progress directly.

We begin by classifying the notion of a process being *blocked* on a particular channel name. We say that a term ϕM is *blocked* on a name a if it is waiting to send on, receive from, or wait on a. Formally:

$$\mathsf{blocked}(M,a) \triangleq \exists E.(M = E[\mathbf{send}\, V\, a]) \vee (M = E[\mathbf{receive}\, a]) \vee (M = E[\mathbf{wait}\, a])$$

If a thread M is blocked on a name a, then we know that it may only reduce when another process N is blocked on a, performing the dual communication action. Any communications on some name b in $\mathsf{fn}(E)$ require the communication on a to occur prior to the communication on b. We can formalise this notion of dependency, and also lift the result to arbitrary configurations of threads; note that the dependency may pass through an arbitrary third thread.

$$\begin{aligned}
\mathsf{depends}(a,b,M) &\triangleq \mathsf{blocked}(a,M) \wedge b \in \mathsf{fn}(M) \\
\mathsf{depends}(a,b,\mathcal{C}) &= (\exists \mathcal{G}, M.\mathcal{C} \equiv \mathcal{G}[M] \wedge \mathsf{depends}(a,b,M)) \vee \\
&\qquad (\exists \mathcal{G}, c.\mathcal{C} \equiv \mathcal{G}[\mathcal{D} \parallel \mathcal{E}] \\
&\qquad\quad \wedge \mathsf{depends}(a,c,\mathcal{D}) \wedge \mathsf{depends}(c,b,\mathcal{E}))
\end{aligned}$$

Deadlocked configurations are those which contain cyclic dependencies. As our notion of dependency encompasses transitive dependencies, it suffices to define a deadlocked configuration as the existence of a single cyclic dependency. Formally:

$$\mathsf{deadlocked}(\mathcal{C}) \triangleq \exists \mathcal{D}, \mathcal{E}, a, b.\mathcal{C} \equiv \mathcal{G}[\mathcal{D} \parallel \mathcal{E}] \wedge \mathsf{depends}(a,b,\mathcal{D}) \wedge \mathsf{depends}(b,a,\mathcal{E})$$

We can now state that well-typed SGV programs are not deadlocked.

Theorem 3 (Deadlock Freedom (SGV))**.** *If* $\Gamma \vdash^{\phi} \mathcal{C}$*, then* $\neg$*deadlocked*$(\mathcal{C})$.

Proof. By contradiction. Suppose that $\mathsf{deadlocked}(\mathcal{C})$. By the definition of $\mathsf{deadlocked}(\mathcal{C})$, there would have to exist some configurations $\mathcal{D}, \mathcal{E}$, configuration context $\mathcal{G}$, and names a and b such that $\mathcal{C} \equiv \mathcal{G}[\mathcal{D} \parallel \mathcal{E}]$ such that $\mathsf{depends}(a,b,\mathcal{D})$ and $\mathsf{depends}(b,a,\mathcal{E})$. However, this would require either $b = a$, violating linearity, or would require both a and b to be shared between $\mathcal{D}$ and $\mathcal{E}$, which is ill-typed by Lemma 9. □

Deadlock-freedom does not necessarily imply progress: just because we know that there are no cyclic dependencies does not necessarily mean that the system can always make a reduction step.

Our functional core satisfies a form of progress, under an environment containing only runtime names. In particular a term is either a value, an evaluation context focused on a concurrency construct, or can β-reduce. For convenience, let us define Ψ as a typing environment containing only runtime names:

$$\Psi ::= \cdot \mid \Psi, a : S$$

Lemma 10 (Progress (SGV terms)). *If $\Psi \vdash M : A$, then either:*

1. *M is a value*
2. *There exists some N such that $M \longrightarrow_M N$*
3. *There exist some E and N such that M may be written $E[N]$, where N is a communication and concurrency construct, i.e., **fork** V, **send** $V\,W$, **receive** V, or **wait** V.*

Proof. Standard: by induction on the derivation of $\Psi \vdash M : A$. □

We cannot reason inductively in a similar way about the progress of configurations. Instead, we define a *canonical form* which allows us to reason about the configuration as a whole. Let $\mathcal{M}$ denote configurations of the following form:

$$\mathcal{M} ::= \circ M_1' \parallel \cdots \parallel \circ M_n' \parallel \phi N$$

Definition 2 (Canonical Form). *A configuration $\mathcal{C}$ is in* canonical form *if it has names $a_1, \ldots a_n$ and threads $L_1, \ldots L_n$ and N such that $\mathcal{C}$ can be written:*

$$\mathcal{C} = (\nu a_1)(\circ L_1 \parallel \cdots \parallel (\nu a_n)(\circ L_n \parallel \mathcal{M}) \cdots)$$

where for each M_i, we have that $a_i \in \mathsf{fn}(M_i)$.

A canonical form makes the sharing of names in a configuration explicit. Note that canonical forms do not necessarily have to be unique: to take a simple example, suppose that we have a configuration $\mathcal{C} = (\nu a_1)(\nu a_2)(\circ M_1 \parallel \circ M_2 \parallel \bullet N)$, with $a_1 \in \mathsf{fn}(M_1)$, $a_2 \in \mathsf{fn}(M_2)$, and $a_1, a_2 \in \mathsf{fn}(N)$. Both:

1. $(\nu a_1)(\circ M_1 \parallel (\nu a_2)(\circ M_2 \parallel \bullet N))$
2. $(\nu a_2)(\circ M_2 \parallel (\nu a_1)(\circ M_1 \parallel \bullet N))$

are valid canonical forms of $\mathcal{C}$. Now, we can show that every well-typed configuration may be written in canonical form.

Theorem 4. *Suppose $\Gamma \vdash^{\phi} \mathcal{C}$. Then there exists some $\mathcal{C}'$ such that $\Gamma \vdash^{\phi} \mathcal{C}'$ and $\mathcal{C}'$ is in canonical form.*

Proof. By induction on the count of ν-bound names. Without loss of generality, assume that the ν-bound names of $\mathcal{C}$ are distinct. Let $\{a_i \mid 1 \leq i \leq n\}$ be the set of ν-bound names in $\mathcal{C}$ and let $\{\mathcal{D}_j \mid 1 \leq j \leq m\}$ be the set of threads in $\mathcal{C}$.

In the case that $n = 0$, by Lemma 7 we can safely commute the thread determining the thread flag such that it is the rightmost configuration, and associate parallel composition to the right using Lemma 8 to derive a well-typed canonical form.

In the case that $n \geq 1$, pick some a_i and $\mathcal{D}_j$ such that a_i is the only ν-bound name in $\mathsf{fn}(\mathcal{D}_j)$; Lemma 9 and a standard counting argument ensure that such a name and configuration exist. By the equivalence rules, there exists $\mathcal{E}$ such that $\mathcal{C} \equiv (\nu a_i)(\mathcal{D}_j \parallel \mathcal{E})$ and $\Gamma \vdash^{\phi} (\nu a_i)(\mathcal{D}_j \parallel \mathcal{E})$ (that a_i is the only ν-bound name in $\mathsf{fn}(\mathcal{D}_j)$ ensures well-typing). Moreover, we have that there exist $\Gamma' \subseteq \Gamma$ and S, such that $\Gamma', a_i : S \vdash^{\phi} \mathcal{E}$. By the induction hypothesis, there exists $\mathcal{E}'$ in canonical form such that $\Gamma', a_i : S \vdash^{\phi} \mathcal{E} \equiv \mathcal{E}'$. Let $\mathcal{C}' = (\nu a_i)(\mathcal{D}_j \parallel \mathcal{E}')$. By construction it holds that $\mathcal{C} \equiv \mathcal{C}'$, that $\Gamma \vdash^{\phi} \mathcal{C}'$, and that $\mathcal{C}'$ is in canonical form. □

A canonical form gives us a global view of the configuration, and we can use a canonical form to state a precise progress result. In particular, for open configurations, each thread must be blocked on either a variable in the typing environment, or a ν-bound name.

We show this result by defining the notion of *open progress*, allowing us to deconstruct canonical forms step-by-step, and to reason about the form of each subconfiguration.

Definition 3 (Open Progress)**.** *Suppose $\Psi \vdash^{\phi} \mathcal{C}$, where $\mathcal{C}$ is in canonical form and $\mathcal{C} \not\Longrightarrow$.*

We say that $\mathcal{C}$ satisfies open progress *if:*

1. *$\mathcal{C} = (\nu a)(\circ M \parallel \mathcal{D})$, where:*
   - *There exists a session type S and a session type $T \in \{S, \overline{S}\}$*
   - *$\Psi = \Psi_1, \Psi_2$*
   - *$\Psi_1, a : T \vdash^{\circ} \circ M$ where either $M = a$, or there exists some $b \in \mathsf{fn}(\Psi_1, a : T)$ such that blocked(b, M); and*
   - *$\Psi_2, a : \overline{T} \vdash^{\phi} \mathcal{D}$ and $\mathcal{D}$ satisfies open progress*
2. *$\mathcal{C} = \circ M' \parallel \mathcal{M}$, where:*
   - *There exists a session type S and a session type $T \in \{S, \overline{S}\}$*
   - *$\Psi = \Psi_1, \Psi_2, a : S^{\sharp}$*
   - *$\Psi_1, a : T \vdash^{\circ} \circ M'$ where either $M' = a$, or there exists some $b \in \mathsf{fn}(\Psi_1, a : T)$ and blocked(b, M'); and*
   - *$\Psi_2, a : \overline{T} \vdash^{\phi} \mathcal{M}$ and $\mathcal{M}$ satisfies open progress*
3. *$\mathcal{C} = \phi M$ where either M is a value, or blocked(M, a) for some $a \in \mathsf{fn}(\Psi)$.*

Our definition of open progress exploits the observation that canonical forms can be defined inductively. A canonical form is either a name restriction and child thread in parallel with a configuration in canonical form; a child thread without a name restriction in parallel with a configuration in canonical form but not including any name restrictions; or the thread determining the thread flag of the whole configuration.

Open progress details the conditions under which a well-typed configuration in canonical form cannot reduce. Moreover, the property states that each child thread is either blocked on the ν-bound name that immediately precedes the thread, or a name in the typing environment, and that the main thread must either be a value or blocked on a variable in the typing environment.

Every well-typed configuration satisfies open progress.

Lemma 11. *If $\Psi \vdash^{\phi} \mathcal{C}$ and $\mathcal{C} \not\Longrightarrow$, then $\mathcal{C}$ satisfies open progress.*

Proof. By induction on the derivation of $\Psi \vdash^{\phi} \mathcal{C}$. By the definition of canonical forms, $\mathcal{C}$ must either be of the form $(\nu a)(\circ M \parallel \mathcal{D})$, where $\mathcal{D}$ is in canonical form; $\circ M \parallel \mathcal{M}$; or ϕM.

Case $\mathcal{C} = (\nu a)(\circ M \parallel \mathcal{D})$, where $\mathcal{D}$ is in canonical form

Assumption:

$$\dfrac{\dfrac{\Psi_1, a : T \vdash^{\circ} \circ M \quad \Psi_2, a : \overline{T} \vdash^{\phi} \mathcal{D}}{\Psi_1, \Psi_2, a : S^{\sharp} \vdash^{\phi} \circ M \parallel \mathcal{D}}}{\Psi_1, \Psi_2 \vdash^{\phi} (\nu a)(\circ M \parallel \mathcal{D})}$$

where $T \in \{S, \overline{S}\}$, depending on whether T-Connect$_1$ or T-Connect$_2$ is used for parallel composition. Note that it *must* be the case that name a is split over the parallel composition, due to the requirement that $a_i \in \mathsf{fn}(M_i)$ for each a_i, M_i in a canonical form.

By T-Thread, we have that $\Psi_1, a : T \vdash M : \mathsf{End}_!$.

By Lemma 10, we have that either M is a value, or M can be written $E[N]$ where N is a communication or concurrency construct. If M is a value, then it must be the case that $M = a$ and $T = \mathsf{End}_!$, satisfying the definition of open pogress. If M is of the form $E[N]$, then N cannot be of the form **fork** M as **fork** can always reduce, so it must be the case that $\mathsf{blocked}(M, b)$ for some name $b \in \mathsf{fn}(\Psi_1, a : T)$. By the induction hypothesis, $\Psi_2, a : \overline{T} \vdash^{\phi} \mathcal{D}$ satisfies open progress, so $\Psi \vdash^{\phi} (\nu a)(\circ M \parallel \mathcal{D})$ satisfies open progress.

Case $\mathcal{C} = \circ M \parallel \mathcal{M}$

Similar to the previous case, however the name split by the parallel composition is not introduced by a name restriction and hence must be in $\mathsf{fn}(\Psi)$.

Case $\mathcal{C} = \phi M$

By similar reasoning to the first case, either M is a value, or M can be written $E[N]$ where N is a communication or concurrency construct that is not **fork**. Hence, there must exist some $a \in \mathsf{fn}(\Psi)$ such that $\mathsf{blocked}(M, a)$.

□

As an immediate corollary, we obtain a more global view of the structure of non-reducing, well-typed configurations.

Corollary 1 (Open Progress (SGV Configurations))**.** *Suppose $\Psi \vdash^{\phi} \mathcal{C}$, where $\mathcal{C}$ is in canonical form, and $\mathcal{C} \not\Longrightarrow$.*

Let $\mathcal{C} = (\nu a_1)(\circ M_1 \parallel \cdots \parallel (\nu a_n)(\circ M_m \parallel \phi\mathcal{M})\cdots)$, with $\mathcal{M} = M_{m+1} \parallel (M_{m+2} \parallel \cdots \parallel (M_n \parallel \phi N)\cdots)$.

Then:

1. *For each $M_i \in M_1, \ldots, M_n$, either $M_i = a_i$, or $\mathsf{blocked}(b, M_i)$ for some $b \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{fn}(\Psi)$*
2. *N is either a value, or $\mathsf{blocked}(b, N)$ for some $b \in \{a_j \mid 1 \leq j \leq n\} \cup \mathsf{fn}(\Psi)$*

We can substantially tighten this result when we consider only closed configurations with a main thread.

Corollary 2 (Closed Progress (Synchronous GV Configurations))**.** *Suppose $\cdot \vdash^{\bullet} \mathcal{C}$, where $\mathcal{C}$ is in canonical form, and $\mathcal{C} \not\Longrightarrow$.*

Let $\mathcal{C} = (\nu a_1)(\circ M_1 \parallel \cdots \parallel (\nu a_n)(\circ M_n \parallel \bullet N)\cdots)$. Then:

1. *For each M_i, either $M_i = a_i$, or $\mathsf{blocked}(a_i, M_i)$*
2. *$N = V$ for some value V.*

The closed progress result is substantially stronger than the open progress result. The reason for this is that each M_i must be blocked on the corresponding a_i: for example, M_1 must be either a value or blocked on a_1; M_2 cannot be blocked on a_1 since it could then reduce (by the typing of configurations, duality ensures that M_1 would be blocked on a send, and M_2 would be blocked on a receive, and thus the configuration could reduce by E-Comm), so must be blocked on a_2, and so on.

Our progress result for closed canonical forms can be strengthened even further should the main thread not contain any free names (recall that by T-Main, the main thread can have any type, including session types). A conservative way of ensuring that the main thread contains no free names is to require that the configuration is a *ground configuration*—a closed configuration where the type of the main thread does not contain any session types or function types. Any non-reducing ground configuration must be a value.

Theorem 5 (Global Progress (SGV)). *Suppose* $\cdot \vdash^{\bullet} \mathcal{C}$*, where* $\mathcal{C}$ *is a ground configuration in canonical form, and* $\not\Longrightarrow$*. Then* $\mathcal{C} = \bullet V$*.*

Proof. By Corollary 2, we have that $\mathcal{C}$ may be written $\mathcal{C} = (\nu a_1)(\circ M_1 \parallel \cdots \parallel (\nu a_n)(\circ M_n \parallel \bullet V))$ where each M_i either equal to or blocked on a_i. Since $\mathcal{C}$ is a ground configuration, $\mathsf{fn}(V) = \emptyset$ and so no child thread may be blocked or awaiting collection by **wait**. □

3.3.4 Confluence

We described the design of the operational semantics for Synchronous GV as a deterministic reduction relation on terms, and a nondeterministic reduction relation on configurations. While it is true that reduction on configurations is nondeterministic, it does in fact satisfy a strong notion of confluence, known as the *diamond property* [15]: if a term can reduce to two separate configurations, then the configurations are either equivalent, or the configurations will converge in a single step.

Theorem 6 (Diamond Property). *If* $\Gamma \vdash^{\phi} \mathcal{C}$*, and* $\mathcal{C} \Longrightarrow \mathcal{D}_1$*, and* $\mathcal{C} \Longrightarrow \mathcal{D}_2$*, then either* $\mathcal{D}_1 \equiv \mathcal{D}_2$*, or there exists some* $\mathcal{D}_3$ *such that* $\mathcal{D}_1 \Longrightarrow \mathcal{D}_3$ *and* $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$*.*

Proof. Firstly, observe that $\longrightarrow_{\mathsf{M}}$ is deterministic due to the call-by-value, left-to-right evaluation strategy imposed by evaluation contexts.

Thus, critical pairs only arise as a result of term reductions in different threads, or when performing communication actions on two separate channels. By linearity, these must occur independently and therefore may occur in any order. □

3.3.5 Termination

The linearity of GV leads to an elementary termination proof.

Theorem 7 (Termination). *If* $\Gamma \vdash^{\phi} \mathcal{C}$*, then there are no infinite* $\Longrightarrow$ *reductions from* $\mathcal{C}$*.*

Proof. We have an elementary proof due to linearity. Define the *measure* of a configuration to be the size of the sums of the ASTs of all threads. The measure of an AST strictly decreases by $\longrightarrow_{\mathsf{M}}$, since due to linearity, substitution may not duplicate a variable. The measure of a configuration strictly decreases under $\longrightarrow$, and remains invariant under $\equiv$, thus no infinite reduction sequences exist. □

Of course, this is unsurprising given that GV does not include unrestricted types or shared channels. If we were to introduce unrestricted types, a CPS translation along the lines of Lindley and Morris [133] would suffice; in the presence of shared channels, we would require a logical relations argument along the lines of Pérez et al. [170].

Modified Syntax

$$\text{Session Types} \qquad S ::= !A.S \mid ?A.S \mid \mathsf{End}$$
$$\text{Terms} \qquad L, M, N ::= \cdots \mid \mathbf{close}\, M$$

Modified typing rules for terms $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Fork}\ \frac{\Gamma \vdash M : (S \multimap \mathbf{1})}{\Gamma \vdash \mathbf{fork}\, M : \overline{S}} \qquad \textsc{T-Close}\ \frac{\Gamma \vdash M : \mathsf{End}}{\Gamma \vdash \mathbf{close}\, M : \mathbf{1}}$$

Duality

$$\overline{!A.S} = ?A.\overline{S} \qquad \overline{?A.S} = !A.\overline{S} \qquad \overline{\mathsf{End}} = \mathsf{End}$$

Additional Equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\circ() \parallel \mathcal{C} \equiv \mathcal{C}$$

Additional reduction rule for configurations $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$\textsc{E-Close} \qquad (\nu a)(\mathcal{F}[\mathbf{close}\, a] \parallel \mathcal{F}'[\mathbf{close}\, a]) \longrightarrow \mathcal{F}[()] \parallel \mathcal{F}'[()]$$

Modified typing rules for configurations $\boxed{\Gamma \vdash^{\phi} \mathcal{C}}$

$$\textsc{T-Thread}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash^{\circ} \circ M} \qquad \textsc{T-Mix}\ \frac{\Gamma_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2 \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}}$$

Figure 3.6: Modifications to GV to include a self-dual End type

3.4 Design Decisions

We have presented a particular base design for SGV, but have taken several technical design decisions. In this section, we discuss alternative designs, the consequences of the designs, and our reasoning for making the decisions we have.

3.4.1 Split Ends vs. Conditioned Ends

Previous work on session calculi, and implementations of languages with session types, often provide a single, self-dual End type to signify that a session has no more communication actions. In contrast, SGV provides two such types: $\mathsf{End}_!$ and $\mathsf{End}_?$, which are dual.

Interestingly, the choice of having separate $\mathsf{End}_!$ and $\mathsf{End}_?$ types ('split ends') or a single, self-dual End type ('conditioned ends') is a fundamental decision regarding the logical corre-

spondence. In fact Atkey et al. [13] show that having a single, self-dual End type corresponds to conflating the $\mathbf{1}$ and $\bot$ types in CP, which is equivalent to the addition of the logical Mix and Mix0 rules [79]:

$$\frac{}{\vdash \cdot}\ \textsc{Mix0} \qquad\qquad \frac{\vdash \Gamma \quad \vdash \Delta}{\vdash \Gamma, \Delta}\ \textsc{Mix}$$

Interpreted computationally, the addition of the Mix rule allows two independent processes to be composed in parallel, without being linked by a channel.

We will now investigate this logical observation from a language design perspective. Figure 3.6 shows the modifications to SGV in order to allow conditioned ends. In addition to replacing $\mathsf{End}_!$ and $\mathsf{End}_?$ with a single, self-dual End type, we replace the $\mathbf{wait}\,M$ construct with $\mathbf{close}\,M$, which takes a channel of type End and produces the unit value. Additionally, we modify the typing rule for $\mathbf{fork}\,M$ such that the function M takes a session of type S as before, but produces the unit value instead of a value of type $\mathsf{End}_!$. We also introduce a 'garbage collection' equivalence axiom which allows completed threads to be discarded.

The reduction rule E-Close provides perhaps the most stark justification as to why the Mix rule becomes necessary. Recall the reduction rule E-Wait:

$$(\nu a)(\mathcal{F}[\mathbf{wait}\,a] \parallel \circ a) \longrightarrow \mathcal{F}[()]$$

Here, we have that the **wait** construct synchronises with the finished thread, eliminating both the child thread and the name.

With E-Close, however, while the name restriction is eliminated, the child thread is not—and indeed, it is not safe to do so as the evaluation context $\mathcal{F}'$ may contain linear variables. The *communication link* is severed between the threads, but the threads should be able to evaluate regardless. As a result, we require the rule T-Mix, which allows processes to be composed in parallel even though they are not linked by a channel.

Justification. A self-dual End type is arguably easier to conceptualise as a developer. Nonetheless, for SGV (and indeed AGV which is described in Chapter 8), we choose to retain split ends. From a more theoretical perspective, we retain tighter logical connections. But from a more pragmatic point of view, the language with split ends has fewer reduction rules; the communication topology always remains a tree instead of a forest; and reasoning about the metatheory (in particular canonical forms) is more uniform.

That said, in Chapter 9, we discuss Exceptional GV which *does* have a self-dual End type. The justification for this is that in EGV, we will see that exceptions indeed necessarily partition the communication topology, thus requiring Mix; as conditioned ends and the Mix and Mix_0 rules are logically equivalent, it makes little sense to have one without the other.

Unrestricted Types $\boxed{\mathsf{un}(A)}\ \boxed{\mathsf{un}(\Gamma)}$

$$\frac{}{\mathsf{un}(\mathsf{End})} \qquad \frac{\mathsf{un}(A) \quad \mathsf{un}(B)}{\mathsf{un}(A \times B)} \qquad \frac{\mathsf{un}(A) \quad \mathsf{un}(B)}{\mathsf{un}(A + B)} \qquad \frac{\forall x : A \in \Gamma.\mathsf{un}(A)}{\mathsf{un}(\Gamma)}$$

Modified Typing Rules for Terms $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Var}\ \frac{\mathsf{un}(\Gamma)}{\Gamma, x : A \vdash x : A} \qquad \textsc{T-Name}\ \frac{\mathsf{un}(\Gamma)}{\Gamma, a : S \vdash a : S}$$

Figure 3.7: Modified term typing rules for affine End types

3.4.2 Linear Ends vs. Affine Ends

If we were to adopt conditioned ends, we could actually go a step further and eliminate **close** from the language altogether. Figure 3.7 shows modified typing rules which allow channels of type End to be discarded implicitly. The key idea is that we mark type End (as well as sums and products which only contain values of type End) as *unrestricted*, written $\mathsf{un}(A)$. We say that a typing environment is unrestricted, written $\mathsf{un}(\Gamma)$, if it only contains unrestricted types.

As a result, it is possible to type the following program:

$$\begin{aligned}&\textbf{let}\, s = \textbf{fork}\ (\lambda t.\textbf{let}\, t = \textbf{send}\, 5\, t\ \textbf{in}\ ())\ \textbf{in}\\ &\textbf{let}\, (x, s) = \textbf{receive}\, s\ \textbf{in}\\ &x\end{aligned}$$

We fork off a thread which sends 5 along channel t, which has type !Int.End. As End is unrestricted, it is not necessary to close t and as such it can be discarded implicitly. Similarly, after receiving along s, the variable has type End and thus may be discarded implicitly.

Treating End as affine increases the amount of concurrency, since threads need not synchronise when closing a channel.

Justification. Having affine End types only makes sense with conditioned ends, so we inherit the advantages and disadvantages of conditioned ends. These notwithstanding, implicitly discarding End can be convenient for a programmer, and indeed the Links implementation of session types has affine End types. That said, a purely linear calculus is closer to linear logic, and is more convenient theoretically as the typing rules are simpler, and reasoning about the metatheory is cleaner.

3.4.3 Reduction under Name Restrictions

In SGV, we elect to allow reduction on open configurations. To demonstrate, consider the rule E-Comm:

$$\text{E-Comm} \quad \mathcal{F}[\mathbf{send}\, V\, a] \parallel \mathcal{F}'[\mathbf{receive}\, a] \quad \longrightarrow \quad \mathcal{F}[a] \parallel \mathcal{F}'[(V,a)]$$

Note that the reduction rule does not include any name restrictions. An alternative formulation would be to include name restrictions in the reduction rule:

$$\text{E-CommNu} \quad (\nu a)(\mathcal{F}[\mathbf{send}\, V\, a] \parallel \mathcal{F}'[\mathbf{receive}\, a]) \quad \longrightarrow \quad (\nu a)(\mathcal{F}[a] \parallel \mathcal{F}'[(V,a)])$$

Formulating reduction in this way means that we need not have a reduction relation on session types or typing environments, and we therefore have a simpler preservation theorem. We can be sure that name a does not appear elsewhere in the configuration, and thus that the reduction rule does not require another process $\mathcal{C}$ to be present, as a consequence of linearity.

Consider again the statement of Theorem 1:

Theorem 1 *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then there exists some* $\Gamma \longrightarrow^{?} \Gamma'$ *such that* $\Gamma' \vdash^{\phi} \mathcal{D}$.

The clause stating that the environment must reduce is shaded.

It is worth investigating the revised theorem and proof case to understand in more detail why the theorem is simpler.

Theorem 8 (Preservation (SGV with E-CommNu))**.** *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then* $\Gamma \vdash^{\phi} \mathcal{D}$.

Proof. The only difference in the preservation proof is the case for E-Comm. Again, we consider the case where the first thread flag is $\bullet$ and the second thread flag is $\circ$, but the other combinations are similar.

Case E-CommNu

$$(\nu a)(\bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a]) \quad \longrightarrow \quad (\nu a)(\bullet E[a] \parallel \circ E'[(V,a)])$$

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma_1,\Gamma_2,a:S \vdash E[\mathbf{send}\, V\, a] : C}{\Gamma_1,\Gamma_2,a:S \vdash^{\bullet} \bullet E[\mathbf{send}\, V\, a]} \quad \dfrac{\Gamma_3,a:\overline{S} \vdash E'[\mathbf{receive}\, a] : \mathsf{End}_!}{\Gamma_3,a:\overline{S} \vdash^{\circ} \circ E'[\mathbf{receive}\, a]}}{\Gamma_1,\Gamma_2,\Gamma_3,a:S^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a]}}{\Gamma_1,\Gamma_2,\Gamma_3 \vdash^{\bullet} (\nu a)(\bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a])}$$

By Lemma 2:

$$\dfrac{\Gamma_2 \vdash V : A \quad a : !A.S' \vdash a : !A.S'}{\Gamma_2, a : !A.S' \vdash \mathbf{send}\, V\, a : S'}$$

Also by Lemma 2:

$$\frac{a : ?A.\overline{S'} \vdash a : ?A.\overline{S'}}{a : ?A.\overline{S'} \vdash \mathbf{receive}\, a : (A \times \overline{S'})}$$

This reasoning allows us to refine our original derivation:

$$\frac{\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : !A.S' \vdash E[\mathbf{send}\, V\, a] : C}{\Gamma_1, \Gamma_2, a : !A.S' \vdash^{\bullet} \bullet E[\mathbf{send}\, V\, a]} \quad \dfrac{\Gamma_3, a : ?A.\overline{S'} \vdash E'[\mathbf{receive}\, a] : \mathsf{End}_!}{\Gamma_3, a : ?A.\overline{S'} \vdash^{\circ} \circ E'[\mathbf{receive}\, a]}}{\Gamma_1, \Gamma_2, \Gamma_3, a : (!A.S')^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a]}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash^{\bullet} (\nu a)(\bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a])}$$

By Lemma 3, $\Gamma_1, a : S' \vdash E[a] : C$, and $\Gamma_2, \Gamma_3 \vdash E'[(V, a)] : \mathsf{End}_!$ (that Γ_2, Γ_3 is well-defined follows from the fact that the two environments are disjoint).

Recomposing:

$$\frac{\dfrac{\dfrac{\Gamma_1, a : S' \vdash E[a] : C}{\Gamma_1, a : S' \vdash^{\bullet} \bullet E[a]} \quad \dfrac{\Gamma_2, \Gamma_3, a : \overline{S'} \vdash E'[(V, a)] : \mathsf{End}_!}{\Gamma_2, \Gamma_3, a : \overline{S'} \vdash^{\circ} \circ E'[(V, a)]}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S'^{\sharp} \vdash^{\bullet} \bullet E[a] \parallel \circ E'[(V, a)]}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash^{\bullet} (\nu a)(\bullet E[a] \parallel \circ E'[(V, a)])}$$

as required.

□

In short, the type of the name a changes as a result of reduction. In open terms, this must be reflected in the theorem statement, which requires reduction relations on session types and typing environments. When communication occurs under name restrictions however, name a is not present in the typing environment to start, and thus we do not need to consider a reduction relation on types.

Justification. It is a matter of taste as to which formulation is "better". On the one hand, it is worthwhile to strive for fewer judgements and simpler theorem statements. Additionally, the principal cut reductions of CP all take place under name restrictions, and thus having GV reductions also take place under name restrictions is more in line with the language's logical foundations.

On the other hand, omitting name restrictions simplifies the reduction rules and is closer to the π-calculus; indeed, without this fairly technical justification for the inclusion of name restrictions, readers familiar with the π-calculus may find the formulation confusing. Additionally, the reduction on typing environments makes the notion of *session fidelity*—the property

that programs exhibit the communication behaviour prescribed by their session type—more explicit.

We opt not to require communication reduction to happen under a name restriction, primarily to avoid surprise from readers more familiar with the π-calculus, and to reduce syntactic noise in extensions where there are more reduction rules. The alternative formulation would be a perfectly valid choice, however.

3.5 Conclusion

In this chapter, we have described SGV, a well-behaved core functional language based on the GV calculus as originally proposed by Wadler [213] and expanded upon by Lindley and Morris [132]. Our formulation is similar to that of Lindley and Morris [132], but removes features such as weak explicit substitutions that are only used to prove a logical correspondence with CP. We have also seen the strong metatheory enjoyed by SGV, and proofs of its correctness.

We build upon SGV later in the thesis as a basis for asynchrony and exception handling.

Part II

Mixing Metaphors: Actors as Channels and Channels as Actors

Chapter 4

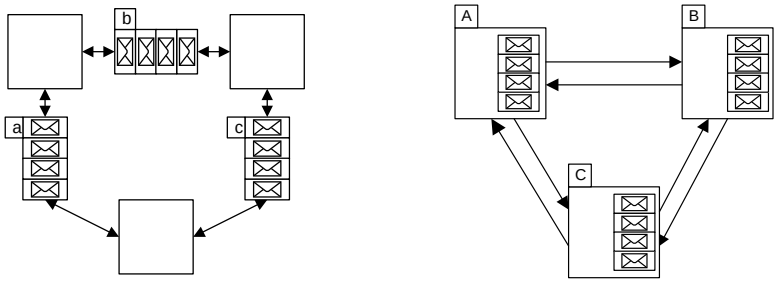
Type-parameterised Channels and Actors

4.1 Introduction

When comparing channels (as used by Go) and actors (as implemented in Erlang), one runs into an immediate mixing of metaphors. The words themselves do not refer to comparable entities!

In languages such as Go, anonymous processes pass messages via named channels, whereas in languages such as Erlang, named processes accept messages from an associated mailbox. A channel is either a named rendezvous point or buffer, whereas an actor is a process. We should really be comparing named processes (actors) with anonymous processes, and buffers tied to a particular process (mailboxes) with buffers that can link any process to any process (channels). Nonetheless, we will stick with the popular names, even if it is as inapposite as comparing TV channels with TV actors.

Figure 4.1 compares asynchronous channels with actors. On the left, three anonymous processes communicate via channels named a, b, c. On the right, three processes named A, B, C



(a) Asynchronous Channels

(b) Actors

Figure 4.1: Channels and Actors

send messages to each others' associated mailboxes. Actors are necessarily asynchronous, allowing non-blocking sends and buffering of received values, whereas channels can either be asynchronous or synchronous (rendezvous-based). Indeed, Go provides both synchronous *and* asynchronous channels, and libraries such as `core.async` [88] provide library support for asynchronous channels. However, this is not the only difference: each actor has a single buffer which only it can read—its *mailbox*—whereas asynchronous channels are free-floating buffers that can be read by any process with a reference to the channel.

Channel-based languages such as Go enjoy a firm basis in process calculi such as CSP [92] and the π-calculus [144]. It is easy to type channels, either with simple types (see [189], p. 231) or more complex systems such as session types [77, 94, 97]. Actor-based languages such as Erlang are seen by many as the "gold standard" for distributed computing due to their support for fault tolerance through supervision hierarchies [10, 33].

Both models are popular with developers, with channel-based languages and frameworks such as Go, `core.async`, and Hopac [101]; and actor-based languages and frameworks such as Erlang, Elixir, and Akka.

4.2 Motivation

This chapter provides a formal account of actors and channels as implemented in programming languages. Our motivation for a formal account is threefold: it helps clear up confusion; it clarifies results that have been described informally by putting practice into theory; and it provides a foundation for future research.

Confusion. There is often confusion over the differences between channels and actors. For example, the following questions appear on StackOverflow and Quora respectively:

> If I wanted to port a Go library that uses Goroutines, would Scala be a good choice because its inbox/[A]kka framework is similar in nature to coroutines?" [112], and

> "I don't know anything about [the] actor pattern however I do know goroutines and channels in Go. How are [the] two related to each other?" [102]

In academic circles, the term *actor* is often used imprecisely. For instance, Albert et al. [8] refer to Go as an actor language. Similarly, Harvey [85] refers to his language Ensemble as actor-based. Ensemble is a language specialised for writing distributed applications running on heterogeneous platforms. It is actor-based to the extent that it has lightweight, addressable, single-threaded processes, and forbids co-ordination via shared memory. However, Ensemble communicates using channels as opposed to mailboxes so we would argue that it is channel-based (with actor-like features) rather than actor-based.

Putting practice into theory. The success of actor-based languages is largely due to their support for *supervision*. A popular pattern for writing actor-based applications is to arrange processes in *supervision hierarchies* [10], where *supervisor* processes restart child processes should they fail. Projects such as Proto.Actor [181] emulate actor-style programming in a channel-based language in an attempt to gain some of the benefits, by associating queues with processes. Hopac [101] is a channel-based library for F#, inspired by Concurrent ML [186]. The documentation [3] contains a comparison with actors, including an implementation of a simple actor-based communication model using Hopac-style channels, as well as an implementation of Hopac-style channels using an actor-based communication model. By comparing the two, we provide a formal model for the underlying techniques, and study properties arising from the translations.

A foundation for future research. Traditionally, actor-based languages have had untyped mailboxes. More recent advancements such as TAkka [86], Akka Typed [7], and Typed Actors [206] have added types to mailboxes in order to gain additional safety guarantees. Our formal model provides a foundation for these innovations, characterises why naïvely adding types to mailboxes is problematic, and provides a core language for future experimentation.

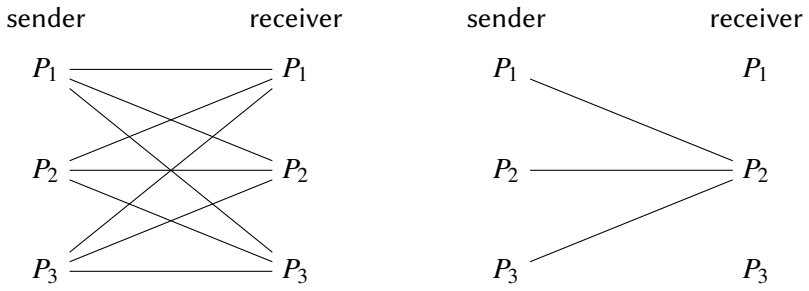
4.3 Our approach

We define two concurrent λ-*calculi*, describing *asynchronous* channels and type-parameterised actors, define translations between them, and then discuss various extensions.

Why the λ calculus? Our common framework is that of a simply-typed concurrent λ-calculus: that is, a λ-calculus equipping a term language with primitives for communication and concurrency, as well as a language of *configurations* to model concurrent behaviour. We work with the λ-calculus rather than a process calculus for two reasons: first, the simply-typed λ-calculus has a well-behaved core with a strong metatheory (for example, confluent reduction and strong normalisation), as well as a direct propositions-as-types correspondence with logic. We can therefore modularly extend the language, knowing which properties remain; typed process calculi typically do not have such a well-behaved core.

Second, we are ultimately interested in functional programming languages; the λ calculus is the canonical choice for studying such extensions.

Why asynchronous channels? While actor-based languages must be asynchronous by design, channels may be either synchronous (requiring a rendezvous between sender and receiver) or asynchronous (where sending happens immediately). In this part of the thesis, we consider asynchronous channels since actor communication must be asynchronous, and



(a) Channel (b) Mailbox

Figure 4.2: Mailboxes as pinned channels

it is possible to emulate asynchronous channels using synchronous channels [186]. We could adopt synchronous channels, use these to encode asynchronous channels, and then do the translations. We elect not to since it complicates the translations, and we argue that the distinction between synchronous and asynchronous communication is not *the* defining difference between the two models.

4.4 Summary of results

We identify four key differences between the models, which are exemplified by the formalisms and the translations: process addressability, the restrictiveness of communication patterns, the granularity of typing, and the ability to control the order in which messages are processed.

Process addressability. In channel-based systems, processes are *anonymous*, whereas channels are named. In contrast, in actor-based systems, processes are named.

Restrictiveness of communication patterns. Communication over full-duplex channels is more liberal than communication via mailboxes, as shown in Figure 4.2. Figure 4.2a shows the communication patterns allowed by a single channel: each process P_i can use the channel to communicate with every other process. Conversely, Figure 4.2b shows the communication patterns allowed by a mailbox associated with process P_2: while any process can send to the mailbox, only P_2 can read from it. Viewed this way, it is apparent that the restrictions imposed on the communication behaviour of actors are exactly those captured by Merro and Sangiorgi's localised π-calculus [140].

Readers familiar with actor-based programming may be wondering whether such a characterisation is too crude, as it does not account for processing messages out-of-order. Fear not—we show in Chapter 6 that our actor calculus can simulate this functionality.

Restrictiveness of communication patterns is not necessarily a bad thing: while it is easy to

distribute actors, *delegation* of asynchronous channels is more involved, requiring a distributed algorithm [106]. Associating mailboxes with addressable processes also helps with structuring applications for reliability [33].

Granularity of typing. As a result of the fact that each process has a single incoming message queue, mailbox types tend to be less precise; in particular, they are most commonly variant types detailing all of the messages that can be received. Naïvely implemented, this gives rise to the *type pollution problem*, which we describe further in §4.5.

Message ordering. Channels and mailboxes are ordered message queues, but there is no inherent ordering between messages on two different channels. Channel-based languages allow a user to specify from which channel a message should be received, whereas processing messages out-of-order can be achieved in actor languages using a selective receive construct.

The remainder of this part of the thesis captures these differences both in the design of the formalisms, and the techniques used in the encodings and extensions.

4.5 Channels and actors side-by-side

Let us consider the example of a concurrent stack. A concurrent stack carrying values of type A can receive a command to push a value onto the top of the stack, or to pop a value and return it to the process making the request. Assuming a standard encoding of algebraic datatypes, we define a type $\mathsf{Operation}(\mathsf{A}) = \mathsf{Push}(A) \mid \mathsf{Pop}(B)$ (where $B = \mathsf{ChanRef}(A)$ for channels, and $\mathsf{ActorRef}(A)$ for actors) to describe operations on the stack, and $\mathsf{Option}(\mathsf{A}) = \mathsf{Some}(A) \mid \mathsf{None}$ to handle the possibility of popping from an empty stack.

Figure 4.3 shows the stack implemented using channels (Figure 4.3a) and using actors (Figure 4.3b). Each implementation uses a common core language based on the simply-typed λ-calculus extended with recursion, lists, and sums.

At first glance, the two stack implementations seem remarkably similar. Each:

1. Waits for a command
2. Case splits on the command, and either:
   - Pushes a value onto the top of the stack, or;
   - Takes the value from the head of the stack and returns it in a response message
3. Loops with an updated state.

The main difference is that chanStack is parameterised over a channel *ch*, and retrieves a value from the channel using **take** *ch*. Conversely, actorStack retrieves a value from its mailbox using the nullary primitive **receive**.

```
chanStack(ch) ≜ rec loop(st).
    let cmd ⇐ take ch in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resCh) ↦
        case st {
          [ ] ↦ give (None) resCh;
                loop [ ]
          x :: xs ↦ give (Some(x)) resCh;
                     loop xs   }
    }

chanClient(stackCh) ≜
  give (Push(5)) stackCh;
  let resCh ⇐ newCh in
  give (Pop(resCh)) stackCh;
  take resCh

chanMain ≜
  let stackCh ⇐ newCh in
  fork (chanStack(stackCh) [ ]);
  chanClient(stackCh)
```

(a) Channel-based stack

```
actorStack ≜ rec loop(st).
    let cmd ⇐ receive in
    case cmd {
      Push(v) ↦ loop(v :: st)
      Pop(resPid) ↦
        case st {
          [ ] ↦ send (None) resPid;
                loop [ ]
          x :: xs ↦ send (Some(x)) resPid;
                     loop xs   }
    }

actorClient(stackPid) ≜
  send (Push(5)) stackPid;
  let selfPid ⇐ self in
  send (Pop(selfPid)) stackPid;
  receive

actorMain ≜
  let stackPid ⇐ spawn (actorStack [ ]) in
  actorClient(stackPid)
```

(b) Actor-based stack

Figure 4.3: Concurrent stacks using channels and actors

```
chanClient2(intStackCh, stringStackCh) ≜
  let intResCh ⇐ newCh in
  let strResCh ⇐ newCh in
  give (Pop(intResCh)) intStackCh;
  let res1 ⇐ take intResCh in
  give (Pop(strResCh)) stringStackCh;
  let res2 ⇐ take strResCh in
  return (res1, res2)
```

```
actorClient2(intStackPid, stringStackPid) ≜
  let selfPid ⇐ self in
  send (Pop(selfPid)) intStackPid;
  let res1 ⇐ receive in
  send (Pop(selfPid)) stringStackPid;
  let res2 ⇐ receive in
  return (res1, res2)
```

Figure 4.4: Clients interacting with multiple stacks

Let us now consider functions which interact with the stacks. The chanClient function sends commands over the *stackCh* channel, and begins by pushing 5 onto the stack. Next, it creates a channel *resCh* to be used to receive the result and sends this in a request, before retrieving the result from the result channel using **take**. In contrast, actorClient performs a similar set of steps, but sends its process ID (retrieved using **self**) in the request instead of creating a new channel; the result is then retrieved from the mailbox using **receive**.

Type pollution. The differences become more prominent when considering clients which interact with multiple stacks of different types, as shown in Figure 4.4. Here, chanClient2 creates new result channels for integers and strings, sends requests for the results, and creates a pair of type $(\mathsf{Option}(\mathsf{Int}) \times \mathsf{Option}(\mathsf{String}))$. The actorClient2 function attempts to do something similar, but cannot create separate result channels. Consequently, the actor must be able to handle messages either of type $\mathsf{Option}(\mathsf{Int})$ *or* type $\mathsf{Option}(\mathsf{String})$, meaning that the final pair has type $(\mathsf{Option}(\mathsf{Int}) + \mathsf{Option}(\mathsf{String})) \times (\mathsf{Option}(\mathsf{Int}) + \mathsf{Option}(\mathsf{String}))$.

Additionally, it is necessary to modify actorStack to use the correct injection into the actor type when sending the result; for example an integer stack would have to send a value $\mathbf{inl}\,(\mathsf{Some}(5))$ instead of simply $\mathsf{Some}(5)$. This *type pollution* problem can be addressed through the use of subtyping [86], or synchronisation abstractions such as futures [55].

In the remainder of this chapter, we will precisely characterise typed channels and actors by introducing two small concurrent λ-calculi: λ_{ch} and λ_{act}.

4.6 λ_{ch}: A concurrent λ-calculus for channels

We begin by introducing λ_{ch}, a concurrent λ-calculus extended with asynchronous channels. To concentrate on the core differences between channel- and actor-style communication, we begin with small calculi; note that these do not contain all features (such as lists, sums, and recursion) needed to express the examples in §4.5.

Syntax

$$
\begin{array}{lrcl}
\text{Types} & A, B & ::= & \mathbf{1} \mid A \to B \mid \mathsf{ChanRef}(A) \\
\text{Values} & U, V, W & ::= & x \mid \lambda x.M \mid () \\
\text{Computations} & L, M, N & ::= & V\,W \\
& & \mid & \textbf{let}\, x \Leftarrow M\, \textbf{in}\, N \mid \textbf{return}\, V \\
& & \mid & \textbf{fork}\, M \mid \textbf{give}\, V\, W \mid \textbf{take}\, V \mid \textbf{newCh}
\end{array}
$$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}$$

Computation typing rules $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-App}\ \frac{\Gamma \vdash V : A \to B \quad \Gamma \vdash W : A}{\Gamma \vdash V\,W : B} \qquad \textsc{T-EffLet}\ \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B}{\Gamma \vdash \textbf{let}\, x \Leftarrow M\, \textbf{in}\, N : B} \qquad \textsc{T-Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \textbf{return}\, V : A}$$

$$\textsc{T-Give}\ \frac{\Gamma \vdash V : A \quad \Gamma \vdash W : \mathsf{ChanRef}(A)}{\Gamma \vdash \textbf{give}\, V\, W : \mathbf{1}} \qquad \textsc{T-Take}\ \frac{\Gamma \vdash V : \mathsf{ChanRef}(A)}{\Gamma \vdash \textbf{take}\, V : A} \qquad \textsc{T-Fork}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \textbf{fork}\, M : \mathbf{1}}$$

$$\textsc{T-NewCh}\ \frac{}{\Gamma \vdash \textbf{newCh} : \mathsf{ChanRef}(A)}$$

Figure 4.5: Syntax and typing rules for λ_{ch} terms and values

4.6.1 Syntax and typing of terms

Figure 4.5 gives the syntax and typing rules of λ_{ch}, a λ-calculus based on fine-grain call-by-value [131]: terms are partitioned into values and computations. Key to this formulation are two constructs: $\textbf{return}\, V$ represents a computation that has completed, whereas $\textbf{let}\, x \Leftarrow M\, \textbf{in}\, N$ evaluates M to $\textbf{return}\, V$, substituting V for x in M. Fine-grain call-by-value is convenient since it makes evaluation order explicit and, unlike A-normal form [68], is closed under reduction.

Types consist of the unit type $\mathbf{1}$, function types $A \to B$, and channel reference types $\mathsf{ChanRef}(A)$ which can be used to communicate along a channel of type A. We write $\textbf{let}\, x = V\, \textbf{in}\, M$ for $(\lambda x.M)\, V$ and $M; N$ for $\textbf{let}\, x \Leftarrow M\, \textbf{in}\, N$, where x is fresh.

Communication and concurrency for channels. The $\textbf{give}\, V\, W$ operation sends value V along channel W, while $\textbf{take}\, V$ retrieves a value from a channel V. Assuming an extension

Runtime Syntax

Names	a, b, c
Variables and names	$\alpha ::= x \mid a$
Values	$U, V, W ::= \cdots \mid a$
Term typing environments	$\Gamma ::= \cdots \mid \Gamma, a : \mathsf{ChanRef}(A)$
Runtime typing environments	$\Delta ::= \cdot \mid \Delta, a : A$
Evaluation contexts	$E ::= [\,] \mid \mathbf{let}\, x \Leftarrow E\, \mathbf{in}\, M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid a(\overrightarrow{V}) \mid M$
Configuration contexts	$\mathcal{G} ::= [\,] \mid \mathcal{G} \parallel \mathcal{C} \mid (\nu a)\mathcal{G}$

Additional value typing rule $\boxed{\Gamma \vdash V : A}$

$$\textsc{T-Name}\ \frac{a : \mathsf{ChanRef}(A) \in \Gamma}{\Gamma \vdash a : \mathsf{ChanRef}(A)}$$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{T-Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C} \quad \Gamma; \Delta_2 \vdash \mathcal{D}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C} \parallel \mathcal{D}} \qquad \textsc{T-Chan}\ \frac{\Gamma, a : \mathsf{ChanRef}(A); \Delta, a{:}A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

$$\textsc{T-Buf}\ \frac{(\Gamma, a : \mathsf{ChanRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ChanRef}(A); a : A \vdash a(\overrightarrow{V})} \qquad \textsc{T-Thread}\ \frac{\Gamma \vdash M : A}{\Gamma; \cdot \vdash M}$$

Figure 4.6: λ_{ch} runtime typing

of the language with integers and arithmetic operators, we can define a function $\mathsf{neg}(c)$ which receives a number n along channel c and replies with the negation of n as follows:

$$\mathsf{neg}(c) \triangleq \mathbf{let}\, n \Leftarrow \mathbf{take}\, c\, \mathbf{in}\, \mathbf{let}\, negN \Leftarrow (-n)\, \mathbf{in}\, \mathbf{give}\, negN\, c$$

The **fork** M operation spawns a new process to evaluate term M. The operation returns the unit value, and therefore it is not possible to interact with the process directly. The **newCh** operation creates a new channel. Note that channel creation is decoupled from process creation, meaning that a process can have access to multiple channels.

4.6.2 Operational semantics

Runtime names. We let a, b, c range over *runtime names*, which do not occur in closed programs and are only introduced as a result of evaluation. We let α range over variables and runtime names, which is useful when defining progress results and translations.

Reduction on terms $\boxed{M \longrightarrow_{\mathsf{M}} N}$

$$(\lambda x.M)\, V \longrightarrow_{\mathsf{M}} M\{V/x\}$$
$$\textbf{let}\ x \Leftarrow \textbf{return}\ V\ \textbf{in}\ M \longrightarrow_{\mathsf{M}} M\{V/x\}$$
$$E[M_1] \longrightarrow_{\mathsf{M}} E[M_2] \quad (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)$$

Structural congruence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel D) \parallel \mathcal{E} \qquad \mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \text{ if } a \notin \mathsf{fv}(\mathcal{C})$$
$$(\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

Reduction on configurations $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$\begin{array}{lrcll}
\text{E-Give} & E[\textbf{give}\, W\, a] \parallel a(\overrightarrow{V}) & \longrightarrow & E[\textbf{return}\ ()] \parallel a(\overrightarrow{V} \cdot W) & \\
\text{E-Take} & E[\textbf{take}\, a] \parallel a(W \cdot \overrightarrow{V}) & \longrightarrow & E[\textbf{return}\, W] \parallel a(\overrightarrow{V}) & \\
\text{E-Fork} & E[\textbf{fork}\, M] & \longrightarrow & E[\textbf{return}\ ()] \parallel M & \\
\text{E-NewCh} & E[\textbf{newCh}] & \longrightarrow & (\nu a)(E[\textbf{return}\, a] \parallel a(\varepsilon)) & (a \text{ is fresh}) \\
\text{E-LiftM} & M_1 & \longrightarrow & M_2 \quad (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2) & \\
\text{E-Lift} & \mathcal{G}[\mathcal{C}_1] & \longrightarrow & \mathcal{G}[\mathcal{C}_2] \quad (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2) &
\end{array}$$

Figure 4.7: Reduction on λ_{ch} terms and configurations

Configurations. The concurrent behaviour of λ_{ch} is given by a nondeterministic reduction relation on *configurations* (Figure 4.6). Configurations consist of parallel composition ($\mathcal{C} \parallel \mathcal{D}$), restrictions ($(\nu a)\mathcal{C}$), computations ($M$), and buffers ($a(\overrightarrow{V})$, where $\overrightarrow{V} = V_1 \cdot \ldots \cdot V_n$).

Evaluation contexts. Evaluation contexts E are simplified due to fine-grain call-by-value. We also define configuration contexts $\mathcal{G}$, allowing reduction under parallel compositions and name restrictions.

Reduction. Figure 4.7 shows the reduction rules for λ_{ch}. Reduction is defined as a deterministic reduction on terms ($\longrightarrow_{\mathsf{M}}$) and a nondeterministic reduction relation on configurations ($\longrightarrow$).

We define $\equiv$ as the smallest congruence relation satisfying the equivalence axioms in Figure 4.7. These axioms capture scope extrusion, reordering of name restrictions, and the commutativity and associativity of parallel composition.

Relation notation. Given a relation R, we write R^+ for its transitive closure, and R^* for its reflexive, transitive closure. We use juxtaposition for the combination of relations.

We write $\Longrightarrow$ for the relation $\equiv\!\!\longrightarrow\!\!\equiv$ (that is, reduction modulo equivalence).

Runtime typing. We extend the value typing rules with rule T-Name, which types runtime names.

To ensure that buffers are well-scoped and contain values of the correct type, we define typing rules on configurations (Figure 4.6). The judgement $\Gamma;\Delta \vdash \mathcal{C}$ states that under environments Γ and Δ, $\mathcal{C}$ is well-typed. Γ is a typing environment for terms, whereas Δ is a linear typing environment for configurations, mapping names a to channel types A. Linearity in Δ is a technical device to ensure that a configuration $\mathcal{C}$ under a name restriction $(\nu a)\mathcal{C}$ contains exactly one buffer with name a.

Note that T-Chan extends both Γ *and* Δ, adding an (unrestricted) *reference* into Γ and the *capability* to type a buffer into Δ. T-Par states that $\mathcal{C} \parallel \mathcal{D}$ is typeable if $\mathcal{C}$ and $\mathcal{D}$ are typeable under disjoint linear environments, and T-Buf states that under a term environment Γ and a singleton linear environment $a{:}A$, it is possible to type a buffer $a(\overrightarrow{V})$ if $\Gamma \vdash V_i{:}A$ for all $V_i \in \overrightarrow{V}$. For example, $(\nu a)(a(\overrightarrow{V}))$ is well-typed, but $(\nu a)(a(\overrightarrow{V}) \parallel a(\overrightarrow{W}))$ and $(\nu a)(\mathbf{return}\ ())$ are not:

$$\frac{\frac{(\Gamma, a : \mathsf{ChanRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ChanRef}(A); a : A \vdash a(\overrightarrow{V})}}{\Gamma;\cdot \vdash (\nu a)(a(\overrightarrow{V}))}$$

$$\frac{\frac{(\Gamma, a : \mathsf{ChanRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ChanRef}(A); a : A \vdash a(\overrightarrow{V})} \quad \Gamma, a : \mathsf{ChanRef}(A);\cdot \nvdash a(\overrightarrow{W})}{\Gamma;\cdot \nvdash (\nu a)(a(\overrightarrow{V}) \parallel a(\overrightarrow{W}))}$$

$$\frac{\Gamma, a : \mathsf{ChanRef}(A); a : A \nvdash \mathbf{return}\ ()}{\Gamma;\cdot \nvdash (\nu a)(\mathbf{return}\ ())}$$

Properties of the term language. We begin by defining lemmas to allow us to manipulate evaluation contexts, which prove useful when proving preservation for both term and configuration reduction.

Lemma 12 allows us to type the subterm of an evaluation context.

Lemma 12 (Subterm typeability (λ_{ch} terms))**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma \vdash E[M] : A$*, then there exists some type* B *and a subderivation* $\mathbf{D}'$ *of* $\mathbf{D}$ *such that* $\mathbf{D}'$ *concludes* $\Gamma \vdash M : B$*, and the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to the position of the hole in* $\mathbf{D}$*.*

Proof. By induction on the structure of E, noting that evaluation contexts are not defined under λ abstractions. □

Next, Lemma 13 shows how we may replace the subterm of an evaluation context.

Lemma 13 (Subterm replacement (λ_{ch} terms))**.** *If:*

1. $\mathbf{D}$ *is a derivation of* $\Gamma \vdash E[M] : A$
2. $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma \vdash M : B$
3. $\Gamma \vdash N : B$ *for some* N
4. *The position of* $\mathbf{D}'$ *corresponds to that of the hole in* E

then $\Gamma \vdash E[N] : A$.

With these, we can prove that reduction on terms preserves typing.

Lemma 14 (Preservation (λ_{ch} terms))**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M N$*, then* $\Gamma \vdash N : A$.

Proof. Standard; by induction on the derivation of $M \longrightarrow_{\mathsf{M}} N$, using Lemma 12, Lemma 13, and a substitution lemma. □

Pure terms enjoy progress under the term reduction relation ($\longrightarrow_{\mathsf{M}}$). We let Ψ range over typing environments which contain only channel references:

$$\Psi ::= \cdot \mid \Psi, a : \mathsf{ChanRef}(A)$$

Lemma 15 (Progress (λ_{ch} terms))**.** *If* $\Psi \vdash M : A$*, then either:*

1. $M = \textbf{return}\, V$ *for some value* V*; or*
2. *there exists some* N *such that* $M \longrightarrow_M N$*; or*
3. *there exist unique* E, N *such that* M *can be written* $E[N]$*, where* N *is a communication or concurrency primitive (i.e.,* $\textbf{give}\, V\, W$, $\textbf{take}\, V$, $\textbf{fork}\, M$*, or* $\textbf{newCh}$*)*

Proof. Standard: by induction on the derivation of $\Psi \vdash M : A$. □

Reduction on configurations. Communication and concurrency is captured by reduction on configurations. The E-Give rule reduces $\textbf{give}\, W\, a$ in parallel with a buffer $a(\overrightarrow{V})$ by adding the value W onto the end of the buffer. The E-Take rule reduces $\textbf{take}\, a$ in parallel with a non-empty buffer by returning the first value in the buffer. The E-Fork rule reduces $\textbf{fork}\, M$ by spawning a new thread M in parallel with the parent process. The E-NewCh rule reduces $\textbf{newCh}$ by creating an empty buffer and returning a fresh name for that buffer.

We now need to define lemmas to allow us to work with configuration contexts $\mathcal{G}$. These are slightly different to term contexts in that the configuration context $(\nu a)G$ binds a name a, and the configuration context $\mathcal{G} \parallel \mathcal{C}$ splits the runtime typing context.

Lemma 16 (Subterm Typeability (λ_{ch} configurations))**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma; \Delta \vdash \mathcal{G}[\mathcal{C}]$*, then there exist some* Γ' *and* Δ' *such that* $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma'; \Delta' \vdash \mathcal{C}$*, and the position* $\mathbf{D}$ *in* $\mathbf{D}'$ *corresponds to the position of the hole in* $\mathcal{G}$.

Proof. By induction on the structure of $\mathcal{G}$. □

Lemma 17 (Subterm Replacement (λ_{ch} configurations))**.** *If:*

1. $\mathbf{D}$ *is a derivation of* $\Gamma;\Delta \vdash \mathcal{G}[\mathcal{C}]$
2. $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma';\Delta' \vdash \mathcal{C}$
3. *The position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* $\mathcal{G}$
4. $\Gamma';\Delta' \vdash \mathcal{D}$

then $\Gamma;\Delta \vdash \mathcal{G}[\mathcal{D}]$.

Proof. By induction on the structure of $\mathcal{G}$. □

Equivalence and reduction preserve the typeability of configurations.

Lemma 18. *If* $\Gamma;\Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$ *for some configuration* $\mathcal{D}$*, then* $\Gamma;\Delta \vdash \mathcal{D}$.

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$. □

Theorem 9 (Preservation (λ_{ch} configurations))**.** *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ *then* $\Gamma;\Delta \vdash \mathcal{C}_2$.

Proof. By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$. We prove case E-Give here; full details can be found in Appendix B.

Case E-Give

$$E[\mathbf{give}\, W\, a] \parallel a(\overrightarrow{V}) \quad \longrightarrow \quad E[\mathbf{return}\, ()] \parallel a(\overrightarrow{V} \cdot W)$$

Assumption:

$$\frac{\dfrac{\Gamma \vdash E[\mathbf{give}\, W\, a] : B}{\Gamma;\cdot \vdash E[\mathbf{give}\, W\, a]} \qquad \dfrac{(\Gamma \vdash V_i : A)_i}{\Gamma;a : A \vdash a(\overrightarrow{V})}}{\Gamma;a : A \vdash E[\mathbf{give}\, W\, a] \parallel a(\overrightarrow{V})}$$

Note that by T-Chan, Γ must contain $a : \mathsf{ChanRef}(A)$.
By Lemma 12, we have:

$$\frac{\Gamma \vdash W : A \quad \Gamma \vdash a : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathbf{give}\, W\, a : \mathbf{1}}$$

By Lemma 13, we have that $\Gamma \vdash E[\mathbf{return}\, ()] : B$.
By T-Buf:

$$\frac{(\Gamma \vdash V_i : A)_i \quad \Gamma \vdash W : A}{\Gamma;a : A \vdash a(\overrightarrow{V} \cdot W)}$$

Recomposing, we have:

$$\dfrac{\dfrac{\Gamma \vdash E[\mathbf{return}\ ()] : B}{\Gamma;\cdot \vdash E[\mathbf{return}\ ()]} \qquad \dfrac{(\Gamma \vdash V_i : A)_i \quad \Gamma, a : \mathsf{ChanRef}(A) \vdash W : A}{\Gamma; a : A \vdash a(\overrightarrow{V} \cdot W)}}{\Gamma; a : A \vdash E[\mathbf{return}\ ()] \parallel a(\overrightarrow{V} \cdot W)}$$

□

As a corollary of Lemma 18 and Theorem 9, we have that reduction modulo equivalence preserves typeability of configurations.

Corollary 3 (Preservation: Reduction modulo equivalence (λ_{ch}))**.** *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \Longrightarrow \mathcal{C}_2$ *then* $\Gamma;\Delta \vdash \mathcal{C}_2$.

4.6.3 Progress and canonical forms

While it is possible to prove deadlock-freedom in systems with more discerning type systems based on linear logic [132, 213] or those using channel priorities [167], more liberal calculi such as λ_{ch} and λ_{act} allow deadlocked configurations. We thus define a form of progress which does not preclude deadlock; to help with proving a progress result, it is useful to consider the notion of a *canonical form* in order to allow us to reason about the configuration as a whole.

Definition 4 (Canonical form (λ_{ch}))**.** *A configuration* $\mathcal{C}$ *is in* canonical form *if it can be written* $(\nu a_1)\ldots(\nu a_n)(M_1 \parallel \ldots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \ldots \parallel a_n(\overrightarrow{V_n}))$.

Well-typed open configurations can be written in a form similar to canonical form, but without bindings for names already in the environment. An immediate corollary is that well-typed closed configurations can always be written in a canonical form.

Lemma 19. *If* $\Gamma;\Delta \vdash \mathcal{C}$ *with* $\Delta = a_1 : A_1, \ldots, a_k : A_k$*, then there exists a* $\mathcal{C}' \equiv \mathcal{C}$ *such that* $\mathcal{C}' = (\nu a_{k+1})\ldots(\nu a_n)(M_1 \parallel \ldots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \ldots \parallel a_n(\overrightarrow{V_n}))$.

Proof. By induction on the derivation of $\Gamma;\Delta \vdash \mathcal{C}$. T-Buf and T-Thread follow immediately. T-Chan follows by the induction hypothesis. For T-Par, by the induction hypothesis we have that each subconfiguration can be written in canonical form; the composition of the two can be written in canonical form by using scope extrusion to move all name restrictions to the front of the configuration, and the commutativity and associativity of parallel composition to move terms and buffers to the required positions. □

Corollary 4 (Preservation modulo equivalence (λ_{ch} configurations))**.** *If* $\cdot;\cdot \vdash \mathcal{C}$*, then there exists some* $\mathcal{C}' \equiv \mathcal{C}$ *such that* $\mathcal{C}'$ *is in canonical form.*

Armed with the notion of a canonical form, we can now state that the only situation in which a well-typed closed configuration cannot reduce further is if all threads are either blocked or fully evaluated. Let a *leaf configuration* be a configuration without subconfigurations, i.e., a term or a buffer.

Theorem 10 (Weak progress (λ_{ch} configurations))**.**
Let $\cdot;\cdot \vdash C$, $C \not\Longrightarrow$, *and let* $C' = (\nu a_1)\ldots(\nu a_n)(M_1 \parallel \ldots \parallel M_m \parallel a_1(\overrightarrow{V_1}) \parallel \ldots \parallel a_n(\overrightarrow{V_n}))$ *be a canonical form of* C. *Then every leaf of* C *is either:*

1. *A buffer* $a_i(\overrightarrow{V_i})$;
2. *A fully-reduced term of the form* **return** V, *or;*
3. *A term of the form* $E[\textbf{take}\, a_i]$, *where* $\overrightarrow{V_i} = \varepsilon$.

Proof. By Lemma 15, we know that each M_i is either of the form **return** V, or that there exist some E, N such that M_i can be written $E[N]$ where N is a communication or concurrency primitive. It cannot be the case that $N = \textbf{fork}\, N'$ or $N = \textbf{newCh}$, since both can reduce. Let us now consider **give** and **take**, blocked on a variable α. As we are considering closed configurations, a blocked term must be blocked on a ν-bound name a_i, and as per the canonical form, we have that there exists some buffer $a_i(\overrightarrow{V_i})$. Consequently, **give** $V\, a_i$ can always reduce via E-Give. A term **take** a_i can reduce by E-Take if $\overrightarrow{V_i} = W \cdot \overrightarrow{V_i'}$; the only remaining case is where $\overrightarrow{V_i} = \varepsilon$, satisfying (3). □

Syntax

$$
\begin{array}{lrcl}
\text{Types} & A, B, C & ::= & \mathbf{1} \mid A \to^{C} B \mid \mathsf{ActorRef}(A) \\
\text{Values} & U, V, W & ::= & x \mid \lambda x.M \mid () \\
\text{Computations} & L, M, N & ::= & V\,W \\
& & \mid & \textbf{let}\, x \Leftarrow M\, \textbf{in}\, N \mid \textbf{return}\, V \\
& & \mid & \textbf{spawn}\, M \mid \textbf{send}\, V\, W \mid \textbf{receive} \mid \textbf{self}
\end{array}
$$

Value typing rules $\boxed{\Gamma \vdash V : A}$

$$\textsc{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : A \mid C \vdash M : B}{\Gamma \vdash \lambda x.M : A \to^{C} B} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \mathbf{1}}$$

Computation typing rules $\boxed{\Gamma \mid B \vdash M : A}$

$$\textsc{T-App}\ \frac{\Gamma \vdash V : A \to^{C} B \qquad \Gamma \vdash W : A}{\Gamma \mid C \vdash V\,W : B} \qquad \textsc{T-EffLet}\ \frac{\Gamma \mid C \vdash M : A \qquad \Gamma, x : A \mid C \vdash N : B}{\Gamma \mid C \vdash \textbf{let}\, x \Leftarrow M\, \textbf{in}\, N : B}$$

$$\textsc{T-EffReturn}\ \frac{\Gamma \vdash V : A}{\Gamma \mid B \vdash \textbf{return}\, V : A}$$

$$\textsc{T-Send}\ \frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : \mathsf{ActorRef}(A)}{\Gamma \mid B \vdash \textbf{send}\, V\, W : \mathbf{1}} \qquad \textsc{T-Recv}\ \frac{}{\Gamma \mid A \vdash \textbf{receive} : A} \qquad \textsc{T-Spawn}\ \frac{\Gamma \mid A \vdash M : B}{\Gamma \mid C \vdash \textbf{spawn}\, M : \mathsf{ActorRef}(A)}$$

$$\textsc{T-Self}\ \frac{}{\Gamma \mid A \vdash \textbf{self} : \mathsf{ActorRef}(A)}$$

Figure 4.8: Syntax and typing rules for λ_{act}

4.7 λ_{act}: A concurrent λ-calculus for actors

In this section, we introduce λ_{act}, a core language describing actor-based concurrency. There are many variations of actor-based languages (by the taxonomy of De Koster et al; [56], λ_{act} is *process-based*), but each have named processes associated with a mailbox.

Typed channels are well-established, whereas typed actors are less so, partly due to the type pollution problem. Nonetheless, Akka Typed [7] aims to replace untyped Akka actors, so studying a typed actor calculus is of practical relevance.

Following Erlang, we provide an explicit **receive** operation to allow an actor to retrieve a message from its mailbox: unlike **take** in λ_{ch}, **receive** takes no arguments, so it is necessary to use a simple *type-and-effect system* [78] to track the mailbox type. We treat mailboxes as a FIFO queues to keep λ_{act} as small as possible, as opposed to considering behaviours or selective receive. This is orthogonal to the core model of communication, as we show in Chapter 6.

4.7.1 Syntax and typing of terms

Figure 4.8 shows the syntax and typing rules for λ_{act}. Type $\mathsf{ActorRef}(A)$ is an *actor reference* or process ID, and allows messages to be sent to an actor. As for communication and concurrency primitives, **spawn** M spawns a new actor to evaluate a computation M; **send** $V\ W$ sends a value V to an actor referred to by reference W; **receive** receives a value from the actor's mailbox; and **self** returns an actor's own process ID.

Function arrows $A \to^{C} B$ are annotated with a type C which denotes the type of the mailbox of the actor evaluating the term. As an example, consider a function which receives an integer and converts it to a string (assuming a function intToString):

$$\mathsf{recvAndShow} \triangleq \lambda().\mathbf{let}\ x \Leftarrow \mathbf{receive}\ \mathbf{in}\ \mathsf{intToString}(x)$$

Such a function would have type $\mathbf{1} \to^{\mathsf{Int}} \mathsf{String}$, and as an example could not be applied in an actor with a mailbox typethat could only receive booleans. Nevertheless, it is perfectly possible to send such function to another actor, even if it cannot be used.

Again, we work in the setting of fine-grain call-by-value; the distinction between values and computations is helpful when reasoning about the metatheory. We have two typing judgements: the standard judgement on values $\Gamma \vdash V : A$, and a judgement $\Gamma \mid B \vdash M : A$ which states that a term M has type A under typing context Γ, and can receive values of type B. The typing of **receive** and **self** depends on the type of the actor's mailbox.

4.7.2 Operational semantics

Figure 4.9 shows the syntax of λ_{act} evaluation contexts, as well as the syntax and typing rules of λ_{act} configurations. Evaluation contexts for terms and configurations are similar to λ_{ch}. The

Runtime syntax

Names	a, b	
Variables and names	α	$::= x \mid a$
Values	U, V, W	$::= \cdots \mid a$
Term typing environments	Γ	$::= \cdots \mid \Gamma, a : \mathsf{ActorRef}(A)$
Runtime typing environments	Δ	$::= \cdot \mid \Delta, a : A$
Evaluation contexts	E	$::= [\,] \mid \textbf{let}\ x \Leftarrow E\ \textbf{in}\ M$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	$::= \mathcal{C} \parallel \mathcal{D} \mid (\nu a)\mathcal{C} \mid \langle a, M, \overrightarrow{V} \rangle$
Configuration contexts	$\mathcal{G}$	$::= [\,] \mid \mathcal{G} \parallel \mathcal{C} \mid (\nu a)\mathcal{G}$

Additional value typing rule $\boxed{\Gamma \vdash V : A}$

$$\textsc{T-Name}\ \frac{a : \mathsf{ActorRef}(A) \in \Gamma}{\Gamma \vdash a : \mathsf{ActorRef}(A)}$$

Typing rules for configurations $\boxed{\Gamma; \Delta \vdash \mathcal{C}}$

$$\textsc{T-Par}\ \frac{\Gamma; \Delta_1 \vdash \mathcal{C} \quad \Gamma; \Delta_2 \vdash \mathcal{D}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{C} \parallel \mathcal{D}} \qquad \textsc{T-Pid}\ \frac{\Gamma, a : \mathsf{ActorRef}(A); \Delta, a : A \vdash \mathcal{C}}{\Gamma; \Delta \vdash (\nu a)\mathcal{C}}$$

$$\textsc{T-Actor}\ \frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash M : B \quad (\Gamma, a : \mathsf{ActorRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, M, \overrightarrow{V} \rangle}$$

Figure 4.9: λ_{act} evaluation contexts and configurations

primary difference from λ_{ch} is the actor configuration $\langle a, M, \overrightarrow{V} \rangle$, which can be read as "an actor with name a evaluating term M, with a mailbox consisting of values $\overrightarrow{V}$". Whereas a term M is itself a configuration in λ_{ch}, a term in λ_{act} must be evaluated as part of an actor configuration in order to support context-sensitive operations such as receiving from the mailbox. We again stratify the reduction rules into functional reduction on terms, and reduction on configurations. The typing rules for λ_{act} configurations ensure that all values contained in an actor mailbox are well-typed with respect to the mailbox type, and that a configuration $\mathcal{C}$ under a name restriction $(\nu a)\mathcal{C}$ contains an actor with name a. Figure 4.10 shows the reduction rules for λ_{act}.

Again, reduction on terms preserves typing, and the functional fragment of λ_{act} enjoys progress. We start once more by defining lemmas which allow us to manipulate evaluation contexts.

Lemma 20 (Subterm typeability (λ_{act} terms))**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma \mid B \vdash E[M] : A$*, then there exists some type* C *and a subderivation* $\mathbf{D}'$ *of* $\mathbf{D}$ *such that* $\mathbf{D}'$ *concludes* $\Gamma \mid B' \vdash M : C$*, and the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to the position of the hole in* $\mathbf{D}$*.*

Proof. By induction on the structure of E. □

Lemma 21 (Subterm replacement (λ_{act} terms))**.** *If:*

1. $\mathbf{D}$ *is a derivation of* $\Gamma \mid B \vdash E[M] : A$
2. $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma \mid B' \vdash M : C$
3. $\Gamma \mid B' \vdash N : C$ *for some* N
4. *The position of* $\mathbf{D}'$ *corresponds to that of the hole in* E

then $\Gamma \mid B \vdash E[N] : A$*.*

Proof. By induction on the structure of E. □

Lemma 22 (Preservation (λ_{act} terms))**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M N$*, then* $\Gamma \vdash N : A$*.*

Proof. By induction on the derivation of $M \longrightarrow_{\text{M}} N$ □

Again, we let Ψ denote typing environments containing only actor references.

$$\Psi = \cdot \mid \Psi, a : \mathsf{ActorRef}(A)$$

Lemma 23 (Progress (λ_{act} terms))**.** *If* $\Psi \mid B \vdash M : A$*, then either:*

1. $M =$ ***return*** V *for some value* V*; or*
2. *there exists some* N *such that* $M \longrightarrow_M N$*; or*

Reduction on terms $\boxed{M \longrightarrow_{\mathrm{M}} N}$

$$
\begin{aligned}
(\lambda x.M)\,V &\longrightarrow_{\mathrm{M}} M\{V/x\} \\
\mathbf{let}\ x \Leftarrow \mathbf{return}\ V\ \mathbf{in}\ M &\longrightarrow_{\mathrm{M}} M\{V/x\} \\
E[M] &\longrightarrow_{\mathrm{M}} E[M'] \quad (\text{if } M \longrightarrow_{\mathrm{M}} M')
\end{aligned}
$$

Configuration equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad (\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

$$\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}) \quad \text{if } a \notin \mathrm{fv}(\mathcal{C})$$

Reduction on configurations $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$
\begin{aligned}
&\text{E-Spawn} && \langle a, E[\mathbf{spawn}\,M], \overrightarrow{V}\rangle \longrightarrow (\nu b)(\langle a, E[\mathbf{return}\,b], \overrightarrow{V}\rangle \parallel \langle b, M, \varepsilon\rangle) \\
& && \quad (b \text{ is fresh}) \\
&\text{E-Send} && \langle a, E[\mathbf{send}\,U\,b], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W}\rangle \longrightarrow \langle a, E[\mathbf{return}\,()], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W} \cdot U\rangle \\
&\text{E-SendSelf} && \langle a, E[\mathbf{send}\,W\,a], \overrightarrow{V}\rangle \longrightarrow \langle a, E[\mathbf{return}\,()], \overrightarrow{V} \cdot W\rangle \\
&\text{E-Self} && \langle a, E[\mathbf{self}], \overrightarrow{V}\rangle \longrightarrow \langle a, E[\mathbf{return}\,a], \overrightarrow{V}\rangle \\
&\text{E-Receive} && \langle a, E[\mathbf{receive}], W \cdot \overrightarrow{V}\rangle \longrightarrow \langle a, E[\mathbf{return}\,W], \overrightarrow{V}\rangle \\
&\text{E-Lift} && \mathcal{G}[\mathcal{C}_1] \longrightarrow \mathcal{G}[\mathcal{C}_2] \quad (\text{if } \mathcal{C}_1 \longrightarrow \mathcal{C}_2) \\
&\text{E-LiftM} && \langle a, M_1, \overrightarrow{V}\rangle \longrightarrow \langle a, M_2, \overrightarrow{V}\rangle \quad (\text{if } M_1 \longrightarrow_{\mathrm{M}} M_2)
\end{aligned}
$$

Figure 4.10: Reduction on λ_{act} terms and configurations

3. *there exist E, N such that M can be written as $E[N]$, where N is a communication or concurrency primitive (i.e. $\mathbf{spawn}\,N$, $\mathbf{send}\,V\,W$, $\mathbf{receive}$, or $\mathbf{self}$).*

Proof. Standard: by induction on the derivation of $\Psi \mid B \vdash M : A$. □

Reduction on configurations. While λ_{ch} makes use of separate constructs to create new processes and channels, λ_{act} uses a single construct $\mathbf{spawn}\,M$ to spawn a new actor with an empty mailbox to evaluate term M. Communication happens directly between actors instead of through an intermediate entity: as a result of evaluating $\mathbf{send}\,V\,a$, the value V will be appended directly to the end of the mailbox of actor a. E-SendSelf allows an actor to send a message to itself; an alternative would be to decouple mailboxes from the definition of actors, but this complicates both the configuration typing rules and belies the intuition. E-Self returns the name of the current process, and E-Receive retrieves the head value of a non-empty mailbox.

The lemmas for subconfiguration typeability and replacement are identical to those in λ_{ch}.

Lemma 24 (Subterm Typeability (λ_{act} configurations))**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma;\Delta \vdash \mathcal{G}[\mathcal{C}]$*, then there exist some* Γ' *and* Δ' *such that* $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma';\Delta' \vdash \mathcal{C}$*, and the position* $\mathbf{D}$ *in* $\mathbf{D}'$ *corresponds to the position of the hole in* $\mathcal{G}$*.*

Proof. By induction on the structure of $\mathcal{G}$. □

Lemma 25 (Subterm Replacement (λ_{act} configurations))**.** *If:*

1. $\mathbf{D}$ *is a derivation of* $\Gamma;\Delta \vdash \mathcal{G}[\mathcal{C}]$
2. $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma';\Delta' \vdash \mathcal{C}$
3. $\Gamma';\Delta' \vdash \mathcal{D}$
4. *The position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* $\mathcal{G}$

then $\Gamma;\Delta \vdash \mathcal{G}[\mathcal{D}]$*.*

Proof. By induction on the structure of $\mathcal{G}$. □

As before, typing is preserved modulo structural congruence and under reduction.

Lemma 26. *If* $\Gamma;\Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$ *for some* $\mathcal{D}$*, then* $\Gamma;\Delta \vdash \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$. □

Theorem 11 (Preservation (λ_{act} configurations))**.** *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then* $\Gamma;\Delta \vdash \mathcal{C}_2$*.*

Proof. By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$. We show the case for E-Send here; the remaining cases can be found in Appendix B.

Case E-Send

$$\text{E-Send} \quad \langle a, E[\mathbf{send}\, U\, b], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W}\rangle \longrightarrow \langle a, E[\mathbf{return}\, ()], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W} \cdot U\rangle$$

Let $\Gamma = \Gamma', a : \mathsf{ActorRef}(A), b : \mathsf{ActorRef}(B)$ for some Γ'.

Assumption:

$$\frac{\dfrac{\Gamma \mid A \vdash E[\mathbf{send}\, U\, b] : C \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash \langle a, E[\mathbf{send}\, U\, b], \overrightarrow{V}\rangle} \qquad \dfrac{\Gamma \mid B \vdash M : C' \quad (\Gamma \vdash W_i : B)_i}{\Gamma; b : B \vdash \langle b, M, \overrightarrow{W}\rangle}}{\Gamma; a : A, b : B \vdash \langle a, E[\mathbf{send}\, U\, b], \overrightarrow{V}\rangle \parallel \langle b, M, \overrightarrow{W}\rangle}$$

Note that $a : \mathsf{ActorRef}(A)$ and $b : \mathsf{ActorRef}(B)$ must be in Γ due to rule T-Actor.

By Lemma 20:

$$\frac{\Gamma \vdash U : B \quad \Gamma \vdash b : \mathsf{ActorRef}(B)}{\Gamma \mid A \vdash \mathbf{send}\, U\, b : \mathbf{1}}$$

By Lemma 21, we have that $\Gamma \mid A \vdash E[\textbf{return}\ ()] : C$.

By T-Actor:

$$\frac{\Gamma \mid B \vdash M : C' \qquad (\Gamma \vdash W_i : B)_i \quad \Gamma \vdash U : B}{\Gamma; b : B \vdash \langle b, M, \overrightarrow{W} \cdot U \rangle}$$

Recomposing:

$$\frac{\dfrac{\Gamma \mid A \vdash E[\textbf{return}\ ()] : C \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash \langle a, E[\textbf{return}\ ()], \overrightarrow{V} \rangle} \qquad \dfrac{\Gamma \mid B \vdash M : C' \qquad (\Gamma \vdash W_i : B)_i \quad \Gamma \vdash U : B}{\Gamma; b : B \vdash \langle b, M, \overrightarrow{W} \cdot U \rangle}}{\Gamma; a : A, b : B \vdash \langle a, E[\textbf{return}\ ()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot U \rangle}$$

as required.

□

As a corollary of Lemma 26 and Theorem 11, we have that reduction modulo equivalence preserves typeability of configurations.

Corollary 5 (Preservation modulo equivalence (λ_{act} configurations))**.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \Longrightarrow \mathcal{D}$*, then* $\Gamma; \Delta \vdash \mathcal{D}$*.*

4.7.3 Progress and canonical forms

Again, we cannot guarantee deadlock-freedom for λ_{act}. Instead, we proceed by defining a canonical form, and characterising the form of progress that λ_{act} enjoys. The technical development follows that of λ_{ch}.

Definition 5 (Canonical form (λ_{act}))**.** *A* λ_{act} *configuration* $\mathcal{C}$ *is in* canonical form *if* $\mathcal{C}$ *can be written* $(\nu a_1) \ldots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1} \rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle)$.

Lemma 27. *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\Delta = a_1 : A_1, \ldots a_k : A_k$*, then there exists* $\mathcal{C}' \equiv \mathcal{C}$ *such that* $\mathcal{C}' = (\nu a_{k+1}) \ldots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1} \rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle)$.

Proof. Similar to the proof of Lemma 19. □

As in λ_{ch}, it follows as a corollary of Lemma 27 that closed configurations can be written in canonical form. We can therefore classify the notion of progress enjoyed by λ_{act}.

Corollary 6. *If* $\cdot; \cdot \vdash \mathcal{C}$*, then there exists some* $\mathcal{C}' \equiv \mathcal{C}$ *such that* $\mathcal{C}'$ *is in canonical form.*

Theorem 12 (Weak progress (λ_{act} configurations))**.**
Let $\cdot;\cdot \vdash C$, $C \not\Longrightarrow$, *and let* $C' = (\nu a_1)\ldots(\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \ldots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$ *be a canonical form of* C. *Each actor with name* a_i *is either of the form* $\langle a_i, \textbf{return}\, W, \overrightarrow{V_i}\rangle$ *for some value* W, *or* $\langle a_i, E[\textbf{receive}], \varepsilon\rangle$.

Proof. Similar to the proof of Theorem 10. Lemma 23 shows that the term language enjoys progress. Only communication and concurrency actions cannot reduce using a term reduction; $\textbf{spawn}\, M$ can always reduce, and the canonical form ensures that $\textbf{send}\, V\, b$ can always reduce as b must be in scope. The term **receive** will only fail to reduce if the mailbox is empty, satisfying the final condition of the theorem. □

4.8 Summary

In this chapter, we have informally introduced programming with languages using type-parameterised channels and type-parameterised actors, and have formally characterised the two paradigms by distilling them into two concurrent λ-calculi: λ_{ch} and λ_{act} respectively. We have proven that both languages satisfy preservation, and characterised the notion of progress that each language enjoys. We have also begun to informally compare and contrast the two models.

In Chapter 5, we investigate the relationship between the two models more formally by encoding actors using channels, and channels using actors.

Chapter 5

Actors as Channels and Channels as Actors

In Chapter 4, we introduced two paradigms of typed communication-centric programming languages: channel-based languages, where anonymous processes communicate over named channels, and languages inspired by the actor model of computation, where named processes communicate directly by sending messages to message queues. We distilled these paradigms down to two minimal concurrent λ-calculi: λ_{ch}, a concurrent λ-calculus with typed asynchronous channels, and λ_{act}, a concurrent λ-calculus with type-parameterised actors.

In this chapter, we provide formal comparisons of the two models by showing type- and semantics-preserving translations between λ_{ch} and λ_{act} in both directions.

5.1 From λ_{act} to λ_{ch}

The key idea in translating λ_{act} into λ_{ch} is to emulate a mailbox using a channel, and to pass the channel as an argument to each function. The translation on terms is parameterised over the channel name, which is used to implement context-dependent operations (i.e., **receive** and **self**). Consider again recvAndShow.

$$\mathsf{recvAndShow} \triangleq \lambda().\mathbf{let}\ x \Leftarrow \mathbf{receive}\ \mathbf{in}\ \mathsf{intToString}(x)$$

A possible configuration would be an actor evaluating recvAndShow (), with some name a and mailbox with values $\overrightarrow{V}$, under a name restriction for a.

$$(\nu a)(\langle a, \mathsf{recvAndShow}\,(), \overrightarrow{V}\rangle)$$

The translation on terms takes a channel name ch as a parameter. As a result of the translation (assuming that $[\![\,\mathsf{intToString}\,]\!]_{ch} = \mathsf{intToString}$ in λ_{act}), we have that:

$$[\![\,\mathsf{recvAndShow}\,()\,]\!]_{ch} = \mathbf{let}\ x \Leftarrow \mathbf{take}\ ch\ \mathbf{in}\ \mathsf{intToString}(x)$$

Translation on types

$$[\![\,\mathsf{ActorRef}(A)\,]\!] = \mathsf{ChanRef}([\![\,A\,]\!]) \qquad [\![\,A \to^{C} B\,]\!] = [\![\,A\,]\!] \to \mathsf{ChanRef}([\![\,C\,]\!]) \to [\![\,B\,]\!] \qquad [\![\,\mathbf{1}\,]\!] = \mathbf{1}$$

Translation on values

$$[\![\,x\,]\!] = x \qquad [\![\,a\,]\!] = a \qquad [\![\,\lambda x.M\,]\!] = \lambda x.\lambda ch.([\![\,M\,]\!]_{ch}) \qquad [\![\,()\,]\!] = ()$$

Translation on computation terms

$$[\![\,\mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N\,]\!]_{ch} = \mathbf{let}\ x \Leftarrow ([\![\,M\,]\!]_{ch})\ \mathbf{in}\ [\![\,N\,]\!]_{ch}$$

$$[\![\,V\,W\,]\!]_{ch} = \mathbf{let}\ f \Leftarrow ([\![\,V\,]\!]\,[\![\,W\,]\!])\ \mathbf{in}\ f\ ch$$

$$[\![\,\mathbf{return}\ V\,]\!]_{ch} = \mathbf{return}\ [\![\,V\,]\!]$$

$$[\![\,\mathbf{self}\,]\!]_{ch} = \mathbf{return}\ ch$$

$$[\![\,\mathbf{receive}\,]\!]_{ch} = \mathbf{take}\ ch$$

$$[\![\,\mathbf{spawn}\ M\,]\!]_{ch} = \mathbf{let}\ chMb \Leftarrow \mathbf{newCh}\ \mathbf{in}\ \mathbf{fork}\,([\![\,M\,]\!]_{chMb});\ \mathbf{return}\ chMb$$

$$[\![\,\mathbf{send}\ V\,W\,]\!]_{ch} = \mathbf{give}\,([\![\,V\,]\!])\,([\![\,W\,]\!])$$

Translation on configurations

$$[\![\,\mathcal{C} \parallel \mathcal{D}\,]\!] = [\![\,\mathcal{C}\,]\!] \parallel [\![\,\mathcal{D}\,]\!] \qquad [\![\,(\nu a)\mathcal{C}\,]\!] = (\nu a)\,[\![\,\mathcal{C}\,]\!] \qquad [\![\,\langle a, M, \overrightarrow{V}\rangle\,]\!] = a([\![\,\overrightarrow{V}\,]\!]) \parallel ([\![\,M\,]\!]_a)$$

Figure 5.1: Translation from λ_{act} into λ_{ch}

with the corresponding configuration $(\nu a)(a([\![\,\overrightarrow{V}\,]\!]) \parallel [\![\,\mathsf{recvAndShow}\,()\,]\!]_a)$. The values from the mailbox are translated pointwise and form the contents of a buffer with name a. The translation of recvAndShow is provided with the name a which is used to emulate **receive**.

5.1.1 Translation (λ_{act} to λ_{ch})

Figure 5.1 shows the formal translation from λ_{act} into λ_{ch}. Of particular note is the translation on terms: $[\![-]\!]_{ch}$ translates a λ_{act} term into a λ_{ch} term using a channel with name ch to emulate a mailbox. An actor reference is represented as a channel reference in λ_{ch}; we emulate sending a message to another actor by writing to the channel emulating the recipient's mailbox. Key to translating λ_{act} into λ_{ch} is the translation of function arrows $A \to^{C} B$; the effect annotation C is replaced by a second parameter $\mathsf{ChanRef}(C)$, which is used to emulate the local mailbox. Names, variables, and the unit value translate to themselves. The translation of λ abstractions takes an additional parameter denoting the channel used to emulate operations on a mailbox. Given parameter ch, the translation function for terms emulates **receive** by taking a value from ch, and emulates **self** by returning ch.

Though the translation is straightforward, it is a *global* translation [66], as all functions must be modified in order to take the mailbox channel as an additional parameter.

5.1.2 Properties of the translation

The translation on terms and values preserves typing. We extend the translation function pointwise to typing environments: $[\![\alpha_1 : A_1, \ldots, \alpha_n : A_n]\!] = \alpha_1 : [\![A_1]\!], \ldots, \alpha_n : [\![A_n]\!]$.

Lemma 28 ($[\![-]\!]$ preserves typing (terms and values))**.**

1. *If* $\Gamma \vdash V : A$ *in* λ_{act}*, then* $[\![\Gamma]\!] \vdash [\![V]\!] : [\![A]\!]$ *in* λ_{ch}*.*
2. *If* $\Gamma \mid B \vdash M : A$ *in* λ_{act}*, then* $[\![\Gamma]\!], \alpha : \mathit{ChanRef}([\![B]\!]) \vdash [\![M]\!]_\alpha : [\![A]\!]$ *in* λ_{ch}*.*

Proof. By simultaneous induction on the derivations of $\Gamma \vdash V{:}A$ and $\Gamma \mid B \vdash M{:}A$.

We show the cases for T-Abs and T-Receive. The remaining cases are similar.

Case T-Abs

Assumption:

$$\frac{\Gamma, x : A \mid C \vdash M : B}{\Gamma \vdash \lambda x.M : A \to^C B}$$

By the IH (Premise 2), $[\![\Gamma]\!], x : [\![A]\!], ch : \mathsf{ChanRef}([\![C]\!]) \vdash [\![M]\!]_{ch} : [\![B]\!]$. Thus by T-Abs in λ_{ch}, we have

$$\frac{\dfrac{[\![\Gamma]\!], x : [\![A]\!], ch : \mathsf{ChanRef}([\![C]\!]) \vdash [\![M]\!]_{ch} : [\![B]\!]}{[\![\Gamma]\!], x : [\![A]\!] \vdash \lambda ch.[\![M]\!]_{ch} : \mathsf{ChanRef}([\![C]\!]) \to [\![B]\!]}}{[\![\Gamma]\!] \vdash \lambda x.\lambda ch.[\![M]\!]_{ch} : [\![A]\!] \to \mathsf{ChanRef}([\![C]\!]) \to [\![B]\!]}$$

as required.

Case T-Receive

Assumption:

$$\frac{}{\Gamma \mid A \vdash \textbf{receive}}$$

By the definition of the translation on **receive**, we can show

$$\frac{\dfrac{}{[\![\Gamma]\!], ch : [\![A]\!] \vdash ch : [\![A]\!]}}{[\![\Gamma]\!], ch : [\![A]\!] \vdash \textbf{take}\ ch : [\![A]\!]}$$

as required.

□

To state an operational correspondence result, we also define a translation on configurations (Figure 5.1). The translations on parallel composition and name restrictions are homomorphic. An actor configuration $\langle a, M, \overrightarrow{V} \rangle$ is translated as a buffer $a([\![\overrightarrow{V}]\!])$, (writing $[\![\overrightarrow{V}]\!] = [\![V_0]\!] \cdot \ldots \cdot [\![V_n]\!]$ for each $V_i \in \overrightarrow{V}$), composed in parallel with the translation of M, using a as the mailbox channel. We can now see that the translation preserves typeability of configurations.

Theorem 13 ($[\![-]\!]$ preserves typeability (configurations))**.**
If $\Gamma; \Delta \vdash \mathcal{C}$ *in* λ_{act}*, then* $[\![\Gamma]\!]; [\![\Delta]\!] \vdash [\![\mathcal{C}]\!]$ *in* λ_{ch}*.*

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \mathcal{C}$. Cases T-Par and T-Pid are immediate by the induction hypothesis, so we need only consider T-Actor.

Case T-Actor

Assumption:

$$\frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash M : B \quad (\Gamma, a : \mathsf{ActorRef}(A) \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, M, \overrightarrow{V} \rangle}$$

By repeated applications of Lemma 28, we have that $([\![\Gamma]\!], a : \mathsf{ChanRef}([\![A]\!]) \vdash [\![V]\!]_i : [\![A]\!])_i$ for each V_i.

By Lemma 28, we have that $[\![\Gamma]\!], a : \mathsf{ChanRef}([\![A]\!]) \vdash [\![M]\!]_a : [\![B]\!]$.

Thus we can show:

$$\frac{\dfrac{([\![\Gamma]\!], a : \mathsf{ChanRef}([\![A]\!]) \vdash [\![V_i]\!] : [\![A]\!])_i}{[\![\Gamma]\!], a : \mathsf{ChanRef}([\![A]\!]); a : [\![A]\!] \vdash a([\![\overrightarrow{V}]\!])} \quad \dfrac{[\![\Gamma]\!], a : \mathsf{ChanRef}([\![A]\!]) \vdash [\![M]\!]_a : [\![B]\!]}{[\![\Gamma]\!], a : \mathsf{ChanRef}([\![A]\!]); \cdot \vdash [\![M]\!]_a}}{[\![\Gamma]\!], a : \mathsf{ChanRef}([\![A]\!]); a : [\![A]\!] \vdash a([\![\overrightarrow{V}]\!]) \parallel [\![M]\!]_a}$$

as required.

□

We may now show that the translation is semantics-preserving, which we achieve by showing an operational correspondence. An operational correspondence states that reduction in the source language is simulated by the translation, and that the translation does not introduce any behaviour that does not correspond to reduction in the source language.

To establish the result, we begin by showing that λ_{act} term reduction can be simulated in λ_{ch}.

Lemma 29 (Simulation (λ_{act} term reduction in λ_{ch}))**.**
If $\Gamma \vdash M_1 : A$ *and* $M_1 \longrightarrow_M M_2$ *in* λ_{act}*, then given some* α*,* $[\![M_1]\!]_\alpha \longrightarrow_M^* [\![M_2]\!]_\alpha$ *in* λ_{ch}*.*

Proof. By induction on the derivation of $M_1 \longrightarrow_{\text{M}} M_2$. □

As term reduction is entirely deterministic, it follows that reduction in the images of the translation on terms is reflected in the source language.

Lemma 30 (Reflection (λ_{act} term reduction in λ_{ch}))**.** *If* $\Gamma \vdash M_1 : A$ *and* $[\![M_1]\!]_\alpha \longrightarrow_M N$*, then there exists some* M_2 *such that* $M_1 \longrightarrow_M M_2$ *and* $N \longrightarrow_M^* [\![M_2]\!]_\alpha$*.*

Proof. A direct consequence of Lemma 29 and the determinism of term reduction. □

The translations on terms and configurations are compositional, and the structure of both evaluation contexts and configuration contexts is identical in both λ_{ch} and λ_{act}. Correspondingly, it immediately follows that we can lift the translation on terms to evaluation and configuration contexts.

Lemma 31.

1. *Given an evaluation context* E*, a term* M*, and a name* α*, we have that* $[\![E[M]]\!]_\alpha = [\![E]\!][[\![M]\!]]\,\alpha$*.*
2. *Given a configuration context* $\mathcal{G}$ *and a subconfiguration* $\mathcal{C}$*, we have that* $[\![\mathcal{G}[\mathcal{C}]]\!] = [\![\mathcal{G}]\!][[\![\mathcal{C}]\!]]$*.*

As the equivalence axioms are identical in λ_{ch} and λ_{act}, it follows immediately that if two configurations are equivalent in λ_{act}, then their translations are equivalent in λ_{ch}.

Lemma 32. *If* $\Gamma;\Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, then* $[\![\mathcal{C}]\!] \equiv [\![\mathcal{D}]\!]$*.*

With these auxiliary lemmas defined, we may show a sound and complete operational correspondence for the translation from λ_{act} into λ_{ch}.

Theorem 14 (Operational Correspondence ($[\![-]\!]$))**.**

Simulation *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then* $[\![\mathcal{C}_1]\!] \Longrightarrow^* [\![\mathcal{C}_2]\!]$

Reflection *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $[\![\mathcal{C}_1]\!] \Longrightarrow \mathcal{D}$*, then there exists some* $\mathcal{C}_2$ *such that* $\mathcal{C}_1 \Longrightarrow \mathcal{C}_2$ *and* $\mathcal{D} \Longrightarrow^* [\![\mathcal{C}_2]\!]$

Proof. **Simulation**: By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$. We show the case for E-Send here; full details can be found in Appendix B.

Case E-Send

$$
\begin{array}{l}
[\![\langle a, E[\mathbf{send}\, V'\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \rangle]\!] \\
\quad \text{Def. } [\![-]\!] \\
a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{give}\, [\![V']\!]\, b]\, a) \parallel b([\![\overrightarrow{W}]\!]) \parallel ([\![M]\!]_b) \\
\quad \Longrightarrow \text{(E-Give)} \\
a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{return}\, ()]\, a) \parallel b([\![\overrightarrow{W}]\!] \cdot [\![V']\!]) \parallel ([\![M]\!]_b) \\
\quad = \\
[\![\langle a, E[\mathbf{return}\, ()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot V' \rangle]\!]
\end{array}
$$

Reflection:

Assume Δ contains entries $a_1 : A_1, \ldots, a_k : A_k$.

By Lemma 27:

$$\mathcal{C}_1 \equiv (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1} \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle) = \mathcal{C}_{\text{canon}}$$

By the definition of $[\![-]\!]$:

$$[\![C_{\text{canon}}]\!] = (\nu a_{k+1}) \cdots (\nu a_n)(a_1(\overrightarrow{V_1}) \parallel [\![M_1]\!]_{a_1} \parallel \cdots \parallel a_n(\overrightarrow{V_n}) \parallel [\![M_n]\!]_{a_n})$$

We proceed by case analysis on the structure of translated terms, inspecting the reduction rules for λ_{ch}. Without loss of generality, we assume that reduction occurs in the thread translated with respect to name a_1. We show the case for $[\![\mathbf{send}\, V\, W]\!]_{a_1}$ here; the remaining cases can be found in Appendix B.

Case $[\![\mathbf{send}\, V\, W]\!]_{a_1}$

$$[\![E[\mathbf{send}\, V\, W]]\!]_{a_1} = [\![E]\!][\mathbf{give}\, [\![V]\!]\, [\![W]\!]]\, a_1$$

The applicable reduction rule is E-Give. Thus, there exists some $\mathcal{G}$ such that:

$$\mathcal{C}_{\text{canon}} \equiv \mathcal{G}[[\![E]\!][\mathbf{give}\, [\![W]\!]\, b]\, a_1 \parallel b([\![\overrightarrow{V}]\!])]$$

We have two subcases, based on the value of b. We need not consider free names or variables without an associated buffer, as these would not reduce.

Subcase $b = a_1$

$$
\begin{array}{l}
[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_1]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) \\
\quad \longrightarrow \text{(E-Give)} \\
[\![E]\!][\mathbf{return}\, ()]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!] \cdot [\![W]\!]) \\
\quad = \\
[\![\langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W \rangle]\!]
\end{array}
$$

By the definition of $[\![-]\!]$:

$$[\![E]\!][\mathbf{give}\, W\, b]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) = [\![\langle a_1, E[\mathbf{send}\, W\, a_1], \overrightarrow{V_1} \rangle]\!]$$

In λ_{act}, we can show:

$$\begin{array}{l} \langle a_1, E[\mathbf{send}\, W\, a_1], \overrightarrow{V_1} \rangle \\ \quad \longrightarrow \text{(E-SendSelf)} \\ \langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W \rangle \end{array}$$

Thus, we can see that:

$$\begin{array}{l} [\![\mathcal{C}_{\text{canon}}]\!] \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)([\![E]\!][\mathbf{return}\, ()]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!] \cdot [\![W]\!]) \parallel \cdots \parallel [\![M_n]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!])) \\ \quad = \\ [\![(\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle)]\!] \end{array}$$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle)$$

as required.

Subcase $b = a_j$ for some $j \neq 1$

Without loss of generality, consider the case where $j = n$. By the definition of $[\![\mathcal{C}_{\text{canon}}]\!]$, we have that there must exist some $\mathcal{G}'$ such that

$$\begin{array}{l} \mathcal{G}[[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_n([\![\overrightarrow{V_n}]\!])] \\ \quad \equiv \\ \mathcal{G}'[[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_n([\![\overrightarrow{V_n}]\!]) \parallel [\![M]\!]_{a_n} \parallel a_1([\![\overrightarrow{V_1}]\!])] \\ \quad \equiv \\ \mathcal{G}'[[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!])] \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!]) \end{array}$$

By E-Give:

$$\begin{array}{l} [\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!]) \\ \quad \Longrightarrow \text{(E-Give)} \\ [\![E]\!][\mathbf{return}\, ()]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!] \cdot [\![W]\!]) \\ \quad = \\ [\![\langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \rangle \parallel \langle a_n, M_n, V_n \cdot W \rangle]\!] \end{array}$$

By the definition of $[\![-]\!]$:

$$[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_1(\overrightarrow{V_1}) \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!]) = [\![\langle a_1, E[\mathbf{send}\, W\, a_n], \overrightarrow{V_1} \rangle \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle]\!]$$

In λ_{act}, we can show:

$$\begin{array}{l} \langle a_1, E[\mathbf{send}\, W\, a_n], \overrightarrow{V_1} \rangle \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle \\ \quad \longrightarrow \text{(E-Send)} \\ \langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \rangle \parallel \langle a_n, M_n, \overrightarrow{V_n} \cdot W \rangle \end{array}$$

Thus, we can see that:

$$[\![\, \mathcal{C}_{\text{canon}} \,]\!] \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)([\![E]\!][\mathbf{return}\ ()]\, a_1 \parallel a_1([\![\, \overrightarrow{V_1} \,]\!]) \parallel \cdots \parallel [\![M]\!]_{a_n} \parallel a_n([\![\, \overrightarrow{V_n} \,]\!] \cdot [\![W]\!]))$$

$$=$$

$$[\![\, (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\ ()], \overrightarrow{V_1} \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \cdot W \rangle) \,]\!]$$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\ ()], \overrightarrow{V_1} \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \cdot W \rangle)$$

as required.

□

Remark. *The work of Gorla [81] on encodability criteria for process calculi refers to simulation as* completeness*, and reflection as* soundness*. We elect instead to adopt the terminology of Carbone et al. [30] so as to avoid confusion with soundness and completeness with respect to a denotational semantics, where the property names are seemingly flipped.*

5.2 From λ_{ch} to λ_{act}

The translation from λ_{act} into λ_{ch} emulates an actor mailbox using a channel to implement operations which normally rely on the context of the actor. Though global, the translation is straightforward due to the limited form of communication supported by mailboxes. Translating from λ_{ch} into λ_{act} is more challenging, as would be expected from Figure 4.2. Each channel in a system may have a different type; each process may have access to multiple channels; and (crucially) channels may be freely passed between processes.

5.2.1 Extensions to the core language

We require several more language constructs: sums, products, recursive functions, and iso-recursive types. Recursive functions are used to implement an event loop, and recursive types are used to maintain a term-level buffer. Products are used to record both a list of values in the buffer and a list of pending requests. Sum types allow the disambiguation of the two types of messages sent to an actor: one to queue a value (emulating **give**) and one to dequeue a value (emulating **take**). Sums are also used to encode monomorphic variant types; we write $\langle \ell_1 : A_1, \ldots, \ell_n : A_n \rangle$ for variant types and $\langle \ell_i = V \rangle$ for variant values.

Figure 5.2 shows the extensions to the core term language and their reduction rules. With products, sums, and recursive types, we can encode lists. The typing rules are shown for λ_{ch} but can be easily adapted for λ_{act}, and it is straightforward to verify that the extended languages still enjoy progress and preservation.

Additional syntax

$$\begin{array}{lrcl} \text{Types} & A,B,C & ::= & \ldots \mid A \times B \mid A + B \mid \mathsf{List}(A) \mid \mu t.A \mid t \\ \text{Values} & V,W & ::= & \ldots \mid \mathbf{rec}\, f(x).M \mid (V,W) \mid \mathbf{inl}\, V \mid \mathbf{inr}\, W \mid \mathbf{roll}\, V \\ \text{Terms} & L,M,N & ::= & \ldots \mid \mathbf{let}\,(x,y) = V\,\mathbf{in}\,M \mid \mathbf{case}\, V\,\{\mathsf{inl}\, x \mapsto M; \mathsf{inr}\, y \mapsto N\} \mid \mathbf{unroll}\, V \end{array}$$

Additional value typing rules $\boxed{\Gamma \vdash V : A}$

T-Rec
$$\frac{\Gamma, x : A, f : A \to B \vdash M : B}{\Gamma \vdash \mathbf{rec}\, f(x).M : A \to B}$$

T-Pair
$$\frac{\Gamma \vdash V : A \quad \Gamma \vdash W : B}{\Gamma \vdash (V,W) : A \times B}$$

T-Inl
$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{inl}\, V : A + B}$$

T-Inr
$$\frac{\Gamma \vdash V : B}{\Gamma \vdash \mathbf{inr}\, V : A + B}$$

T-Roll
$$\frac{\Gamma \vdash V : A\{\mu t.A/t\}}{\Gamma \vdash \mathbf{roll}\, V : \mu t.A}$$

Additional term typing rules $\boxed{\Gamma \vdash M : A}$

T-Let
$$\frac{\Gamma \vdash V : A \times B \qquad \Gamma, x : A, y : B \vdash M : C}{\Gamma \vdash \mathbf{let}\,(x,y) = V\,\mathbf{in}\,M : C}$$

T-Case
$$\frac{\Gamma \vdash V : A + B \qquad \Gamma, x : A \vdash M : C \quad \Gamma, y : B \vdash N : C}{\Gamma \vdash \mathbf{case}\, V\,\{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} : C}$$

T-Unroll
$$\frac{\Gamma \vdash V : \mu t.A}{\Gamma \vdash \mathbf{unroll}\, V : A\{\mu t.A/t\}}$$

Additional term reduction rules $\boxed{M \longrightarrow_{\mathsf{M}} M'}$

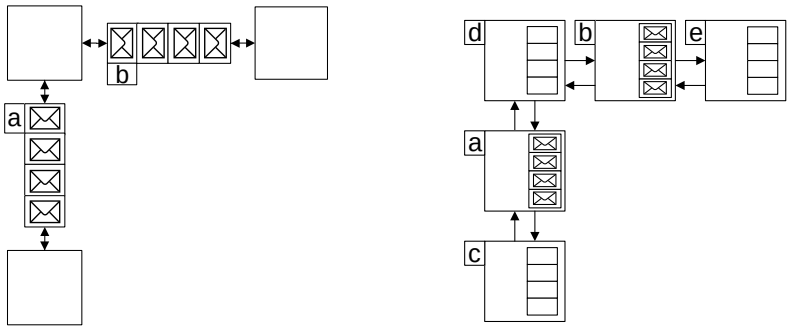
$$\begin{aligned} (\mathbf{rec}\, f(x).M)\, V &\longrightarrow_{\mathsf{M}} M\{(\mathbf{rec}\, f(x).M)/f, V/x\} \\ \mathbf{let}\,(x,y) = (V,W)\,\mathbf{in}\,M &\longrightarrow_{\mathsf{M}} M\{V/x, W/y\} \\ \mathbf{case}\,(\mathbf{inl}\, V)\,\{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} &\longrightarrow_{\mathsf{M}} M\{V/x\} \\ \mathbf{case}\,(\mathbf{inr}\, V)\,\{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} &\longrightarrow_{\mathsf{M}} N\{V/y\} \\ \mathbf{unroll}\,(\mathbf{roll}\, V) &\longrightarrow_{\mathsf{M}} \mathbf{return}\, V \end{aligned}$$

Encoding of lists

$$\mathsf{List}(A) \triangleq \mu t.\mathbf{1} + (A \times t) \qquad [\,] \triangleq \mathbf{roll}\,(\mathbf{inl}\,()) \qquad V :: W \triangleq \mathbf{roll}\,(\mathbf{inr}\,(V,W))$$

$$\mathbf{case}\, V\,\{[\,] \mapsto M; x :: y \mapsto N\} \triangleq \mathbf{let}\, z \Leftarrow \mathbf{unroll}\, V\,\mathbf{in}\,\mathbf{case}\, z\,\{\mathbf{inl}\,() \mapsto M; \mathbf{inr}\,(x,y) \mapsto N\}$$

Figure 5.2: Extensions to core languages to allow translation from λ_{ch} into λ_{act}



(a) Before Translation (b) After Translation

Figure 5.3: Translation strategy: λ_{ch} into λ_{act}

5.2.2 Translation strategy (λ_{ch} into λ_{act})

To translate typed channels into typed actors (shown in Figure 5.3), we emulate each channel using an actor process, which is crucial in retaining the mobility of channel endpoints. Channel types describe the typing of a *communication medium* between communicating processes, where processes are unaware of the identity of other communicating parties, and the types of messages that another party may receive. Unfortunately, the same does not hold for mailboxes. Consequently, we require that before translating into actors, *every channel has the same type.* Although this may seem restrictive, it is both possible and safe to transform a λ_{ch} program with multiple channel types into a λ_{ch} program with a single channel type.

As an example, suppose we have a program which contains channels carrying values of types Int, String, and $\mathsf{ChanRef}(\mathsf{String})$. It is possible to construct a recursive variant type $\mu t.\langle \ell_1 : \mathsf{Int}, \ell_2 : \mathsf{String}, \ell_3 : \mathsf{ChanRef}(t)\rangle$ which can be assigned to all channels in the system. Then, supposing we wanted to send a 5 along a channel which previously had type $\mathsf{ChanRef}(\mathsf{Int})$, we would instead send a value $\mathbf{roll}\,\langle \ell_1 = 5\rangle$ (where $\mathbf{roll}\,V$ is the introduction rule for an iso-recursive type). We call this transformation *coalescing*.

Remark. *Note that applying the translation from λ_{ch} to λ_{act} on a configuration C, followed by the translation from λ_{act} to λ_{ch}, will not result in the original program: the round-trip translation will introduce additional processes which emulate buffers. Nevertheless, by virtue of the operational correspondence results, the behaviour of $[\![(\!| C |\!)]\!]$ simulates and reflects the behaviour of C, albeit with many administrative reductions.*

5.2.3 Translation

We write λ_{ch} judgements of the form $\{B\}\ \Gamma \vdash M : A$ for a term where all channels have type B, and similarly for value and configuration typing judgements. Under such a judgement, we can write Chan instead of $\mathsf{ChanRef}(B)$.

Meta level definitions. The majority of the translation lies within the translation of **newCh**, which makes use of the meta-level definitions body and drain. The body function emulates a channel. Firstly, the actor receives a message *recvVal*, which is either of the form **inl** V to store a message V, or **inr** W to request that a value is dequeued and sent to the actor with ID W. We assume a standard implementation of list concatenation $(+\!\!+)$. If the message is **inl** V, then V is appended to the tail of the list of values stored in the channel, and the new state is passed as an argument to drain. If the message is **inr** W, then the process ID W is appended to the end of the list of processes waiting for a value. The drain function satisfies all requests that can be satisfied, returning an updated channel state. Note that drain does not need to be recursive, since one of the lists will either be empty or a singleton.

Translation on types. Figure 5.4 shows the translation from λ_{ch} into λ_{act}. The translation function on types $(\!|-|\!)$ is defined with respect to the type of all channels C and is used to annotate function arrows and to assign a parameter to ActorRef types. The (omitted) translations on sums, products, and lists are homomorphic. The translation of Chan is $\text{ActorRef}((\!|C|\!)+\text{ActorRef}((\!|C|\!)))$, meaning an actor which can receive a request to either store a value of type $(\!|C|\!)$, or to dequeue a value and send it to a process ID of type $\text{ActorRef}((\!|C|\!))$.

Translation on communication and concurrency primitives. We omit the translation on values and functional terms, which are homomorphisms. Processes in λ_{ch} are anonymous, whereas all actors in λ_{act} are addressable; to emulate **fork**, we therefore discard the reference returned by **spawn**. The translation of **give** wraps the translated value to be sent in the left injection of a sum type, and sends to the translated channel name $(\!|W|\!)$. To emulate **take**, the process ID (retrieved using **self**) is wrapped in the right injection and sent to the actor emulating the channel, and the actor waits for the response message. Finally, the translation of **newCh** spawns a new actor to execute body.

Translation on configurations. The translation function $(\!|-|\!)$ is homomorphic on parallel composition and name restriction. Unlike λ_{ch}, a term cannot exist outside of an enclosing actor context in λ_{act}. Hence, the translation of a process evaluating term M is an actor evaluating $(\!|M|\!)$ with some fresh name a and an empty mailbox, enclosed in a name restriction. A buffer is translated to an actor with an empty mailbox, evaluating body with a state containing the (term-level) list of values previously stored in the buffer.

Although the translation from λ_{ch} into λ_{act}, is much more verbose than the translation from λ_{act} to λ_{ch}, it is (once all channels have the same type) a *local transformation* [66].

Translation on types (wrt. a channel type C)

$$(\!|\mathsf{Chan}|\!) = \mathsf{ActorRef}((\!|C|\!) + \mathsf{ActorRef}((\!|C|\!))) \qquad (\!|A \to B|\!) = (\!|A|\!) \to^{(\!|C|\!)} (\!|B|\!)$$

Translation on communication and concurrency primitives

$$(\!|\mathbf{fork}\, M|\!) = \mathbf{let}\, x \Leftarrow \mathbf{spawn}\, (\!|M|\!)\, \mathbf{in}\, \mathbf{return}\, ()$$
$$(\!|\mathbf{give}\, V\, W|\!) = \mathbf{send}\, (\mathbf{inl}\, (\!|V|\!))\, (\!|W|\!)$$
$$(\!|\mathbf{newCh}|\!) = \mathbf{spawn}\, (\mathsf{body}\, ([\,], [\,]))$$

$$(\!|\mathbf{take}\, V|\!) = \mathbf{let}\, \mathit{selfPid} \Leftarrow \mathbf{self}\, \mathbf{in}\; \mathbf{send}\, (\mathbf{inr}\, \mathit{selfPid})\, (\!|V|\!);\; \mathbf{receive}$$

Translation on configurations

$$(\!|\mathcal{C} \parallel \mathcal{D}|\!) = (\!|\mathcal{C}|\!) \parallel (\!|\mathcal{D}|\!) \qquad (\!|(\nu a)\mathcal{C}|\!) = (\nu a)(\!|\mathcal{C}|\!) \qquad (\!|M|\!) = (\nu a)(\langle a, (\!|M|\!), \varepsilon\rangle)$$
a is a fresh name

$$(\!|a(\overrightarrow{V})|\!) = \langle a, \mathsf{body}\, ((\!|\overrightarrow{V}|\!), [\,]), \varepsilon\rangle \qquad \text{where } (\!|\overrightarrow{V}|\!) = (\!|V_0|\!) :: \cdots :: (\!|V_n|\!) :: [\,]$$

Meta level definitions

```
body ≜ rec g(state) .
  let recvVal ⇐ receive in
  let (vals, pids) = state in
  case recvVal {
      inl v ↦
        let vals' ⇐ vals ++ [v] in
        let state' ⇐ drain (vals', pids) in
        g (state')
      inr pid ↦
        let pids' ⇐ pids ++ [pid] in
        let state' ⇐ drain (vals, pids') in
        g (state')
  }
```

```
drain ≜ λx.
  let (vals, pids) = x in
  case vals {
    [] ↦ return (vals, pids)
    v :: vs ↦
      case pids {
        [] ↦ return (vals, pids)
        pid :: pids ↦
          send v pid;
          return (vs, pids)
      }
  }
```

Figure 5.4: Translation from λ_{ch} into λ_{act}

5.2.4 Properties of the translation

Since all channels in the source language of the translation have the same type, we can assume that each entry in the codomain of Δ is the same type B.

Definition 6 (Translation of typing environments wrt. a channel type B)**.**

1. *Given* $\Gamma = \alpha_1{:}A_1, \ldots, \alpha_n : A_n$, *define* $(\!|\Gamma|\!) = \alpha_1 : (\!|A_1|\!), \ldots, \alpha_n : (\!|A_n|\!)$.
2. *Given* $\Delta = a_1 : B, \ldots, a_n : B$, *define* $(\!|\Delta|\!) =$
   $a_1 : ((\!|B|\!) + \mathit{ActorRef}((\!|B|\!))), \ldots, a_n : ((\!|B|\!) + \mathit{ActorRef}((\!|B|\!)))$.

The translation on terms preserves typing.

Lemma 33 ($(\!|-|\!)$ preserves typing (terms and values))**.**

1. *If* $\{B\}\ \Gamma \vdash V{:}A$, *then* $(\!|\Gamma|\!) \vdash (\!|V|\!){:}(\!|A|\!)$.
2. *If* $\{B\}\ \Gamma \vdash M{:}A$, *then* $(\!|\Gamma|\!) \mid (\!|B|\!) \vdash (\!|M|\!){:}(\!|A|\!)$.

Proof. By simultaneous induction on the derivations of $\{B\}\ \Gamma \vdash V{:}A$ and $\{B\}\ \Gamma \vdash M{:}A$. We show the cases for T-Give and T-New here; the remaining cases are similar.

Case T-Give

Assumption:

$$\frac{\{A\}\ \Gamma \vdash V : A \quad \{A\}\ \Gamma \vdash W : \mathsf{Chan}}{\{A\}\ \Gamma \vdash \mathbf{give}\, V\, W : \mathbf{1}}$$

By Lemma 33, we have that $(\!|\Gamma|\!) \vdash (\!|V|\!) : (\!|A|\!)$ and $(\!|\Gamma|\!) \vdash (\!|W|\!) : \mathsf{ActorRef}((\!|A|\!) + \mathsf{ActorRef}((\!|A|\!)))$. Thus we can show

$$\frac{\dfrac{(\!|\Gamma|\!) \vdash (\!|V|\!) : (\!|A|\!)}{(\!|\Gamma|\!) \vdash \mathbf{inl}\,(\!|V|\!) : (\!|A|\!) + \mathsf{ActorRef}((\!|A|\!))} \quad (\!|\Gamma|\!) \vdash (\!|W|\!) : \mathsf{ActorRef}((\!|A|\!) + \mathsf{ActorRef}((\!|A|\!)))}{(\!|\Gamma|\!) \vdash \mathbf{send}\,(\mathbf{inl}\,(\!|V|\!))\,(\!|W|\!) : \mathbf{1}}$$

as required.

Case T-NewCh

Assumption:

$$\frac{}{\{A\}\ \Gamma \vdash \mathbf{newCh} : \mathsf{Chan}}$$

The case amounts to typechecking body $(\!|\overrightarrow{V}|\!), [\,])$. The majority of the details are routine, but it is useful to show the types of the state and the communication and concurrency constructs.

The state has type $\mathsf{List}(\!|A|\!) \times \mathsf{List}(\mathsf{ActorRef}(\!|A|\!))$. Since $(\!|\mathsf{Chan}|\!) = (\!|A|\!) + \mathsf{ActorRef}((\!|A|\!))$, the mailbox type for an actor emulating a channel is $(\!|A|\!) + \mathsf{ActorRef}((\!|A|\!))$, and received values of this type are correctly deconstructed by the **case** construct.

Given that $(\!|\Gamma|\!) \mid (\!|A|\!) + \mathsf{ActorRef}((\!|A|\!)) \vdash \mathsf{body}\,([\,],[\,]) : B$ (as there is no base case to the recursive function, the term may be assigned any type), we can show that

$$\frac{(\!|\Gamma|\!) \mid (\!|A|\!) + \mathsf{ActorRef}((\!|A|\!)) \vdash \mathsf{body}\,([\,],[\,]) : B}{(\!|\Gamma|\!) \mid (\!|A|\!) \vdash \mathbf{spawn}\,\mathsf{body}\,([\,],[\,]) : \mathsf{ActorRef}((\!|A|\!) + \mathsf{ActorRef}((\!|A|\!)))}$$

as required.

□

The translation on configurations also preserves typeability.

Theorem 15 ($(\!|-|\!)$ preserves typeability (configurations))**.**
If $\{A\}\ \Gamma;\Delta \vdash \mathcal{C}$*, then* $(\!|\Gamma|\!);(\!|\Delta|\!) \vdash (\!|\mathcal{C}|\!)$.

Proof. By induction on the derivation of $\{A\}\ \Gamma;\Delta \vdash \mathcal{C}$. Cases T-Par and T-Chan follow immediately by the induction hypothesis; we consider T-Buf and T-Term.

Case T-Buf

Assumption:

$$\frac{(\Gamma, a : \mathsf{Chan} \vdash V_i : A)_i}{\Gamma, a : \mathsf{Chan}; a : A \vdash a(\overrightarrow{V})}$$

Let $B = (\!|A|\!) + \mathsf{ActorRef}((\!|A|\!))$.

By repeated applications of Lemma 33, we have that $((\!|\Gamma|\!), a : \mathsf{ActorRef}(B) \vdash (\!|V_i|\!) : (\!|A|\!))_i$. By similar reasoning to the translation of **newCh**, we can show that

$$(\!|\Gamma|\!), a : \mathsf{ActorRef}(B) \mid B \vdash \mathsf{body}\,(\!|\overrightarrow{V}|\!), [\,]) : C$$

Thus by T-Actor, we have that

$$\frac{(\!|\Gamma|\!), a : \mathsf{ActorRef}(B) \mid B \vdash \mathsf{body}\,(\!|\overrightarrow{V}|\!), [\,]) : C}{(\!|\Gamma|\!), a : \mathsf{ActorRef}(B); B \mid B \vdash \langle a, \mathsf{body}\,(\!|\overrightarrow{V}|\!), [\,]), \varepsilon\rangle}$$

Case T-Term

Assumption:

$$\frac{\Gamma \vdash M : B}{\Gamma; \cdot \vdash M}$$

By Lemma 33, we have that $(\!|\Gamma|\!) \vdash (\!|M|\!) : (\!|B|\!)$. By weakening and by picking a fresh name b, we also have that $(\!|\Gamma|\!), b : \mathsf{ActorRef}((\!|A|\!)) \vdash (\!|M|\!) : (\!|A|\!)$.

Thus we can show

$$\frac{\dfrac{(\!|\Gamma|\!), b : (\!|A|\!) \mid (\!|A|\!) \vdash (\!|M|\!) : (\!|B|\!)}{(\!|\Gamma|\!), b : (\!|A|\!); b : (\!|A|\!) \vdash \langle b, (\!|M|\!), \varepsilon \rangle}}{(\!|\Gamma|\!); \cdot \vdash (\nu b)(\langle b, (\!|M|\!), \varepsilon \rangle)}$$

as required.

□

As before, we can lift the translations on evaluation and configuration contexts since they are identical in both languages.

Lemma 34. *1. Given an evaluation context E and term M, it is the case that $(\!|E[M]|\!) = (\!|E|\!)[(\!|M|\!)]$.*

2. Given a configuration context $\mathcal{G}$ and subconfiguration $\mathcal{C}$, it is the case that $(\!|\mathcal{G}[\mathcal{C}]|\!) = (\!|\mathcal{G}|\!)[(\!|\mathcal{C}|\!)]$.

It is clear that reduction on translated λ_{ch} terms can simulate reduction in λ_{act}. In fact, we obtain a tighter result than the translation from λ_{ch} into λ_{act} as the simulation of β-reduction for function application only takes a single step.

Lemma 35 (Simulation ($(\!|-|\!)$, terms))**.** *If $\{B\}\ \Gamma \vdash M : A$ and $M \longrightarrow_M N$, then $(\!|M|\!) \longrightarrow_M (\!|N|\!)$.*

Again, as term reduction is deterministic, reduction in the image of the translation is reflected in the source language.

Lemma 36 (Reflection ($(\!|-|\!)$, terms))**.** *If $\{B\}\ \Gamma \vdash M_1 : A$ and $(\!|M_1|\!) \longrightarrow_M N$, then there exists some M_2 such that $M_1 \longrightarrow_M M_2$ and $N \longrightarrow_M (\!|M_2|\!)$.*

Since the equivalence axioms in λ_{ch} and λ_{act} are identical, equivalence is preserved by the translation.

Lemma 37. *If $\Gamma; \Delta \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$, then $(\!|\mathcal{C}|\!) \equiv (\!|\mathcal{D}|\!)$.*

Finally, we show a sound and complete operational correspondence for the translation from λ_{ch} into λ_{act}. We must reason up to β-equality when considering reflection since the actor generated by the translation of buffers performs a β-reduction whether or not there is a value contained in its configuration-level mailbox.

We write $=_\beta$ for β-equality.

Definition 7 (β-equality). *The relation* $=_\beta$ *is defined as an equivalence relation on terms satisfying the following axioms:*

$$(\lambda x.M)\,V =_\beta M\{V/x\} \qquad (\mathbf{rec}\, f(x)\,.\,M)\,V =_\beta M\{\mathbf{rec}\, f(x)\,.\,M/f, V/x\}$$

We extend $=_\beta$ to λ_{act} configurations:

$$\frac{\mathcal{C}_1 =_\beta \mathcal{D}_1 \quad \mathcal{C}_2 =_\beta \mathcal{D}_2}{\mathcal{C}_1 \parallel \mathcal{C}_2 =_\beta \mathcal{D}_1 \parallel \mathcal{D}_2} \qquad \frac{\mathcal{C} =_\beta \mathcal{D}}{(\nu a)\mathcal{C} =_\beta (\nu a)\mathcal{D}} \qquad \frac{M =_\beta N}{\langle a, M, \overrightarrow{V}\rangle =_\beta \langle a, N, \overrightarrow{W}\rangle}$$

Recall that $\mathcal{R}^?$ refers to the reflexive closure of a relation $\mathcal{R}$. We can now state an operational correspondence result:

Theorem 16 (Operational Correspondence ($(\!|-|\!)$))**.**

Simulation *If* $\{A\}\ \Gamma;\Delta \vdash \mathcal{C}_1$, *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then* $(\!|\,\mathcal{C}_1\,|\!) \Longrightarrow^* (\!|\,\mathcal{C}_2\,|\!)$.

Reflection *If* $\{A\}\ \Gamma;\Delta \vdash \mathcal{C}_1$, *and* $(\!|\,\mathcal{C}_1\,|\!) \Longrightarrow \mathcal{D}$, *then there exist configurations* $\mathcal{C}_2$ *and* $\mathcal{E}$ *such that* $\mathcal{C}_1 \Longrightarrow^? \mathcal{C}_2$ *and* $\mathcal{D} \Longrightarrow^* \mathcal{E}$, *where* $\mathcal{E} =_\beta (\!|\,\mathcal{C}_2\,|\!)$.

Proof. **Simulation**: By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$. We show the case for E-Give here; the remaining cases can be found in Appendix B.

Case E-Give

$$\begin{aligned}
&E[\mathbf{give}\, W\, a] \parallel a(\overrightarrow{V}) \\
&\quad \text{Definition of } (\!|-|\!) \\
&(\nu b)(\langle b, (\!|E|\!)[\mathbf{send}\,(\mathbf{inl}\,(\!|W|\!))\,a], \varepsilon\rangle) \parallel \langle a, \mathsf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), \varepsilon\rangle \\
&\quad \equiv \\
&(\nu b)(\langle b, (\!|E|\!)[\mathbf{send}\,(\mathbf{inl}\,(\!|W|\!))\,a], \varepsilon\rangle \parallel \langle a, \mathsf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), \varepsilon\rangle) \\
&\quad \longrightarrow \text{(E-Send)} \\
&(\nu b)(\langle b, (\!|E|\!)[\mathbf{return}\,()], \varepsilon\rangle \parallel \langle a, \mathsf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), \mathbf{inl}\,(\!|W|\!)\rangle) \\
&\quad \equiv \\
&(\nu b)(\langle a, \mathsf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), \mathbf{inl}\,(\!|W|\!)\rangle \parallel \langle b, (\!|E|\!)[\mathbf{return}\,()], \varepsilon\rangle)
\end{aligned}$$

Now, let $\mathcal{G}[-] = (\nu b)[-] \parallel \langle b, (\!|E|\!)[\mathbf{return}\,()], \varepsilon\rangle)$.

We now have

$$\mathcal{G}[\langle a, \mathsf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), (\mathbf{inl}\,(\!|W|\!))\rangle]$$

which we can expand to

$$\begin{aligned}
&\mathcal{G}[\langle a,\ (\mathbf{rec}\ g(state)\ . \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \mathbf{inl}\ (\!| W |\!)\rangle] \\
&\qquad \mathbf{let}\ recvVal \Leftarrow \mathbf{receive}\ \mathbf{in} \\
&\qquad \mathbf{let}\ (vals, readers) = state\ \mathbf{in} \\
&\qquad\quad \mathbf{case}\ recvVal\ \{ \\
&\qquad\qquad \mathbf{inl}\ v \mapsto \mathbf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{let}\ state' \Leftarrow \mathrm{drain}\ (newVals, readers)\ \mathbf{in} \\
&\qquad\qquad\qquad g\ (state') \\
&\qquad\qquad \mathbf{inr}\ pid \mapsto \mathbf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \mathbf{in} \\
&\qquad\qquad\qquad \mathbf{let}\ state' \Leftarrow \mathrm{drain}\ (vals, newReaders)\ \mathbf{in} \\
&\qquad\qquad\qquad g\ (state')\})\ ((\!| \overrightarrow{V} |\!), [\,])
\end{aligned}$$

Applying the arguments to the recursive function g; performing the **receive**, and the **let** and case reductions, we have:

$$\begin{aligned}
&\mathcal{G}[\langle a, \mathbf{let}\ newVals \Leftarrow (\!| \overrightarrow{V} |\!) +\!\!+ [(\!| W |\!)]\ \mathbf{in}, \varepsilon\rangle] \\
&\qquad \mathbf{let}\ state' \Leftarrow \mathrm{drain}\ (newVals, [\,])\ \mathbf{in} \\
&\qquad \mathrm{body}\ (state')
\end{aligned}$$

Next, we reduce the append operation, and note that since we pass a state without pending readers into drain, that the argument is returned unchanged:

$$\mathcal{G}[\langle a, \mathbf{let}\ state' \Leftarrow \mathbf{return}\ ((\!| \overrightarrow{V} |\!) :: (\!| W |\!) :: [\,])\ \mathbf{in}\ \mathrm{body}\ state', \varepsilon\rangle]$$

Next, we apply the let-reduction and expand the evaluation context:

$$(\nu b)(\langle a, \mathrm{body}\ ((\!| \overrightarrow{V} |\!) :: (\!| W |\!) :: [\,]), \varepsilon\rangle \parallel \langle b, (\!| E |\!)[\mathbf{return}\ ()], \varepsilon\rangle)$$

which is equivalent to

$$(\nu b)(\langle b, (\!| E |\!)[\mathbf{return}\ ()], \varepsilon\rangle) \parallel \langle a, \mathrm{body}\ ((\!| \overrightarrow{V} |\!) :: (\!| W |\!) :: [\,]), \varepsilon\rangle$$

which is equal to

$$(\!| E[\mathbf{return}\ ()] \parallel a((\!| \overrightarrow{V} \cdot W |\!)) |\!)$$

as required.

Reflection:

By Lemma 19 and assuming that $\Delta = a_1 : A, \ldots, a_k : A$, we have that $\mathcal{C}$ may be written as $(\nu a_{k+1}) \cdots (\nu a_n)(M_1 \parallel \cdots \parallel M_n \parallel a_1(\overrightarrow{V_1}) \parallel \cdots \parallel a_n(\overrightarrow{V_n}))$.

By the definition of the translation, we have that $(\!| C |\!)$ may be written

$$(\!| C |\!) = (\nu a_{k+1}) \cdots (\nu a_n)((\nu b_1)(\langle b_1, (\!| M_1 |\!), \varepsilon\rangle) \parallel \cdots \parallel (\nu b_n)(\langle b_n, (\!| M_n |\!), \varepsilon\rangle) \parallel \langle a_1, \mathsf{body}\,(\!| \overrightarrow{V_1} |\!), [\,]), \varepsilon\rangle \parallel \cdots \parallel \langle a_n, \mathsf{body}\,(\!| \overrightarrow{V_n} |\!), [\,]), \varepsilon\rangle)$$

Suppose reduction happens in an actor a_i emulating a buffer $a_i(\overrightarrow{V_i})$. Here, we have that a β-reduction takes place, and reduction becomes blocked on **receive**. This is β-equivalent to the original translation, as required.

Thus, we need only consider the case where reduction occurs in an actor emulating a thread. Without loss of generality, assume reduction occurs in actor b_1. We may then proceed by case analysis on the structure of $(\!| M_1 |\!)$. We show the case for $(\!| \mathbf{give}\, V\, W |\!)$ here. The remaining cases follow the same pattern.

Case $(\!| \mathbf{give}\, V\, W |\!)$

For reduction to occur, we have that W must be some ν-bound name a_i. Let us assume that this is a_1. Thus, we have that there exists some $\mathcal{D}$ such that $\mathcal{C}_{\text{canon}} \equiv \mathcal{D}$, where

$$\mathcal{D} = \mathcal{G}[\langle b_1, (\!| E |\!)[\mathbf{send}\,(\!| V |\!)\, a_1], \varepsilon\rangle \parallel \langle a_1, \mathsf{body}\,(\!| \overrightarrow{V_1} |\!), [\,]), \varepsilon\rangle]$$

By constructing the same derivation as for the simulation case, we have that $\mathcal{D} \Longrightarrow^{+} \mathcal{D}'$, where

$$\mathcal{D}' = \mathcal{G}[\langle b_1, (\!| E |\!)[\mathbf{return}\,()], \varepsilon\rangle \parallel \langle a_1, \mathsf{body}\,(\!| \overrightarrow{V_1} \cdot V |\!), [\,]), \varepsilon\rangle]$$

In λ_{ch}, we can show

$$\begin{aligned} & E[\mathbf{give}\, V\, a_1] \parallel a_1(\overrightarrow{V_1}) \\ & \quad \longrightarrow \text{(E-Give)} \\ & E[\mathbf{return}\,()] \parallel a_1(\overrightarrow{V_1} \cdot V) \end{aligned}$$

Thus, we have that

$$\begin{aligned} (\!| \mathcal{C}_{\text{canon}} |\!) \Longrightarrow^{+} & (\nu a_{k+1}) \cdots (\nu a_n)((\nu b_1)(\langle b_1, (\!| E |\!)[\mathbf{return}\,()], \varepsilon\rangle) \parallel \cdots \parallel (\nu b_n)(\langle b_n, (\!| M_n |\!), \varepsilon\rangle) \parallel \\ & \langle a_1, \mathsf{body}\,(\!| \overrightarrow{V_1} \cdot V |\!), [\,]), \varepsilon\rangle \parallel \cdots \parallel \langle a_n, \mathsf{body}\,(\!| \overrightarrow{V_n} |\!), [\,]), \varepsilon\rangle) \\ = & (\!| (\nu a_{k+1}) \cdots (\nu a_n)(E[\mathbf{return}\,()] \parallel \cdots \parallel M_m \parallel a_1(\overrightarrow{V_1} \cdot V) \parallel \cdots \parallel a_n(\overrightarrow{V_n})) |\!) \end{aligned}$$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(E[\mathbf{return}\,()] \parallel \cdots \parallel M_m \parallel a_1(\overrightarrow{V_1} \cdot V) \parallel \cdots \parallel a_n(\overrightarrow{V_n}))$$

as required.

□

Remark. The translation from λ_{ch} into λ_{act} is more involved than the translation from λ_{act} into λ_{ch} due to the asymmetry shown in Figure 4.2. Mailbox types are less precise, generally taking the form of a large variant type.

Typical implementations of this translation use synchronisation mechanisms such as futures or shared memory (see Chapter 6); the implementation shown in the Hopac documentation uses ML references [3]. Given the ubiquity of these abstractions, we were surprised to discover that the additional expressive power of synchronisation is not *necessary*. Our original attempt at a synchronisation-free translation was type-directed. We were surprised to discover that the translation can be described so succinctly after factoring out the coalescing step, which precisely captures the type pollution problem.

Chapter 6

Extending λ_{ch} and λ_{act}

6.1 Introduction

In Chapter 4, we introduced λ_{ch} and λ_{act}, small calculi describing channel- and actor-based communication respectively. In this chapter, we discuss common extensions to channel- and actor-based languages, and discuss how they interact with the translations described in Chapter 5. In particular, we describe *synchronisation*, *selective receive*, and *input-guarded choice*.

Synchronisation Mixing the actor model with some form of synchronisation is a common pattern in real-world applications, and is ubiquitous in practical implementations of actor-inspired languages and frameworks [200]. Adding synchronisation simplifies the translation from channels to actors by relaxing the restriction that all channels must have the same type.

Selective Receive Erlang includes a mechanism for *selectively receiving* from a mailbox, allowing messages to be processed out-of-order. We formalise this construct, and show that it may be encoded in plain λ_{act}.

Input-guarded Choice Many channel-based languages allow messages to be received from a collection of channels. We show how such a construct may be added to λ_{ch}, and discuss why it is difficult to emulate using λ_{act}.

Modified types and terms

$$\begin{aligned}\text{Types} &::= \cdots \mid \mathsf{ActorRef}(A,B)\\ \text{Terms} &::= \cdots \mid \mathbf{wait}\, V\end{aligned}$$

Additional reduction rule $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$\langle a, E[\mathbf{wait}\, b], \overrightarrow{V}\rangle \parallel \langle b, \mathbf{return}\, V', \overrightarrow{W}\rangle \longrightarrow \langle a, E[\mathbf{return}\, V'], \overrightarrow{V}\rangle \parallel \langle b, \mathbf{return}\, V', \overrightarrow{W}\rangle$$

Additional equivalence axiom $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$(\nu a)(\langle a, \mathbf{return}\, V, \overrightarrow{V}\rangle) \parallel \mathcal{C} \equiv \mathcal{C}$$

Modified typing rules for terms $\boxed{\Gamma \mid A,B \vdash M : A}$

T-SyncSpawn
$$\frac{\Gamma \mid A,B \vdash M : B}{\Gamma \mid C,C' \vdash \mathbf{spawn}\, M : \mathsf{ActorRef}(A,B)}$$

T-SyncSend
$$\frac{\Gamma \vdash V : A' \quad \Gamma \vdash W : \mathsf{ActorRef}(A',B')}{\Gamma \mid A,B \vdash \mathbf{send}\, V\, W : \mathbf{1}}$$

T-SyncRecv
$$\frac{}{\Gamma \mid A,B \vdash \mathbf{receive} : A}$$

T-SyncWait
$$\frac{\Gamma \vdash V : \mathsf{ActorRef}(A,B)}{\Gamma \mid C,C' \vdash \mathbf{wait}\, V : B}$$

T-SyncSelf
$$\frac{}{\Gamma \mid A,B \vdash \mathbf{self} : \mathsf{ActorRef}(A,B)}$$

Modified typing rules for configurations $\boxed{\Gamma;\Delta \vdash \mathcal{C}}$

T-SyncActor
$$\frac{\Gamma, a{:}\mathsf{ActorRef}(A,B) \mid A,B \vdash M{:}B \qquad (\Gamma, a{:}\mathsf{ActorRef}(A,B) \vdash V_i{:}A)_i}{\Gamma, a : \mathsf{ActorRef}(A,B); a{:}(A,B) \vdash \langle a, M, \overrightarrow{V}\rangle}$$

T-SyncNu
$$\frac{\Gamma, a : \mathsf{ActorRef}(A,B); \Delta, a : (A,B) \vdash \mathcal{C}}{\Gamma;\Delta \vdash (\nu a)\mathcal{C}}$$

Figure 6.1: λ_{act} with synchronisation

6.2 Synchronisation

Although communicating with an actor via asynchronous message passing suffices for many purposes, implementing "call-response" style interactions can become cumbersome. Practical implementations such as Erlang and Akka implement some way of synchronising on a result: Erlang achieves this by generating a unique reference to send along with a request, *selectively receiving* from the mailbox to await a response tagged with the same unique reference. Another method of synchronisation embraced by the Active Object community [55, 114, 128] and Akka is to generate a *future variable* which is populated with the result of the call.

Figure 6.1 details an extension of λ_{act} with a synchronisation primitive, **wait**, which encodes a deliberately restrictive form of synchronisation capable of emulating futures. The key idea behind **wait** is it allows some actor a to block until an actor b evaluates to a value; this value is then returned directly to a, bypassing the mailbox. A variation of the **wait** primitive is implemented as part of the Links [46] concurrency model. This is but one of multiple ways of allowing synchronisation: first-class futures, shared reference cells, or selective receive can achieve a similar result. We discuss **wait** as it avoids the need for new configurations.

We replace the unary type constructor for process IDs with a binary type constructor $\text{ActorRef}(A, B)$, where A is the type of messages that the process can receive from its mailbox, and B is the type of value to which the process will eventually evaluate. We omit the modifications to the remainder of the primitives to take the additional effect type into account.

6.2.1 Correctness

The addition of synchronisation retains type preservation.

Theorem 17 (Preservation (λ_{act} with synchronisation))**.**
If $\Gamma;\Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then* $\Gamma;\Delta \vdash \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$. We consider the case for E-Wait; the remaining cases adapt straightforwardly.

Case E-Wait

$$\langle a, E[\mathbf{wait}\, b], \overrightarrow{V} \rangle \parallel \langle b, \mathbf{return}\, V', \overrightarrow{W} \rangle \longrightarrow \langle a, E[\mathbf{return}\, V'], \overrightarrow{V} \rangle \parallel \langle b, \mathbf{return}\, V', \overrightarrow{W} \rangle$$

Assumption:

$$\dfrac{\dfrac{\Gamma \mid A, B \vdash E[\mathbf{wait}\, b] : B \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : (A, B) \vdash \langle a, E[\mathbf{wait}\, b], \overrightarrow{V} \rangle} \quad \dfrac{\dfrac{\Gamma \vdash V' : B'}{\Gamma \mid A', B' \vdash \mathbf{return}\, V' : B'} \quad (\Gamma \vdash W_i : A')_i}{\Gamma; b : (A', B') \vdash \langle b, \mathbf{return}\, V', \overrightarrow{W} \rangle}}{\Gamma; a : (A, B), b : (A', B') \vdash \langle a, E[\mathbf{wait}\, b], \overrightarrow{V} \rangle \parallel \langle a, \mathbf{return}\, V', \overrightarrow{W} \rangle}$$

where $\Gamma = \Gamma', a : \mathsf{ActorRef}(A,B), b : \mathsf{ActorRef}(A',B')$ for some Γ'.
By the suitable adaptation of Lemma 20 (subterm typeability):

$$\frac{\Gamma \vdash b : \mathsf{ActorRef}(A',B')}{\Gamma \mid A,B \vdash \mathbf{wait}\, b : B'}$$

By the suitable adaptation of Lemma 21 (subterm replacement), $\Gamma \mid (A,B) \vdash E[\mathbf{return}\, V'] : B$.
Thus, recomposing:

$$\frac{\dfrac{\Gamma \mid A,B \vdash E[\mathbf{return}\, V'] : B \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : (A,B) \vdash \langle a, E[\mathbf{return}\, V'], \overrightarrow{V}\rangle} \quad \dfrac{\dfrac{\Gamma \vdash V' : B'}{\Gamma \mid A',B' \vdash \mathbf{return}\, V' : B'} \quad (\Gamma \vdash W_i : A')_i}{\Gamma; b : (A',B') \vdash \langle b, \mathbf{return}\, V', \overrightarrow{W}\rangle}}{\Gamma; a : (A,B), b : (A',B') \vdash \langle a, E[\mathbf{return}\, V'], \overrightarrow{V}\rangle \parallel \langle a, \mathbf{return}\, V', \overrightarrow{W}\rangle}$$

as required.

□

Our characterisation of progress must now take into account actors which are waiting on another process. Note that λ_{act} with synchronisation has a strictly weaker progress property than plain λ_{act}, since an actor waiting on a blocked actor, or itself, cannot reduce.

Theorem 18 (Weak Progress (λ_{act} with synchronisation))**.** *Suppose $\cdot;\cdot \vdash \mathcal{C}$, where $\mathcal{C}$ is in canonical form, and $\mathcal{C} \not\Longrightarrow$. Let $\mathcal{C} = (\nu a_1) \cdots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1}\rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$. Then, for each name a_i, either:*

1. *The actor with name a_i is of the form $\langle a_i, \mathbf{return}\, W, \overrightarrow{V_i}\rangle$; or*
2. *The actor with name a_i is of the form $\langle a_i, E[\mathbf{receive}], \varepsilon\rangle$; or*
3. *The actor with name a_i is of the form $\langle a_i, E[\mathbf{wait}\, a_j], \overrightarrow{V_i}\rangle$, where either $j = i$, or $j \neq i$ and the actor with name a_j is of the form $\langle a_j, M, \overrightarrow{V_j}\rangle$, where $M \neq \mathbf{return}\, W$ for some W.*

Proof. Similar to the proof of Theorem 12, inspecting the reduction rule for **wait**. □

6.2.2 Simplifying the translation from λ_{ch} to λ_{act}

Figure 6.2 shows how to adapt the previous translation from λ_{ch} to λ_{act}, making use of **wait** to avoid the need for the coalescing transformation. Channel references are translated into actor references which can either receive a value of type A, or the PID of a process which can receive a value of type A and will eventually evaluate to a value of type A. Note that the unbound annotation C, C' on function arrows reflects that the mailboxes can be of *any* type, since the mailboxes are unused in the actors emulating threads.

Modified translation on types

$$\begin{aligned}(\!|\,\mathsf{ChanRef}(A)\,|\!) &= \mathsf{ActorRef}((\!|A|\!) + \mathsf{ActorRef}((\!|A|\!),(\!|A|\!)),B)\\ (\!|A \to B|\!) &= (\!|A|\!) \to^{C,C'} (\!|B|\!)\end{aligned}$$

Modified translation on terms

$$\begin{aligned}(\!|\,\mathbf{take}\,V\,|\!) = \mathbf{let}\ &\mathit{requestorPid} \Leftarrow \mathbf{spawn}\ (\\ &\mathbf{let}\ \mathit{selfPid} \Leftarrow \mathbf{self}\ \mathbf{in}\\ &\mathbf{send}\ (\mathbf{inr}\ \mathit{selfPid})\ (\!|V|\!);\\ &\mathbf{receive})\ \mathbf{in}\\ &\mathbf{wait}\ \mathit{requestorPid}\end{aligned}$$

Figure 6.2: Modified translation from λ_{ch} into λ_{act} using synchronisation

The key idea behind the modified translation is to spawn a fresh actor which makes the request to the channel and blocks waiting for the response. Once the spawned actor has received the result, the result can be retrieved synchronously using **wait** *without* reading from the mailbox. The previous soundness theorems adapt to the new setting.

Lemma 38 ($(\!|-|\!)$ preserves typing (with synchronisation, terms))**.**

1. *If* $\Gamma \vdash V{:}A$, *then* $(\!|\Gamma|\!) \vdash (\!|V|\!){:}(\!|A|\!)$.
2. *If* $\Gamma \vdash M{:}A$, *then* $(\!|\Gamma|\!) \mid (\!|B|\!),(\!|C|\!) \vdash (\!|M|\!){:}(\!|A|\!)$, *for any* B,C.

Proof. By simultaneous induction on the derivations of $\Gamma \vdash V : A$ and $\Gamma \vdash M : A$. We prove the case for T-Take. The remaining cases adapt straightforwardly.

Case T-Take

Assumption:

$$\frac{\Gamma \vdash V : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathbf{take}\,V : A}$$

By the IH (premise 1), we have that $(\!|\Gamma|\!) \vdash (\!|V|\!) : \mathsf{ActorRef}((\!|A|\!) + \mathsf{ActorRef}((\!|A|\!),(\!|A|\!)),B')$. Let N be the term:

$$\begin{aligned}&\mathbf{let}\ \mathit{selfPid} \Leftarrow \mathbf{self}\ \mathbf{in}\\ &\mathbf{send}\ (\mathbf{inr}\ \mathit{selfPid})\ (\!|V|\!);\\ &\mathbf{receive}\end{aligned}$$

Let $\Gamma' = \llparenthesis \Gamma \rrparenthesis, \mathit{selfPid} : \mathsf{ActorRef}(\llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis)$. Let $\mathbf{D}'$ be the typing derivation:

$$\frac{}{\llparenthesis \Gamma \rrparenthesis \mid \llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis \vdash \mathbf{self} : \mathsf{ActorRef}(\llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis)}$$

We can construct a typing derivation $\mathbf{D}$ for N as follows:

$$\dfrac{\mathbf{D}' \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma' \vdash \mathbf{inr}\, \mathit{selfPid} : \llparenthesis A \rrparenthesis + \mathsf{ActorRef}(\llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis) \\ \Gamma' \vdash \llparenthesis V \rrparenthesis : \mathsf{ActorRef}(\llparenthesis A \rrparenthesis + \mathsf{ActorRef}(\llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis), B')\end{array}}{\Gamma' \mid \llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis \vdash \mathbf{send}\, (\mathbf{inr}\, \mathit{selfPid})\, \llparenthesis V \rrparenthesis : \mathbf{1}} \qquad \dfrac{}{\Gamma' \mid \llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis \vdash \mathbf{receive} : \llparenthesis A \rrparenthesis}}{\Gamma' \vdash \mathbf{send}\, (\mathbf{inr}\, \mathit{selfPid})\, \llparenthesis V \rrparenthesis ; \mathbf{receive} : \llparenthesis A \rrparenthesis}}{\llparenthesis \Gamma \rrparenthesis \mid \llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis \vdash \begin{array}{l}\mathbf{let}\, \mathit{selfPid} \Leftarrow \mathbf{self}\ \mathbf{in} \\ \mathbf{send}\, (\mathbf{inr}\, \mathit{selfPid})\, \llparenthesis V \rrparenthesis ; \\ \mathbf{receive}\end{array} : \llparenthesis A \rrparenthesis}$$

Let $\Gamma'' = \llparenthesis \Gamma \rrparenthesis, \mathit{requestorPid} : \mathsf{ActorRef}(\llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis)$. Finally, we can show:

$$\dfrac{\dfrac{\mathbf{D}}{\llparenthesis \Gamma \rrparenthesis \mid B, C \vdash \mathbf{spawn}\, N : \mathsf{ActorRef}(\llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis)} \qquad \dfrac{\Gamma'' \vdash \mathit{requestorPid} : \mathsf{ActorRef}(\llparenthesis A \rrparenthesis, \llparenthesis A \rrparenthesis)}{\Gamma'' \mid B, C \vdash \mathbf{wait}\, \mathit{requestorPid} : \llparenthesis A \rrparenthesis}}{\llparenthesis \Gamma \rrparenthesis \mid B, C \vdash \mathbf{let}\, \mathit{requestorPid} \Leftarrow \mathbf{spawn}\, N\ \mathbf{in}\ \mathbf{wait}\, \mathit{requestorPid} : \llparenthesis A \rrparenthesis}$$

for any arbitrary B and C as required.

□

Theorem 19 ($\llparenthesis - \rrparenthesis$ preserves typing (λ_{act} with synchronisation, configurations))**.** *If* $\Gamma; \Delta \vdash \mathcal{C}$, *then* $\llparenthesis \Gamma \rrparenthesis; \llparenthesis \Delta \rrparenthesis \vdash \llparenthesis \mathcal{C} \rrparenthesis$.

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \mathcal{C}$, using Lemma 38. The cases adapt straightforwardly. □

Theorem 20 (Operational Correspondence ($\llparenthesis - \rrparenthesis$ with synchronisation))**.**

Simulation *If* $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then* $\llparenthesis \mathcal{C}_1 \rrparenthesis \Longrightarrow^* \llparenthesis \mathcal{C}_2 \rrparenthesis$.

Reflection *If* $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\llparenthesis \mathcal{C}_1 \rrparenthesis \Longrightarrow \mathcal{D}$, *then there exists some* $\mathcal{C}_2$ *such that* $\mathcal{C}_1 \Longrightarrow^* \mathcal{C}_2$ *and* $\mathcal{D} \Longrightarrow^* \mathcal{E}$, *where* $\mathcal{E} =_{\beta} \llparenthesis \mathcal{C}_2 \rrparenthesis$.

Proof. Similar to the proof of Theorem 16; the main difference is the derivation for case E-Wait. We show this case in Appendix B. □

Preliminary work has shown that the translation from λ_{act} with **wait** into λ_{ch} requires named threads and a **join** construct in λ_{ch}. We leave the details to future work.

Additional syntax

$$\text{Receive Patterns } c ::= (\langle \ell = x \rangle \textbf{ when } M) \mapsto N$$
$$\text{Computations } M ::= \cdots \mid \textbf{receive } \{\overrightarrow{c}\}$$

Additional term typing rule $\boxed{\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash M : A}$

T-SelRecv
$$\frac{\overrightarrow{c} = \{\langle \ell_i = x_i \rangle \textbf{ when } M_i \mapsto N_i\}_i \quad i \in J \qquad \Gamma, x_i : A_i \vdash_{\mathrm{P}} M_i : \mathsf{Bool} \quad \Gamma, x_i : A_i \mid \langle \ell_j : A_j \rangle_{j \in J} \vdash N_i : C}{\Gamma \mid \langle \ell_j : A_j \rangle_{j \in J} \vdash \textbf{receive } \{\overrightarrow{c}\} : C}$$

Typing rules for purely-functional values $\boxed{\Gamma \vdash_{\mathrm{P}} V : A}$

TP-Var
$$\frac{\alpha : A \in \Gamma}{\Gamma \vdash_{\mathrm{P}} \alpha : A}$$

TP-Abs
$$\frac{\Gamma, x : A \vdash_{\mathrm{P}} M : B}{\Gamma \vdash_{\mathrm{P}} \lambda x.M : A \to B}$$

TP-Unit
$$\frac{}{\Gamma \vdash_{\mathrm{P}} () : \mathbf{1}}$$

TP-Rec
$$\frac{\Gamma, f : A \to B, x : A \vdash_{\mathrm{P}} M : B}{\Gamma \vdash_{\mathrm{P}} \textbf{rec } f(x).M : A \to B}$$

TP-Pair
$$\frac{\Gamma \vdash_{\mathrm{P}} V : A \quad \Gamma \vdash_{\mathrm{P}} W : B}{\Gamma \vdash_{\mathrm{P}} (V, W) : (A \times B)}$$

TP-Inl
$$\frac{\Gamma \vdash_{\mathrm{P}} V : A}{\Gamma \vdash_{\mathrm{P}} \textbf{inl } V : A + B}$$

TP-Inr
$$\frac{\Gamma \vdash_{\mathrm{P}} V : B}{\Gamma \vdash_{\mathrm{P}} \textbf{inr } V : A + B}$$

TP-Roll
$$\frac{\Gamma \vdash_{\mathrm{P}} V : A\{\mu t.A/t\}}{\Gamma \vdash_{\mathrm{P}} \textbf{roll } V : \mu t.A}$$

Typing rules for purely-functional computations $\boxed{\Gamma \vdash_{\mathrm{P}} M : A}$

TP-App
$$\frac{\Gamma \vdash_{\mathrm{P}} V : A \to B \qquad \Gamma \vdash_{\mathrm{P}} W : A}{\Gamma \vdash_{\mathrm{P}} V\,W : B}$$

TP-EffLet
$$\frac{\Gamma \vdash_{\mathrm{P}} M : A \qquad \Gamma, x : A \vdash_{\mathrm{P}} N : B}{\Gamma \vdash_{\mathrm{P}} \textbf{let } x \Leftarrow M \textbf{ in } N : B}$$

TP-EffReturn
$$\frac{\Gamma \vdash_{\mathrm{P}} V : A}{\Gamma \vdash_{\mathrm{P}} \textbf{return } V : A}$$

TP-Let
$$\frac{\Gamma \vdash_{\mathrm{P}} V : A \times B \qquad \Gamma, x : A, y : B \vdash_{\mathrm{P}} M : C}{\Gamma \vdash_{\mathrm{P}} \textbf{let } (x, y) = V \textbf{ in } M : C}$$

TP-Case
$$\frac{\Gamma \vdash_{\mathrm{P}} V : A + B \qquad \Gamma, x : A \vdash_{\mathrm{P}} M : C \quad \Gamma, y : B \vdash_{\mathrm{P}} N : C}{\Gamma \vdash_{\mathrm{P}} \textbf{case } V \{\textbf{inl } x \mapsto M; \textbf{inr } y \mapsto N\} : C}$$

TP-Unroll
$$\frac{\Gamma \vdash_{\mathrm{P}} V : \mu t.A}{\Gamma \vdash_{\mathrm{P}} \textbf{unroll } V : A\{\mu t.A/t\}}$$

Additional configuration reduction rule $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

E-SelRecv
$$\frac{\exists k, l. \forall i. i < k \Rightarrow \neg \mathsf{matchesAny}(\overrightarrow{c}, V_i) \wedge \mathsf{matches}(c_l, V_k) \wedge \forall j. j < l \Rightarrow \neg \mathsf{matches}(c_j, V_k)}{\langle a, E[\textbf{receive } \{\overrightarrow{c}\}], \overrightarrow{W} \cdot V_k \cdot \overrightarrow{W'} \rangle \longrightarrow \langle a, E[N_l\{V'_k/x_l\}], \overrightarrow{W} \cdot \overrightarrow{W'} \rangle}$$

where

$$\overrightarrow{c} = \{\langle \ell_i = x_i \rangle \textbf{ when } M_i \mapsto N_i\}_i \qquad \overrightarrow{W} = V_1 \cdot \ldots \cdot V_{k-1} \qquad \overrightarrow{W'} = V_{k+1} \cdot \ldots \cdot V_n \qquad V_k = \langle \ell_k = V'_k \rangle$$

$$\mathsf{matches}((\langle \ell = x \rangle \textbf{ when } M) \mapsto N, \langle \ell' = V \rangle) \triangleq (\ell = \ell') \wedge (M\{V/x\} \longrightarrow^*_{\mathrm{M}} \textbf{return } \mathsf{true})$$
$$\mathsf{matchesAny}(\overrightarrow{c}, V) \triangleq \exists c \in \overrightarrow{c}.\mathsf{matches}(c, V)$$

Figure 6.3: Additional syntax, typing rules, and reduction rules for λ_{act} with selective receive

6.3 Selective receive

The **receive** construct in λ_{act} can only read the first message in the queue, which is cumbersome as it often only makes sense for an actor to handle a subset of messages at a given time.

In practice, Erlang provides a *selective receive* construct, matching messages in the mailbox against multiple pattern clauses. Assume we have a mailbox containing values $V_1, \ldots V_n$ and evaluate **receive** $\{c_1, \ldots, c_m\}$. The construct first tries to match value V_1 against clause c_1—if it matches, then the body of c_1 is evaluated, whereas if it fails, V_1 is tested against c_2 and so on. Should V_1 not match any pattern, then the process is repeated until V_n has been tested against c_m. At this point, the process blocks until a matching message arrives.

More concretely, consider an actor with mailbox type $C = \langle \mathsf{PriorityMessage} : \mathsf{Message}, \mathsf{StandardMessage} : \mathsf{Message}, \mathsf{Timeout} : \mathbf{1} \rangle$ which can receive both high- and low-priority messages. Let *getPriority* be a function which extracts a priority from a message.
Now consider the following definitions:

$$
\begin{array}{l}
\mathsf{handlePriority} \triangleq \\
\quad \textbf{receive}\,\{ \\
\quad\quad \langle \mathsf{PriorityMessage} = msg \rangle \ \textbf{when}\ (\mathit{getPriority}\ msg) > 5 \ \mapsto\ \mathit{handleMessage}\ msg \\
\quad\quad \langle \mathsf{Timeout} = msg \rangle \qquad \textbf{when}\ \mathsf{true} \qquad\qquad\quad \mapsto\ \mathsf{handleAll} \\
\quad \} \\
\mathsf{handleAll} \triangleq \\
\quad \textbf{receive}\,\{ \\
\quad\quad \langle \mathsf{PriorityMessage} = msg \rangle \ \textbf{when}\ \mathsf{true} \ \mapsto\ \mathit{handleMessage}\ msg \\
\quad\quad \langle \mathsf{StandardMessage} = msg \rangle \ \textbf{when}\ \mathsf{true} \ \mapsto\ \mathit{handleMessage}\ msg \\
\quad\quad \langle \mathsf{Timeout} = msg \rangle \qquad \textbf{when}\ \mathsf{true} \ \mapsto\ () \\
\quad \}
\end{array}
$$

An actor evaluating handlePriority begins by handling a message only if it has a priority greater than 5. We leave the *handleMessage* function abstract, but it could, for example, log that the message has been received. After the timeout message is received, however, the actor will evaluate handleAll, which allows it to handle any message—including lower-priority messages that were received beforehand.

Figure 6.3 shows the additional syntax, typing rule, and configuration reduction rule required to encode selective receive; the type Bool and logical operators are encoded using sums in the standard way. We write $\Gamma \vdash_{\text{P}} M : A$ to mean that under context Γ, a term M which does not perform any communication or concurrency actions has type A. Intuitively, this means that no subterm of M is a communication or concurrency construct.

The **receive** $\{\overrightarrow{c}\}$ construct models an ordered sequence of receive pattern clauses c of the form $(\langle \ell = x \rangle\ \textbf{when}\ M) \mapsto N$, which can be read as "If a message with body x has label ℓ

Translation on types

$$\lfloor \mathsf{ActorRef}(\langle \ell_i : A_i \rangle_i) \rfloor = \mathsf{ActorRef}(\langle \ell_i : \lfloor A_i \rfloor \rangle_i) \qquad \lfloor A \times B \rfloor = \lfloor A \rfloor \times \lfloor B \rfloor$$

$$\lfloor A + B \rfloor = \lfloor A \rfloor + \lfloor B \rfloor \qquad \lfloor \mu t.A \rfloor = \mu t.\lfloor A \rfloor \qquad \lfloor t \rfloor = t$$

$$\lfloor A \to^{C} B \rfloor = \lfloor A \rfloor \to^{\lfloor C \rfloor} \mathsf{List}(\lfloor C \rfloor) \to^{\lfloor C \rfloor} (\lfloor B \rfloor \times \mathsf{List}(\lfloor C \rfloor))$$

where $C = \langle \ell_i : A'_i \rangle_i$, and $\lfloor C \rfloor = \langle \ell_i : \lfloor A'_i \rfloor \rangle_i$

Translation on values

$$\lfloor \lambda x.M \rfloor = \lambda x.\lambda mb.(\lfloor M \rfloor\ mb) \qquad \lfloor \mathbf{rec}\, f(x)\, .\, M \rfloor = \mathbf{rec}\, f(x)\, .\, \lambda mb.(\lfloor M \rfloor\ mb)$$

Translation on computation terms (wrt. a mailbox type $\langle \ell_i : A_i \rangle_i$)

$$\begin{aligned}
\lfloor V\, W \rfloor\ mb &= \mathbf{let}\ f \Leftarrow (\lfloor V \rfloor\ \lfloor W \rfloor)\ \mathbf{in}\ f\ mb \\
\lfloor \mathbf{return}\, V \rfloor\ mb &= \mathbf{return}\ (\lfloor V \rfloor, mb) \\
\lfloor \mathbf{let}\ x \Leftarrow M\ \mathbf{in}\ N \rfloor\ mb &= \mathbf{let}\ resPair \Leftarrow \lfloor M \rfloor\ mb\ \mathbf{in}\ \mathbf{let}\ (x, mb') = resPair\ \mathbf{in}\ \lfloor N \rfloor\ mb' \\
\lfloor \mathbf{self} \rfloor\ mb &= \mathbf{let}\ selfPid \Leftarrow \mathbf{self}\ \mathbf{in}\ \mathbf{return}\ (selfPid, mb) \\
\lfloor \mathbf{send}\, V\, W \rfloor\ mb &= \mathbf{let}\ x \Leftarrow \mathbf{send}\ (\lfloor V \rfloor)\ (\lfloor W \rfloor)\ \mathbf{in}\ \mathbf{return}\ (x, mb) \\
\lfloor \mathbf{spawn}\, M \rfloor\ mb &= \mathbf{let}\ spawnRes \Leftarrow \mathbf{spawn}(\lfloor M \rfloor[\,])\ \mathbf{in}\ \mathbf{return}\ (spawnRes, mb) \\
\lfloor \mathbf{receive}\ \{\overrightarrow{c}\} \rfloor\ mb &= \mathsf{find}(\overrightarrow{c})\ ([\,], mb)
\end{aligned}$$

Translation on configurations

$$\begin{aligned}
\lfloor (\nu a) C \rfloor &= \{(\nu a)\mathcal{D} \mid \mathcal{D} \in \lfloor C \rfloor\} \qquad \text{where} \\
\lfloor \mathcal{C}_1 \parallel \mathcal{C}_2 \rfloor &= \{\mathcal{D}_1 \parallel \mathcal{D}_2 \mid \mathcal{D}_1 \in \lfloor \mathcal{C}_1 \rfloor \wedge \mathcal{D}_2 \in \lfloor \mathcal{C}_2 \rfloor\} \qquad \overrightarrow{W_i^1} = \lfloor V_1 \rfloor :: \cdots :: \lfloor V_i \rfloor :: [\,] \\
\lfloor \langle a, M, \overrightarrow{V} \rangle \rfloor &= \{\langle a, (\lfloor M \rfloor\ \overrightarrow{W_i^1}), \overrightarrow{W_i^2} \rangle \mid i \in 0..n\} \qquad \overrightarrow{W_i^2} = \lfloor V_{i+1} \rfloor \cdot \cdots \cdot \lfloor V_n \rfloor
\end{aligned}$$

Figure 6.4: Translation from λ_{act} with selective receive into λ_{act}

and satisfies predicate M, then evaluate N". The typing rule for **receive** $\{\overrightarrow{c}\}$ ensures that for each pattern $\langle \ell_i = x_i \rangle$ **when** $M_i \mapsto N_i$ in $\overrightarrow{c}$, we have that there exists some $\ell_i : A_i$ contained in the mailbox variant type; and when Γ is extended with $x_i : A_i$, that the guard M_i has type Bool and the body N_i has the same type C for each branch. Note that pattern matching need not be exhaustive, and that the construct supports multiple patterns for each variant label.

The reduction rule for selective receive is inspired by that of Fredlund [75]. Assume that the mailbox is of the form $V_1 \cdot \cdots \cdot V_k \cdot \ldots V_n$, with $\overrightarrow{W} = V_1 \cdot \cdots \cdot V_{k-1}$ and $\overrightarrow{W'} = V_{k+1} \cdot \cdots \cdot V_n$. The $\mathsf{matches}(c, V)$ predicate holds if the label matches, and the branch guard evaluates to true. The $\mathsf{matchesAny}(\overrightarrow{c}, V)$ predicate holds if V matches any pattern in $\overrightarrow{c}$. The key idea is that V_k is the first value to satisfy a pattern. The construct evaluates to the body of the matched pattern, with the message payload V'_k substituted for the pattern variable x_k; the final mailbox is $\overrightarrow{W} \cdot \overrightarrow{W'}$ (that is, the original mailbox without V_k).

Reduction in the presence of selective receive preserves typing.

Theorem 21 (Preservation (λ_{act} configurations with selective receive)). *If* $\Gamma;\Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then* $\Gamma;\Delta \mid \langle \ell_i : A_i \rangle_i \vdash \mathcal{C}_2$.

Proof. By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$. □

Translation to λ_{act}. Given the additional constructs (i.e, sums, products, and recursive types) used to translate λ_{ch} into λ_{act}, it is possible to translate λ_{act} with selective receive into plain λ_{act}. Key to the translation is reasoning about values in the mailbox at the term level; we maintain a term-level 'save queue' of values that have been received but not yet matched, and can loop through the list to find the first matching value. Our translation is similar in spirit to the "stashing" mechanism described by Haller [83] to emulate selective receive in Akka, where messages can be moved to an auxiliary queue for processing at a later time.

Figure 6.4 shows the translation formally. Except for function types, the translation on types is homomorphic. Similar to the translation from λ_{act} into λ_{ch}, we add an additional parameter for the save queue.

The translation on terms $\lfloor M \rfloor\ mb$ takes a variable mb representing the save queue as its parameter, returning a pair of the resulting term and the updated save queue. The majority of cases follow a standard state-passing transformation; we omit the translations of constructs of the extended term language. Most important is the translation of **receive** $\{\overrightarrow{c}\}$, which relies on the meta-level definition find$\overrightarrow{c}$ (Figure 6.5), where $\overrightarrow{c}$ is a sequence of clauses. The constituent *findLoop* function takes a pair of lists (mb_1, mb_2), where mb_1 is the list of processed values found not to match, and mb_2 is the list of values still to be processed. The loop inspects the list until one either matches, or the end of the list is reached. Should no values in the term-level representation of the mailbox match, then the loop function repeatedly receives from the mailbox, testing each new message against the patterns.

Note that the **case** construct in the core λ_{act} calculus is more restrictive than selective receive: given a variant $\langle \ell_i : A_i \rangle_i$, **case** requires a single branch for each label. Selective receive allows multiple branches for each label, each containing a possibly-different predicate, and does not require pattern matching to be exhaustive.

We therefore need to perform pattern matching elaboration; this is achieved by the branches meta level definition. We make use of list comprehension notation: for example, $[c \mid (c \leftarrow \overrightarrow{c}) \wedge \mathsf{label}(c) = \ell]$ returns the (ordered) list of clauses in a sequence $\overrightarrow{c}$ such that the label of the receive clause matches a label ℓ. We assume a meta level function noDups which removes duplicates from a list. Case branches are computed using the branches meta level definition: patBranches creates a branch for each label present in the selective receive, creating (via ifPats) a sequence of if-then-else statements to check each predicate in turn; defaultBranches creates a branch for each label that is present in the mailbox type but not in any selective receive clauses.

$$\mathsf{find}(\overrightarrow{c}) \triangleq$$
$$(\textbf{rec}\ \mathit{findLoop}(\mathit{ms})\,.$$
$$\quad \textbf{let}\ (\mathit{mb}_1, \mathit{mb}_2) = \mathit{ms}\ \textbf{in}$$
$$\quad \textbf{case}\ \mathit{mb}_2\ \{$$
$$\quad\quad [\,] \mapsto \mathsf{loop}(\overrightarrow{c})\ \mathit{mb}_1$$
$$\quad\quad x :: \mathit{mb}_2' \mapsto$$
$$\quad\quad\quad \textbf{let}\ \mathit{mb}' \Leftarrow \mathit{mb}_1 +\!\!+\ \mathit{mb}_2'\ \textbf{in}$$
$$\quad\quad\quad \textbf{case}\ x\ \{\mathsf{branches}(\overrightarrow{c}, \mathit{mb}',$$
$$\quad\quad\quad\quad \lambda y.\textbf{let}\ \mathit{mb}_1' \Leftarrow \mathit{mb}_1 +\!\!+\ [y]\ \textbf{in}$$
$$\quad\quad\quad\quad\quad \mathit{findLoop}\,(\mathit{mb}_1', \mathit{mb}_2'))\}$$

$$\mathsf{label}(\langle \ell = x\rangle\ \textbf{when}\ M \mapsto N) = \ell$$
$$\mathsf{labels}(\overrightarrow{c}) = \mathsf{noDups}([\mathsf{label}(c) \mid c \leftarrow \overrightarrow{c}])$$
$$\mathsf{matching}(\ell, \overrightarrow{c}) = [c \mid (c \leftarrow \overrightarrow{c}) \wedge \mathsf{label}(c) = \ell]$$
$$\mathsf{unhandled}(\overrightarrow{c}) = [\ell \mid (\langle \ell : A\rangle \leftarrow \langle \ell_i : A_i\rangle_i) \wedge \ell \notin \mathsf{labels}(\overrightarrow{c})]$$

$$\mathsf{ifPats}(\mathit{mb}, \ell, y, \varepsilon, \mathit{default}) = \mathit{default}\,\langle \ell = y\rangle$$
$$\mathsf{ifPats}(\mathit{mb}, \ell, y,$$
$$\quad (\langle \ell = x\rangle\ \textbf{when}\ M \mapsto N) \cdot \mathit{pats}, \mathit{default}) =$$
$$\quad\quad \textbf{let}\ \mathit{resPair} \Leftarrow (\lfloor M \rfloor\ \mathit{mb})\{y/x\}\ \textbf{in}$$
$$\quad\quad \textbf{let}\ (\mathit{res}, \mathit{mb}') = \mathit{resPair}\ \textbf{in}$$
$$\quad\quad \textbf{if}\ \mathit{res}\ \textbf{then}\ (\lfloor N \rfloor\, \mathit{mb})\{y/x\}$$
$$\quad\quad \textbf{else}\ \mathsf{ifPats}(\mathit{mb}, \ell, y, \mathit{pats}, \mathit{default})$$

$$\quad \mathsf{loop}(\overrightarrow{c}) \triangleq$$
$$\quad\quad (\textbf{rec}\ \mathit{recvLoop}(\mathit{mb})\,.$$
$$\quad\quad\quad \textbf{let}\ x \Leftarrow \textbf{receive}\ \textbf{in}$$
$$\quad\quad\quad \textbf{case}\ x\ \{\mathsf{branches}(\overrightarrow{c}, \mathit{mb},$$
$$\quad\quad\quad\quad \lambda y.\textbf{let}\ \mathit{mb}' \Leftarrow \mathit{mb} +\!\!+\ [y]\ \textbf{in}$$
$$\quad\quad\quad\quad\quad \mathit{recvLoop}\ \mathit{mb}'\})$$

$$\mathsf{branches}(\overrightarrow{c}, \mathit{mb}, \mathit{default}) = \mathsf{patBranches}(\overrightarrow{c}, \mathit{mb}, \mathit{default}) \cdot \mathsf{defaultBranches}(\overrightarrow{c}, \mathit{mb}, \mathit{default})$$
$$\mathsf{patBranches}(\overrightarrow{c}, \mathit{mb}, \mathit{default}) =$$
$$\quad [\langle \ell = x\rangle \mapsto \mathsf{ifPats}(\mathit{mb}, \ell, x, \overrightarrow{c_\ell}, \mathit{default}) \mid (\ell \leftarrow \mathsf{labels}(\overrightarrow{c})) \wedge \overrightarrow{c_\ell} = \mathsf{matching}(\ell, \overrightarrow{c}) \wedge x\ \mathsf{fresh}]$$
$$\mathsf{defaultBranches}(\overrightarrow{c}, \mathit{mb}, \mathit{default}) = [\langle \ell = x\rangle \mapsto \mathit{default}\,\langle \ell = x\rangle \mid (\ell \leftarrow \mathsf{unhandled}(\overrightarrow{c})) \wedge x\ \mathsf{fresh}]$$

Figure 6.5: Meta level definitions for translation from λ_{act} with selective receive to λ_{act} (wrt. a mailbox type $\langle \ell_i : A_i\rangle_i$)

Properties of the translation. The translation preserves typing of terms and values.

Lemma 39 (Translation preserves typing (λ_{act} with selective receive into λ_{act}, values and terms))**.**

1. *If* $\Gamma \vdash V : A$*, then* $\lfloor \Gamma \rfloor \vdash \lfloor V \rfloor : \lfloor A \rfloor$.
2. *If* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash M : B$*, then*
   $\lfloor \Gamma \rfloor, mb : \mathit{List}(\langle \ell_i : \lfloor A_i \rfloor \rangle_i) \mid \langle \ell_i : \lfloor A_i \rfloor \rangle_i \vdash \lfloor M \rfloor\, mb : (\lfloor B \rfloor \times \mathit{List}(\langle \ell_i : \lfloor A_i \rfloor \rangle_i))$.

Proof. By simultaneous induction on the derivations of $\Gamma \vdash V : A$ and $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash M : B$. □

Alas, a direct one-to-one translation on configurations is not possible, since a message in a mailbox in the source language could be either in the mailbox or the save queue in the target language. Consequently, we translate a configuration into a set of possible configurations, depending on how many messages have been processed. We can show that all configurations in the resulting set are type-correct, and can simulate the original reduction. A full formal treatment of reflection is left as future work.

Theorem 22 (Translation preserves typing (λ_{act} with selective receive into λ_{act}, configurations))**.** *If* $\Gamma; \Delta \vdash \mathcal{C}$*, then* $\forall \mathcal{D} \in \lfloor \mathcal{C} \rfloor$*, it is the case that* $\lfloor \Gamma \rfloor; \lfloor \Delta \rfloor \vdash \mathcal{D}$.

Proof. By induction on the derivation of $\Gamma; \Delta \vdash \mathcal{C}$. The proof primarily relies on Lemma 39, and an induction on the length of the save queue. □

To show a simulation result, it is helpful to define some auxiliary lemmas. The first states that λ_{act} simulates term reduction in λ_{act} with selective receive, and does not modify the save queue.

Lemma 40 (Simulation (λ_{act} with selective receive in λ_{act}—terms))**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M M'$*, then given some mb, it is the case that* $\lfloor M \rfloor mb \longrightarrow_M^+ \lfloor M' \rfloor mb$.

Proof. By induction on the derivation of $M \longrightarrow_{\mathsf{M}} M'$. □

Next, we show that given a well-typed term that reduces to a value using only term reductions, its translation will reduce to a value without modifying the save queue. This lemma is useful when reasoning about the matches predicate in the translation of selective receive.

Lemma 41. *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M^* \mathbf{return}\, V$*, then* $\lfloor M \rfloor\ mb \longrightarrow_M^* \mathbf{return}\, (\lfloor V \rfloor, mb)$ *for any mb.*

Proof. By induction on the length of the reduction sequence, with appeal to Lemma 40. □

The next lemma states that a value which does not match any patterns in λ_{act} with selective receive will mean that the *default* branch is evaluated in the translation.

Lemma 42. *Suppose* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash \textbf{receive}\ \{\overrightarrow{c}\}$ *and* $\Gamma \vdash V : \langle \ell_i : A_i \rangle_i$*, where* $V = \langle \ell = V' \rangle$*. If* $\neg(\mathit{matchesAny}(\overrightarrow{c}, V))$*, then*

$$\textbf{case}\ \lfloor V \rfloor\ \{\mathit{branches}(\overrightarrow{c}, mb, \mathit{default})\} \longrightarrow_M^+ \mathit{default}\ \lfloor V \rfloor$$

Proof. See Appendix B. □

Using Lemma 42, we can show that if a value does not match, then when evaluating $\mathsf{find}(\overrightarrow{c})\ (mb_1, \lfloor V \rfloor :: mb_2)$, we have that $\lfloor V \rfloor$ is dequeued from mb_2 and appended to the end of mb_1.

Lemma 43. *Suppose* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash \textbf{receive}\ \{\overrightarrow{c}\}$ *and* $\Gamma \vdash V : \langle \ell_i : A_i \rangle_i$*, where* $V = \langle \ell = V' \rangle$*. If* $\neg(\mathit{matchesAny}(\overrightarrow{c}, V))$*, then*

$$\mathit{find}(\overrightarrow{c})\ (mb_1, \lfloor V \rfloor :: \lfloor \overrightarrow{W} \rfloor) \longrightarrow_M^+ \mathit{find}(\overrightarrow{c})\ (mb_1 +\!\!+ [\lfloor V \rfloor], \lfloor \overrightarrow{W} \rfloor)$$

Proof. See Appendix B. □

The next lemma states that if a value matches a clause in λ_{act} with selective receive, then the body of the clause will be evaluated in the translation.

Lemma 44. *Suppose* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash \textbf{receive}\ \{\overrightarrow{c}\}$.

Suppose:

- $\exists k, l. \forall i. i < k. \neg(\mathit{matchesAny}(\overrightarrow{c}, V_i)) \wedge \mathit{matches}(c_l, V_k) \wedge \forall j. j < l \Rightarrow \neg(\mathit{matches}(c_j, V_k))$.
- $\Gamma \vdash V_k : \langle \ell_i : A_i \rangle_i$.
- $V_k = \langle \ell_k = V_k' \rangle$

Then:

$$\textbf{case}\ \lfloor V_k \rfloor\ \{\mathit{branches}(\overrightarrow{c}, mb, \mathit{default}\}) \longrightarrow_M^+ (\lfloor N_l \rfloor\ mb)\{\lfloor V_k' \rfloor / x_l\}$$

Proof. See Appendix B. □

Our final auxiliary lemma considers the case of evaluating $\mathsf{loop}(\overrightarrow{c})\ mb$, where the head of the actor mailbox is some value U such that $\neg\mathsf{matchesAny}(\overrightarrow{c}, U)$. In this case, the value will be retrieved from the mailbox and added to the save queue.

Lemma 45. *Suppose* $\Gamma; \Delta \vdash \langle a, \textbf{receive}\ \{\overrightarrow{c}\}, \overrightarrow{W} \cdot U \cdot \overrightarrow{W'} \rangle$ *and* $\neg \mathit{matchesAny}(\overrightarrow{c}, U)$*. Then:*

$$\langle a, E[\mathit{loop}(\overrightarrow{c})\ \lfloor \overrightarrow{W} \rfloor], \lfloor U \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle \longrightarrow^+ \langle a, E[\mathit{loop}(\overrightarrow{c})\ (\lfloor \overrightarrow{W} \rfloor +\!\!+ [\lfloor U \rfloor])], \lfloor \overrightarrow{W'} \rfloor \rangle$$

Proof. See Appendix B. □

With all of these lemmas in place, we can state a simulation result.

Theorem 23 (Simulation (λ_{act} with selective receive in λ_{act}))**.** *If* $\Gamma; \Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$*, then* $\forall \mathcal{D} \in \lfloor \mathcal{C} \rfloor$*, there exists a* $\mathcal{D}'$ *such that* $\mathcal{D} \Longrightarrow^+ \mathcal{D}'$ *and* $\mathcal{D}' \in \lfloor \mathcal{C}' \rfloor$.

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{C}'$. The only interesting case is E-SelRecv.

Case E-SelRecv

$$\text{E-SelRecv}\quad \frac{\exists k,l.\forall i.i<k \Rightarrow \neg\mathsf{matchesAny}(\overrightarrow{c},V_i) \wedge \mathsf{matches}(c_l,V_k) \wedge \forall j.j<l \Rightarrow \neg\mathsf{matches}(c_j,V_k)}{\langle a, E[\mathbf{receive}\ \{\overrightarrow{c}\}], \overrightarrow{W}\cdot V_k\cdot\overrightarrow{W'}\rangle \longrightarrow \langle a, E[N_l\{V'_k/x_l\}], \overrightarrow{W}\cdot\overrightarrow{W'}\rangle}$$

Assumption: $\Gamma;\Delta \vdash \langle a, E[\mathbf{receive}\ \{\overrightarrow{c}\}], \overrightarrow{W}\cdot V_k\cdot\overrightarrow{W'}\rangle$. Let $\overrightarrow{W} = \lfloor V_1\rfloor\cdot\ldots\cdot\lfloor V_{k-1}\rfloor$ and let $\overrightarrow{W'} = \lfloor V_{k+1}\rfloor\cdot\ldots\cdot\lfloor V_n\rfloor$.

By the definition of $\lfloor -\rfloor$:

$$\lfloor\langle a, E[\mathbf{receive}\ \{\overrightarrow{c}\}], \overrightarrow{V}\rangle\rfloor$$
$$\{\langle a, \mathsf{find}(\overrightarrow{c})\,([\,],[\,]), \lfloor\overrightarrow{W}\rfloor\cdot\lfloor V_k\rfloor\cdot\lfloor\overrightarrow{W'}\rfloor\rangle\} \cup \{\langle a, \mathsf{find}(\overrightarrow{c})\,([\,],\overrightarrow{W_i^1}), \overrightarrow{W_i^2}\rangle \mid i\in 1..n\}$$

where
$$\overrightarrow{W_i^1} = \lfloor V_1\rfloor :: \ldots :: \lfloor V_i\rfloor :: [\,]$$
$$\overrightarrow{W_i^2} = \lfloor V_{i+1}\rfloor\cdot\ldots\cdot\lfloor V_n\rfloor$$

Without loss of generality, it suffices to consider only the cases where either the save queue or mailbox are empty, namely:

1. $\langle a, \lfloor E\rfloor[\mathsf{find}(\overrightarrow{c})\,([\,], \lfloor\overrightarrow{W}\rfloor +\!\!+ [V_k] :: \lfloor\overrightarrow{W'}\rfloor)], \varepsilon\rangle$
2. $\langle a, \lfloor E\rfloor[\mathsf{find}(\overrightarrow{c})\,([\,],[\,])], \lfloor\overrightarrow{W}\rfloor\cdot\lfloor V_k\rfloor\cdot\lfloor\overrightarrow{W'}\rfloor\rangle$

Subcase $\langle a, \lfloor E\rfloor[\mathsf{find}(\overrightarrow{c})\,([\,], \lfloor\overrightarrow{W}\rfloor +\!\!+ [V_k] :: \lfloor\overrightarrow{W'}\rfloor)], \varepsilon\rangle$

By repeated application of Lemma 43, we have that

$$\langle a, \lfloor E\rfloor[\mathsf{find}(\overrightarrow{c})\,(\lfloor\overrightarrow{W}\rfloor +\!\!+ \lfloor V_k\rfloor :: \lfloor\overrightarrow{W'}\rfloor, [\,])], \varepsilon\rangle \longrightarrow^*_{\mathsf{M}} \langle a, \lfloor E\rfloor[\mathsf{find}(\overrightarrow{c})\,(\lfloor\overrightarrow{W}\rfloor, \lfloor V_k\rfloor :: \lfloor\overrightarrow{W'}\rfloor)], \varepsilon\rangle$$

By β-reducing the pair destructor, case split, and append operations, we have:

$$\langle a, \lfloor E\rfloor[\mathsf{find}(\overrightarrow{c})\,(\lfloor\overrightarrow{W}\rfloor, \lfloor V_k\rfloor :: \lfloor\overrightarrow{W'}\rfloor)], \varepsilon\rangle \longrightarrow^+_{\mathsf{M}}$$
$$\langle a, \lfloor E\rfloor[\mathbf{case}\ \lfloor V_k\rfloor\ \mathbf{of}\ \{\mathsf{branches}(\overrightarrow{c}, \lfloor\overrightarrow{W}\rfloor +\!\!+ \lfloor\overrightarrow{W'}\rfloor, \mathit{default})\}], \varepsilon\rangle$$

where

$$\mathit{default} = \lambda y.\mathbf{let}\ mb'_1 \Leftarrow \lfloor\overrightarrow{W}\rfloor +\!\!+ [y]\ \mathbf{in}$$
$$\mathsf{find}(\overrightarrow{c})\,(mb'_1, \lfloor\overrightarrow{W'}\rfloor))\})$$

By Lemma 44, we have that

$$\langle a, \lfloor E\rfloor[\mathbf{case}\ \lfloor V_k\rfloor\ \mathbf{of}\ \{\mathsf{branches}(\overrightarrow{c}, \lfloor\overrightarrow{W}\rfloor +\!\!+ \lfloor\overrightarrow{W'}\rfloor, \mathit{default})\}], \varepsilon\rangle \longrightarrow^+_{\mathsf{M}}$$
$$\langle a, \lfloor E\rfloor[(\lfloor N_l\rfloor\,(\lfloor\overrightarrow{W}\rfloor +\!\!+ \lfloor\overrightarrow{W'}\rfloor))\{\lfloor V'_k\rfloor/x_l\}], \varepsilon\rangle$$

which is contained in the set

$$\lfloor \langle a, E[N_l\{V'_k/x_l\}], \overrightarrow{W} \cdot \overrightarrow{W'} \rangle \rfloor$$

as required.

Subcase $\langle a, \lfloor E \rfloor[\mathsf{find}(\overrightarrow{c})\,([\,],[\,])], \lfloor \overrightarrow{W} \rfloor \cdot \lfloor V_k \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle$

By β-reducing the recursive function application, pair deconstruction, and case expression, we have

$$\langle a, \lfloor E \rfloor[\mathsf{find}(\overrightarrow{c})\,([\,],[\,])], \lfloor \overrightarrow{W} \rfloor \cdot \lfloor V_k \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle \longrightarrow\longrightarrow\longrightarrow \langle a, \lfloor E \rfloor[\mathsf{loop}(\overrightarrow{c})\,[\,]], \lfloor \overrightarrow{W} \rfloor \cdot \lfloor V_k \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle$$

By repeated applications of Lemma 45:

$$\langle a, \lfloor E \rfloor[\mathsf{loop}(\overrightarrow{c})\,[\,]], \lfloor \overrightarrow{W} \rfloor \cdot \lfloor V_k \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle \longrightarrow^{+} \langle a, \lfloor E \rfloor[\mathsf{loop}(\overrightarrow{c})\,\lfloor \overrightarrow{W} \rfloor], \lfloor V_k \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle$$

By β-reducing the function application and E-Recv:

$$\begin{array}{l}\langle a, \lfloor E \rfloor[\mathsf{loop}(\overrightarrow{c})\,\lfloor \overrightarrow{W} \rfloor], \lfloor V_k \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle \longrightarrow\longrightarrow \\ \langle a, \lfloor E \rfloor[\mathbf{case}\ \lfloor V_k \rfloor\ \mathbf{of}\ \{\mathsf{branches}(\overrightarrow{c}, mb, \mathit{default})\}], \lfloor \overrightarrow{W'} \rfloor \rangle\end{array}$$

where

$$\begin{array}{c}\mathit{default} = \lambda y.\mathbf{let}\ mb' \Leftarrow mb +\!\!+\ [y]\ \mathbf{in} \\ \mathit{recvLoop}\ mb'\end{array}$$

By Lemma 44:

$$\begin{array}{l}\langle a, \lfloor E \rfloor[\mathbf{case}\ \lfloor V_k \rfloor\ \mathbf{of}\ \{\mathsf{branches}(\overrightarrow{c}, \lfloor \overrightarrow{W} \rfloor, \mathit{default})\}], \lfloor V_k \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle \longrightarrow^{+} \\ \langle a, \lfloor E \rfloor[(\lfloor N_l \rfloor\ \lfloor \overrightarrow{W} \rfloor)\{\lfloor V'_k \rfloor/x_l\}], \lfloor \overrightarrow{W'} \rfloor \rangle\end{array}$$

which is contained in the set

$$\lfloor \langle a, N_l\{V'_k/x_l\}, \overrightarrow{W} \cdot \overrightarrow{W'} \rangle \rfloor$$

as required.

□

Remark. Originally we expected to need to add an analogous selective receive construct to λ_{ch} in order to be able to translate λ_{act} with selective receive into λ_{ch}. We were surprised (in part due to the complex reduction rule and the native runtime support in Erlang) when we discovered that selective receive can be emulated in plain λ_{act}. Moreover, we were pleasantly surprised that types pose no difficulties in the translation. This translation gives a formal basis to the *stashing* translation introduced by Haller [83]; our formal presentation is novel.

It is worth emphasising, however, that the translation is *global* since each function must be modified to take the save queue as a parameter.

$$\frac{\Gamma \vdash V : \mathsf{ChanRef}(A) \quad \Gamma \vdash W : \mathsf{ChanRef}(B)}{\Gamma \vdash \mathbf{choose}\, V\, W : A + B}$$

$$\text{E-Choose1}\; E[\mathbf{choose}\, a\, b] \parallel a(U \cdot \overrightarrow{V}) \parallel b(\overrightarrow{W}) \longrightarrow E[\mathbf{return}\, (\mathbf{inl}\, U)] \parallel a(\overrightarrow{V}) \parallel b(\overrightarrow{W})$$
$$\text{E-Choose2}\; E[\mathbf{choose}\, a\, b] \parallel a(\overrightarrow{V}) \parallel b(U \cdot \overrightarrow{W}) \longrightarrow E[\mathbf{return}\, (\mathbf{inr}\, U)] \parallel a(\overrightarrow{V}) \parallel b(\overrightarrow{W})$$

Figure 6.6: Additional typing and evaluation rules for λ_{ch} with choice

6.4 Choice

In λ_{ch}, processes can only block receiving on a *single* channel. A more powerful mechanism is *selective communication*, where a value is taken nondeterministically from *two* channels. An important use case is receiving a value when either channel could be empty.

We consider only the most basic form of selective choice over two channels of different types. More generally, it may be extended to arbitrary regular data types [169]; an alternative design choice (as taken by Concurrent ML) is to perform choice over homogeneously-typed lists. As Concurrent ML [186] embraces rendezvous-based synchronous communication, it provides *generalised selective communication* where a process can synchronise on a mixture of input or output communication events. Similarly, the join patterns of the join calculus [69] provide a general abstraction for selective communication over multiple channels.

As we are working in the asynchronous setting where a **give** operation can reduce immediately, we consider only input-guarded choice. Input-guarded choice can be added straightforwardly to λ_{ch}, as shown in Figure 6.6.

The extension of λ_{ch} with input-guarded choice retains preservation.

Theorem 24 (λ_{ch} with choice)**.** *If* $\Gamma;\Delta \vdash \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$*, then* $\Gamma;\Delta \vdash \mathcal{C}'$*.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{C}'$. We show the case for E-Choose1; the case for E-Choose2 is similar.

Case E-Choose1

$$E[\mathbf{choose}\, a\, b] \parallel a(U \cdot \overrightarrow{V}) \parallel b(\overrightarrow{W}) \longrightarrow E[\mathbf{return}\, (\mathbf{inl}\, U)] \parallel a(\overrightarrow{V}) \parallel b(\overrightarrow{W})$$

Assumption:

$$\frac{\dfrac{\Gamma \vdash E[\mathbf{choose}\, a\, b] : C}{\Gamma;\cdot \vdash E[\mathbf{choose}\, a\, b]} \qquad \dfrac{\dfrac{\Gamma \vdash U : A \quad (\Gamma \vdash V_i : A)_i}{\Gamma;a : A \vdash a(U \cdot \overrightarrow{V})} \qquad \dfrac{(\Gamma \vdash W_i : B)_i}{\Gamma;b : B \vdash b(\overrightarrow{W})}}{\Gamma;a : A, b : B \vdash a(U \cdot \overrightarrow{V}) \parallel b(\overrightarrow{W})}}{\Gamma;a : A, b : B \vdash E[\mathbf{choose}\, a\, b] \parallel a(U \cdot \overrightarrow{V}) \parallel b(\overrightarrow{W})}$$

where $\Gamma = \Gamma', a : \mathsf{ChanRef}(A), b : \mathsf{ChanRef}(B)$ for some Γ'.

By (the suitable adaptation of) Lemma 12 (subterm typeability), we have:

$$\frac{\Gamma \vdash a : \mathsf{ChanRef}(A) \quad \Gamma \vdash b : \mathsf{ChanRef}(B)}{\Gamma \vdash \mathbf{choose}\, a\, b : A + B}$$

We can show that $\Gamma \vdash \mathbf{return}\,(\mathbf{inl}\, U) : A + B$. Thus by (the suitable adaptation of) Lemma 13 (subterm replacement), we have that $\Gamma \vdash E[\mathbf{return}\,(\mathbf{inl}\, U)] : C$.

Recomposing:

$$\dfrac{\dfrac{\Gamma \vdash E[\mathbf{return}\,(\mathbf{inl}\, U)] : C}{\Gamma; \cdot \vdash E[\mathbf{return}\,(\mathbf{inl}\, U)]} \quad \dfrac{\dfrac{(\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash a(\overrightarrow{V})} \quad \dfrac{(\Gamma \vdash W_i : B)_i}{\Gamma; b : B \vdash b(\overrightarrow{W})}}{\Gamma; a : A, b : B \vdash a(\overrightarrow{V}) \parallel b(\overrightarrow{W})}}{\Gamma; a : A, b : B \vdash E[\mathbf{return}\,(\mathbf{inl}\, U)] \parallel a(\overrightarrow{V}) \parallel b(\overrightarrow{W})}$$

as required.

□

Progress is similar to plain λ_{ch}; the only way $\mathbf{choose}\, a\, b$ cannot reduce is if queues a and b are both empty.

Relation to λ_{act}. Emulating such a construct satisfactorily in λ_{act} is nontrivial, because messages must be multiplexed through a local queue. One approach could be to use the work of Chaudhuri [34] which shows how to implement generalised choice using synchronous message passing, but implementing this in λ_{ch} may be difficult due to the asynchrony of **give**. We leave a more thorough investigation to future work.

6.5 Summary

In this chapter, we have investigated three extensions to λ_{ch} and λ_{act}: actor synchronisation, selective receive, and choice. The extensions are interesting to investigate in the context of translations between the two calculi. Synchronisation simplifies the translation from λ_{ch}: synchronisation, and therefore bypassing mailboxes, alleviates the type pollution problem and thus removes the restriction that all channels in a system must have the same type prior to translation. Emulating selective receive in λ_{ch} does *not* require any additional communication or concurrency primitives in λ_{ch}, since it can be emulated directly in λ_{act} (albeit with a global transformation). Finally, it is difficult to see how input-guarded choice in λ_{ch} could be emulated using λ_{act}, due to the requirement of locality in actor systems.

Chapter 7

Discussion

We conclude Part II by summarising the results of Chapters 4–6, and discussing related and future work.

In Chapter 4, we introduced two concurrent λ-calculi, λ_{ch} and λ_{act}, describing languages which include typed channels and type-parameterised actors, respectively. We showed that they satisfied preservation, and classified the notion of progress that both enjoy.

In Chapter 5, we showed a simple (yet global) translation from λ_{act} into λ_{ch}, and a more complex (yet local) translation from λ_{ch} into λ_{act}. Notably, it is only possible to translate from λ_{ch} into λ_{act} when all channels have the same type, which can be done by *coalescing* the channel types into a single recursive variant type. Additionally, the translation requires sums, products, recursive functions, and recursive types.

In Chapter 6, we discussed three extensions to λ_{ch} and λ_{act}. Adding synchronisation simplifies the translation from λ_{ch} into λ_{act}, in particular by removing the coalescing requirement. Erlang-style selective receive can in fact be simulated by λ_{act}, given a global translation and the extended term language. Input-guarded choice can be straightforwardly added to λ_{ch}, but emulating it satisfactorily in λ_{act} appears challenging.

Table 7.1 gives an overview of the translations.

Translation	**Global / Local**	**Coalescing?**	**Extended Language?**
λ_{act} to λ_{ch}	Global	No	No
λ_{ch} to λ_{act}	Local	Yes	Yes
λ_{ch} to λ_{act} + **wait**	Local	No	Yes
λ_{act} + selective receive to λ_{act}	Global	No	Yes

Table 7.1: Overview of translations

7.1 Related work

Our formulation of concurrent λ-calculi is inspired by λ(fut) [156], a concurrent λ-calculus with threads, futures, reference cells, and an atomic exchange construct. In the presence of lists, futures are sufficient to encode asynchronous channels. In λ_{ch}, we concentrate on asynchronous channels to better understand the correspondence with actors. Channel-based concurrent λ-calculi form the basis of functional languages with session types [77, 132].

Concurrent ML [186] extends Standard ML with a rich set of combinators for synchronous channels, which again can emulate asynchronous channels. A core notion in Concurrent ML is nondeterministically synchronising on multiple synchronous events, such as sending or receiving messages; relating such a construct to an actor calculus is nontrivial, and remains an open problem. Hopac [101] is a channel-based concurrency library for F#, based on Concurrent ML. The Hopac documentation relates synchronous channels and actors [3], implementing actor-style primitives using channels, and channel-style primitives using actors. The implementation of channels using actors uses mutable references to emulate the **take** function, whereas our translation achieves this using message passing (however, our translation using **wait** is more closely related to the Hopac translation). Additionally, our translation is formalised and we prove that the translations are type- and semantics-preserving.

Links [46] provides actor-style concurrency, and the paper describes a translation into λ(fut). Our translation is semantics-preserving and can be done without synchronisation.

The actor model was designed by Hewitt et al. [87] and examined in the context of distributed systems by Agha [5]. Agha et al. [6] describe a functional actor calculus based on the λ-calculus augmented by three core constructs: send sends a message; letactor creates a new actor; and become changes an actor's behaviour. The operational semantics is defined in terms of a global actor mapping, a global multiset of messages, a set of *receptionists* (actors which are externally visible to other configurations), and a set of external actor names. Instead of become, we use an explicit **receive** construct, which more closely resembles Erlang (referred to by the authors as "essentially an actor language"). Our concurrent semantics, more in the spirit of process calculi, encodes visibility via name restrictions and structural congruences. The authors consider a behavioural theory in terms of operational and testing equivalences—something we have not investigated. A starting point would be adapting the labelled-transition system semantics for the π-calculus [189] to λ_{ch} and λ_{act} configurations; we could then define bisimulation relations in the standard way.

Scala has native support for actor-style concurrency, implemented efficiently without explicit virtual machine support [84]. The actor model inspires *active objects* [128]: objects supporting asynchronous method calls which return responses using futures. De Boer et al. [55] describe a language for active objects with cooperatively scheduled threads within each object. Core ABS [114] is a specification language based on active objects. Using futures

for synchronisation sidesteps the type pollution problem inherent in call-response patterns with actors, although our translations work in the absence of synchronisation. By working in the functional setting, we obtain more compact calculi.

Since *Mixing Metaphors: Actors as Channels and Channels as Actors* was published at ECOOP 2017, several projects have expanded upon the core calculi and ideas. Priya [180] introduces λir, a concurrent λ-calculus inspired by λ_{act}; a key innovation is to expand the **receive** construct with support for *intensional*, or dynamic type checking. The author additionally introduces a finer-grained type discipline inspired by *process types* as introduced by Igarashi and Kobayashi [108].

Krafft [126] encodes λ_{act} as a shallowly-embedded domain-specific language in the Agda [157] programming language, using a monadic encoding. Krafft formally proves properties such as weak progress, and additionally encodes selective receive as both a language primitive and as a library routine, as well as encoding simply-typed channels and active objects.

In the original ECOOP paper, we speculated on a design for behaviourally-typed actors based on a parameterised monad, however this proved too restrictive. We were missing a key insight; the unidirectional nature of actor mailboxes makes behavioural types better suited to *unordered* interactions. de'Liguoro and Padovani [57] introduce the *Mailbox Calculus*, a small process calculus with first-class mailboxes, selective receive, and nondeterministic choice. The type system guarantees that a process will never receive a message that it cannot receive, and via a *dependency graph*, outlaws deadlocking configurations. The authors demonstrate the expressiveness of the calculus by encoding several benchmarks from the Savina actor benchmark suite [111]. Additionally, the authors show an example of encoding binary sessions through the use of an arbiter process.

7.2 Conclusion

Inspired by languages such as Go which take channels as core constructs for communication, and languages such as Erlang which are based on the actor model of concurrency, we have presented translations back and forth between a concurrent λ-calculus λ_{ch} with channel-based communication constructs and a concurrent λ-calculus λ_{act} with actor-based communication constructs. We have proved that λ_{act} can simulate λ_{ch} and vice-versa.

The translation from λ_{act} to λ_{ch} is straightforward, whereas the translation from λ_{ch} to λ_{act} requires considerably more effort. The relative difficulty of the encodings reflects the asymmetry of Figure 4.2 in Chapter 4.

We have also shown how to extend λ_{act} with synchronisation, greatly simplifying the translation from λ_{ch} into λ_{act}, and have shown how Erlang-style selective receive can be emulated in λ_{act}. Finally, we have discussed input-guarded choice in λ_{ch}.

An interesting further line of study would be to relate different incarnations of the actor model as defined by De Koster et al. [56]. Following on from the work of de'Liguoro and Padovani [57], it would be interesting to more formally specify the relation between the Mailbox Calculus and session-typed channels, and investigate how the ideas from the Mailbox Calculus could be integrated with a functional programming language.

Part III

Session Types without Tiers

Chapter 8

Asynchronous GV

8.1 Introduction

Synchronous GV (Chapter 3) is a core session-typed functional language with a strong correspondence to classical linear logic, and correspondingly a host of strong metatheoretic properties. However, SGV has a *synchronous* communication semantics, where communication requires a synchronous rendezvous between two processes over a channel. In contrast, programming languages such as Erlang provide *asynchronous* communication, where sending does not block. Indeed, asynchronous communication is more amenable to implementation in the distributed setting, and synchronous communication can be described in terms of asynchronous communication through the use of acknowledgements (see Sangiorgi and Walker [189], p. 204).

In this section, we describe Asynchronous GV (AGV), an extension of SGV with an asynchronous communication semantics, while retaining SGV's strong metatheory. This gets us a step closer to our goal of a core calculus for a distributed extension of Links with session-typed communication.

Example. Consider the case where a thread exchanges a single value with a second thread. Writing $\Longrightarrow$ to mean reduction modulo equivalence, in SGV, a reduction would take place as follows:

$$\bullet E[\mathbf{send}\,5\,a] \parallel \circ E'[\mathbf{receive}\,a] \Longrightarrow \bullet E[a] \parallel \circ E'[(5,a)]$$

Here, we have that there is a *rendezvous* between sender and receiver; communication is *synchronous* as the sender may not proceed until the communication action has taken place successfully.

In Asynchronous GV, the interaction would take place as follows:

$$\begin{aligned}
&\bullet E[\textbf{send}\,5\,a] \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon) \parallel \circ E[\textbf{receive}\,b] \\
&\quad\Longrightarrow \\
&\bullet E[a] \parallel a(\varepsilon) \leftrightsquigarrow b(5) \parallel \circ E[\textbf{receive}\,b] \\
&\quad\Longrightarrow \\
&\bullet E[a] \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon) \parallel \circ E[(5, b)]
\end{aligned}$$

Instead of the processes communicating directly, we have a *buffer process* $a(\varepsilon) \leftrightsquigarrow b(\varepsilon)$ which contains queues for endpoints a and b. Sending a value V along endpoint a results in V being appended to the queue for endpoint b, and vice-versa; receiving from endpoint b results in a value being retrieved from the head of the queue for endpoint b. As communication is buffered, evaluation of the sender thread need not block until the peer is ready to receive the communicated value.

8.2 Asynchronous GV

In this section, we define the syntax and semantics of Asynchronous GV.

8.2.1 Syntax and Typing of Terms

Figure 8.1 shows the syntax of Asynchronous GV. It may seem familiar to that of SGV! In fact, the static syntax of AGV is exactly the same; the differences are only in the runtime syntax, reduction rules, and runtime typing.

Let us briefly reprise the syntax of terms and types. Asynchronous GV is based on a linear λ calculus. Its syntax consists of types, ranged over by A, B, C, which consist of the unit type $\mathbf{1}$; linear functions $A \multimap B$; linear sums $A + B$; linear tensor products $A \times B$; and session types S. Session types S type channel endpoints, and consist of output types $!A.S$ (read "output a value of type A and continue as type S"); input types $?A.S$ (read "input a value of type A and continue as type S"); and $\mathsf{End}_!$ and $\mathsf{End}_?$ which denote that a session has ended. Again, as in Chapter 3, we do not include branching and selection in the core calculus as they can be encoded using sum types and delegation, following Dardha et al. [54].

Terms, ranged over by L, M, N include variables x, and the introduction and elimination forms for the unit value, products, and sum types. Primitives in red are the communication and concurrency constructs for the language: $\textbf{fork}\,M$ creates a name a and spawns $M\,a$ as a new thread; $\textbf{send}\,M\,N$ sends M along endpoint N and returns the updated channel endpoint; $\textbf{receive}\,M$ receives from endpoint M and returns a pair of the received value and the updated endpoint; and $\textbf{wait}\,M$ synchronises with a child thread when the session has finished, in turn providing an elimination form for values of type $\mathsf{End}_?$.

$$
\begin{array}{lll}
\text{Types} & A,B,C & ::= \mathbf{1} \mid A \multimap B \mid A + B \mid A \times B \mid S \\
\text{Session Types} & S & ::= !A.S \mid ?A.S \mid \mathsf{End}_! \mid \mathsf{End}_? \\
\text{Variables} & x,y,z & \\
\text{Terms} & L,M,N & ::= x \mid \lambda x.M \mid M\,N \\
 & & \quad \mid () \mid \mathbf{let}\,() = M\,\mathbf{in}\,N \\
 & & \quad \mid (M,N) \mid \mathbf{let}\,(x,y) = M\,\mathbf{in}\,N \\
 & & \quad \mid \mathbf{inl}\,M \mid \mathbf{inr}\,M \mid \mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \\
 & & \quad \mid \mathbf{fork}\,M \mid \mathbf{send}\,M\,N \mid \mathbf{receive}\,M \mid \mathbf{wait}\,M \\
\text{Type Environments} & \Gamma & ::= \cdot \mid \Gamma, x : A
\end{array}
$$

Figure 8.1: Syntax of Asynchronous GV Types and Terms

Typing Rules for Terms Typing rules for AGV terms (Figure 8.2) are the standard rules for a linear λ-calculus with sums and products, extended with primitives for session typing.

8.2.2 Operational Semantics

Runtime Syntax Figure 8.3 shows the runtime syntax of AGV, and is the first substantial departure from SGV. Runtime syntax which is new in AGV is shaded.

Names and Environments. Like SGV, we extend the static syntax of AGV with runtime names, ranged over by a, b.

We extend type environments Γ to include runtime names which have session types, and introduce runtime type environments Δ, which type both buffer endpoints of session type S and channels of type $S^\sharp$ for some S. Note that we do not use Δ to type variables; indeed Δ is only used for typing *configurations* and not terms.

Configurations. The concurrent behaviour of AGV is described using a language of configurations: $(\nu a)\mathcal{C}$ binds name a in $\mathcal{C}$; $\mathcal{C} \parallel \mathcal{D}$ describes $\mathcal{C}$ as running in parallel with $\mathcal{D}$; and ϕM describes a term M running as either a child thread $\circ M$ or a main thread $\bullet M$.

To support asynchrony, AGV includes *buffer processes* $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$. A buffer process consists of two queues: a queue for endpoint a containing a sequence of values $\overrightarrow{V}$, and a queue for endpoint b containing a sequence of values $\overrightarrow{W}$. Receiving along an endpoint V will retrieve the head value of $\overrightarrow{V}$, whereas sending along endpoint a will append a value to $\overrightarrow{W}$. As we will see in §8.3.1, the typing rules ensure that at least one of $\overrightarrow{V}$ or $\overrightarrow{W}$ is empty.

We let $\mathcal{A}$ range over *auxiliary threads*, including child threads and buffers.

Term Typing $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Var}\ \frac{}{x:A \vdash x:A} \qquad \textsc{T-Abs}\ \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M : A \multimap B} \qquad \textsc{T-App}\ \frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M\,N : B}$$

$$\textsc{T-Unit}\ \frac{}{\cdot \vdash () : \mathbf{1}} \qquad \textsc{T-LetUnit}\ \frac{\Gamma_1 \vdash M : \mathbf{1} \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\,() = M\,\mathbf{in}\,N : A}$$

$$\textsc{T-Pair}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash (M, N) : A \times B} \qquad \textsc{T-LetPair}\ \frac{\Gamma_1 \vdash M : A \times B \quad \Gamma_2, x:A, y:B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\,(x,y) = M\,\mathbf{in}\,N : C}$$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}\,M : A + B} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}\,M : A + B} \qquad \textsc{T-Case}\ \frac{\Gamma_1 \vdash L : A + B \quad \Gamma_2, x:A \vdash M : C \quad \Gamma_2, y:B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} : C}$$

$$\textsc{T-Fork}\ \frac{\Gamma \vdash M : S \multimap \mathsf{End}_!}{\Gamma \vdash \mathbf{fork}\,M : \overline{S}} \qquad \textsc{T-Send}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : !A.S}{\Gamma_1, \Gamma_2 \vdash \mathbf{send}\,M\,N : S} \qquad \textsc{T-Recv}\ \frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{receive}\,M : (A \times S)}$$

$$\textsc{T-Wait}\ \frac{\Gamma \vdash M : \mathsf{End}_?}{\Gamma \vdash \mathbf{wait}\,M : \mathbf{1}}$$

Duality $\boxed{\overline{S}}$

$$\overline{!A.S} = ?A.\overline{S} \qquad \overline{?A.S} = !A.\overline{S} \qquad \overline{\mathsf{End}_!} = \mathsf{End}_? \qquad \overline{\mathsf{End}_?} = \mathsf{End}_!$$

Figure 8.2: Term Typing and Duality

Runtime Types	R	$::= S \mid S^\sharp$
Names	a, b, c	
Terms	M	$::= \cdots \mid a$
Values	U, V, W	$::= a \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\, V \mid \mathbf{inr}\, V$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	$::= (\nu a)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M \mid a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$
Thread Flags	ϕ	$::= \bullet \mid \circ$
Auxiliary threads	$\mathcal{A}$	$::= \circ M \mid a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$
Type Environments	Γ	$::= \cdots \mid \Gamma, a : S$
Runtime Type Environments	Δ	$::= \cdot \mid \Delta, a : R$
Evaluation Contexts	E	$::= [\,] \mid E\, M \mid V\, E$
		$\mid \mathbf{let}\,() = E\,\mathbf{in}\,M \mid \mathbf{let}\,(x, y) = E\,\mathbf{in}\,M \mid (E, V) \mid (V, E)$
		$\mid \mathbf{inl}\, E \mid \mathbf{inr}\, E \mid \mathbf{case}\, E\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}$
		$\mid \mathbf{fork}\, E \mid \mathbf{send}\, E\, M \mid \mathbf{send}\, V\, E \mid \mathbf{receive}\, E \mid \mathbf{wait}\, E$
Thread Contexts	$\mathcal{F}$	$::= \phi E$
Configuration Contexts	$\mathcal{G}$	$::= [\,] \mid (\nu a)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$

Figure 8.3: Runtime Syntax

Term reduction. Figure 8.4 describes the reduction rules for AGV terms, reduction rules, and equivalence axioms for AGV configurations. Reduction on terms is again standard β-reduction, with evaluation contexts set up for call-by-value, left-to-right evaluation.

Equivalence. Equivalence axioms are standard except for one additional rule, which allows us to treat buffers symmetrically.

Reduction on Configurations. The reduction relation on configurations is substantially different to Synchronous GV in order to account for asynchronous communication. Thread contexts $\mathcal{F}$ abstract over thread flags. E-Fork evaluates $\mathcal{F}[\mathbf{fork}\,\lambda x.M]$ by creating *two* fresh names, a and b, for each queue in the buffer. It returns a in the calling context, and spawns $M\{b/x\}$ as a child thread, and an empty buffer $a(\varepsilon) \leftrightsquigarrow b(\varepsilon)$. The communication topology remains acyclic.

Due to asynchrony, we replace the rule E-Comm with rules E-Send and E-Receive, as sending and receiving no longer have to occur at the same time. Rule E-Send describes sending a message: a thread $\mathcal{F}[\mathbf{send}\, U\, a]$ sending value U along endpoint a in parallel with a buffer $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ returns the updated endpoint a in the calling context, and appends U to the end of b's queue in the buffer. Conversely, E-Receive details dequeueing a value from the head of a

Term Reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$

$$\begin{array}{lr}
\textsc{E-Lam} & (\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\textsc{E-Unit} & \mathbf{let}\,() = ()\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M \\
\textsc{E-Pair} & \mathbf{let}\,(x,y) = (V,W)\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M\{V/x,W/y\} \\
\textsc{E-Inl} & \mathbf{case}\,\mathbf{inl}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\textsc{E-Inr} & \mathbf{case}\,\mathbf{inr}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} N\{V/y\} \\
\textsc{E-Lift} & E[M] \longrightarrow_{\mathsf{M}} E[M'], \quad \text{if } M \longrightarrow_{\mathsf{M}} M'
\end{array}$$

Configuration Equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad (\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

$$\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}), \quad \text{if } a \notin \mathsf{fn}(\mathcal{C}) \qquad a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \equiv b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V})$$

Configuration Reduction $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$\begin{array}{lr}
\textsc{E-Fork} & \mathcal{F}[\mathbf{fork}\,(\lambda x.M)] \longrightarrow (\nu a)(\nu b)(\mathcal{F}[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)), \quad \text{where } a,b \text{ are fresh} \\
\textsc{E-Send} & \mathcal{F}[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \\
\textsc{E-Receive} & \mathcal{F}[\mathbf{receive}\,a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[(U,a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\
\textsc{E-Wait} & (\nu a)(\nu b)(\mathcal{F}[\mathbf{wait}\,a] \parallel \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)) \longrightarrow \mathcal{F}[()] \\
\textsc{E-LiftC} & \mathcal{G}[\mathcal{C}] \longrightarrow \mathcal{G}[\mathcal{D}], \quad \text{if } \mathcal{C} \longrightarrow \mathcal{D} \\
\textsc{E-LiftM} & \phi M \longrightarrow \phi N, \quad \text{if } M \longrightarrow_{\mathsf{M}} N
\end{array}$$

Figure 8.4: Reduction and Equivalence for Terms and Configurations

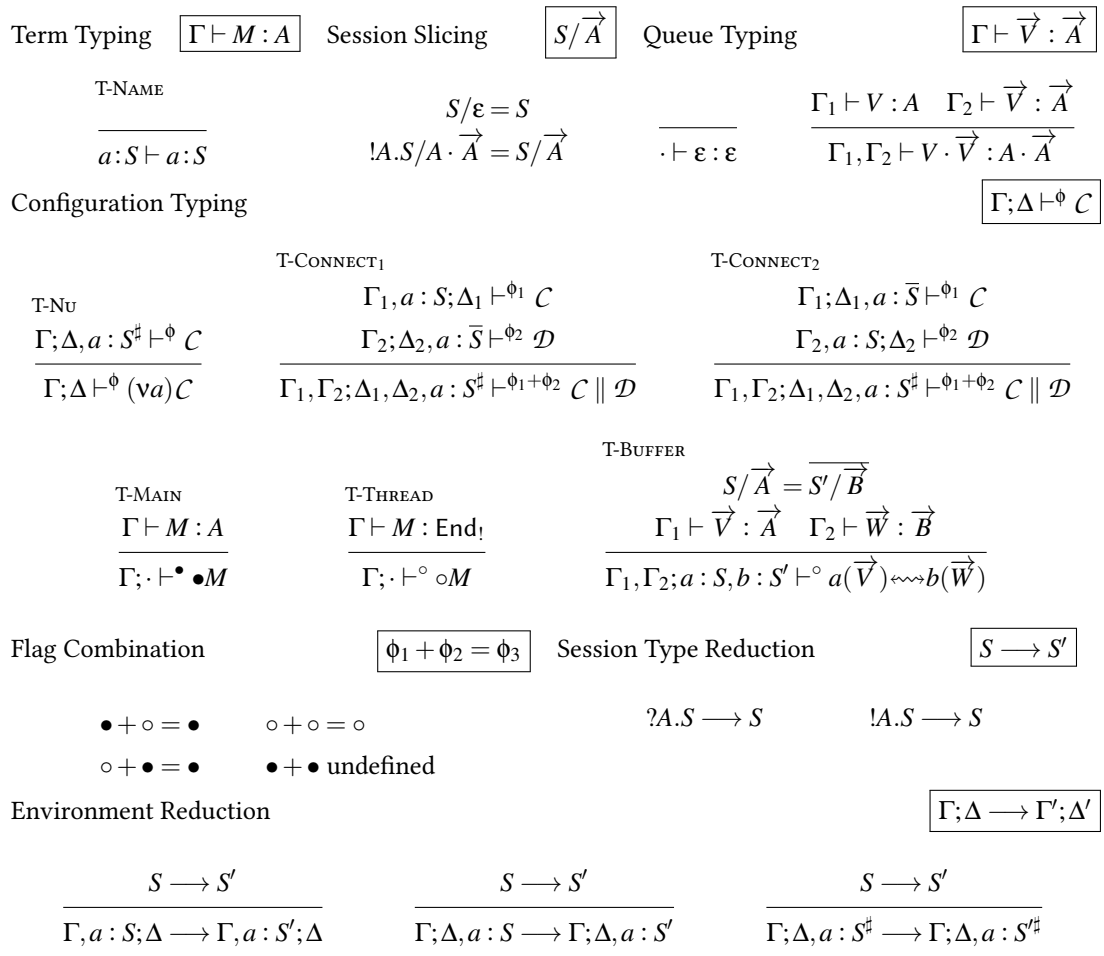


Figure 8.5: Runtime Typing

buffer, returning a pair of the received value and the updated endpoint. Rule E-Wait eliminates the names, the child thread, and the empty buffer of a finished session. Rules E-LiftC and E-LiftM are as before.

8.3 Metatheory

In this section, we describe the metatheory of AGV. We begin by defining a *runtime type system*, which encodes the invariants satisfied by well-typed AGV configurations and thus allows us to prove dynamic properties about the system.

In Chapter 3, we saw that SGV had a strong metatheory, enjoying preservation, deadlock-freedom, global progress, confluence, and termination. AGV retains all of these core properties, but the runtime type system becomes more involved to account for buffered communication.

8.3.1 Runtime Typing

Figure 8.5 shows the runtime typing rules for AGV. The typing judgement $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$ can be read, "under term environment Γ, runtime type environment Δ, and flag ϕ, configuration $\mathcal{C}$ is well-typed". As in Synchronous GV, a flag ϕ details whether a configuration contains a main thread, and the configuration typing rules require that each configuration has at most one main thread. We additionally require that in any derivation of $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$, that $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$.

We write $\Gamma;\Delta \vdash^{\bullet} \mathcal{C} : A$ if the derivation of $\Gamma;\Delta \vdash^{\bullet} \mathcal{C}$ contains a subderivation of the form

$$\frac{}{\Gamma';\cdot \vdash^{\bullet} \bullet M : A}$$

It is again helpful to define the notion of a *ground configuration*.

Definition 8 (Ground Configuration)**.** *A configuration $\mathcal{C}$ is a* ground configuration *if $\Gamma;\Delta \vdash^{\bullet} \mathcal{C} : A$ for some Γ, Δ, A, where A contains no function types or session types.*

Runtime Typing Rules Rule T-Nu introduces a runtime name a with type $S^{\sharp}$ into runtime type environment Δ. Each runtime name can be split with rules T-Connect$_1$ or T-Connect$_2$, allowing each name to be used once as an endpoint (in Γ) and once as part of a buffer (in Δ).

Rule T-Main describes a main thread, which may return a value, and rule T-Child describes a child thread, which must return a channel of type $\mathsf{End}_!$.

Finally, T-Buffer types a buffer between two processes. For a buffer $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ to be well-typed, the runtime type environment must contain entries $a : S$ and $b : T$, where S and T are related by the equation $S/\overrightarrow{A} = \overline{T/\overrightarrow{B}}$, where $S/\overrightarrow{A}$ is the *session slicing* operator, allowing us to consider duality of session types up to values contained in the buffer. The partiality of the session slicing operator coupled with the duality constraint ensures that at least $\overrightarrow{V}$ or $\overrightarrow{W}$ is empty.

Reduction on Session Types. The relation $S \longrightarrow S'$ describes the evolution of session types: more specifically, $?A.S \longrightarrow S$ when a receive action is performed, and $!A.S \longrightarrow S$ when a send action is performed. We extend reduction on session types to reduction on typing environments $\Gamma;\Delta \longrightarrow \Gamma';\Delta'$.

Example. It is helpful to consider a concrete example. Figure 8.6 shows an example typing derivation for the configuration $(\nu a)(\nu b)(\circ E[\mathbf{send}\,5\,a] \parallel (a(\varepsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\,b]))$, which is similar to the configuration discussed in §8.1, but with true already contained in the queue for endpoint b.

Of particular interest is the subderivation for typing a buffer. We abuse notation slightly, writing $\cdot \vdash \mathsf{true} : \mathsf{Bool}$ to mean $\cdot \vdash \mathsf{true} \cdot \varepsilon : \mathsf{Bool} \cdot \varepsilon$.

$$\frac{?\mathsf{Int}.\mathsf{End}_? / \varepsilon = \overline{!\mathsf{Bool}.!\mathsf{Int}.\mathsf{End}_! / \mathsf{Bool}} \quad \cdot \vdash \varepsilon : \varepsilon \quad \cdot \vdash \mathsf{true} : \mathsf{Bool}}{\cdot; a : ?\mathsf{Int}.\mathsf{End}_?, b : !\mathsf{Bool}.!\mathsf{Int}.\mathsf{End}_! \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\mathsf{true})}$$

Here, we have that the queue for endpoint b already contains a value true of type Bool, and we have that the buffer endpoint a is ready to receive a value of type Int to store in the queue for endpoint b. The buffer endpoint b is ready to transmit a value of type Bool to the thread holding endpoint b. Thus, the types for endpoints a and b are related by the equation $?\mathsf{Int}.\mathsf{End}_? / \varepsilon = \overline{!\mathsf{Bool}.!\mathsf{Int}.\mathsf{End}_! / \mathsf{Bool}}$.

We can show that this equation holds as follows:

$$\begin{aligned}
&?\mathsf{Int}.\mathsf{End}_? / \varepsilon = \overline{!\mathsf{Bool}.!\mathsf{Int}.\mathsf{End}_! / \mathsf{Bool}} \\
&\quad \iff (\text{as } \overline{!\mathsf{Bool}.!\mathsf{Int}.\mathsf{End}_! / \mathsf{Bool}} = !\mathsf{Int}.\mathsf{End}_!) \\
&?\mathsf{Int}.\mathsf{End}_? / \varepsilon = \overline{!\mathsf{Int}.\mathsf{End}_!} \\
&\quad \iff (\text{as } ?\mathsf{Int}.\mathsf{End}_? / \varepsilon = ?\mathsf{Int}.\mathsf{End}_?) \\
&?\mathsf{Int}.\mathsf{End}_? = \overline{!\mathsf{Int}.\mathsf{End}_!} \\
&\quad \iff (\text{by definition of duality}) \\
&?\mathsf{Int}.\mathsf{End}_? = ?\mathsf{Int}.\mathsf{End}_?
\end{aligned}$$

$$
\dfrac{
\dfrac{\Gamma_1, a : {!}\mathsf{Int}.\mathsf{End}_{!} \vdash^{\circ} E[\mathbf{send}\,5\,a] : \mathsf{End}_{!}}
{\Gamma_1, a : {!}\mathsf{Int}.\mathsf{End}_{!}; \cdot \vdash^{\circ} E[\mathbf{send}\,5\,a]}\,\textsc{T-Thread}
\quad
\dfrac{
\dfrac{{?}\mathsf{Int}.\mathsf{End}_{?}/\varepsilon = \overline{{!}\mathsf{Bool}.{!}\mathsf{Int}.\mathsf{End}_{!}/\mathsf{Bool}} \quad \cdot \vdash \varepsilon : \varepsilon \quad \cdot \vdash \mathsf{true} : \mathsf{Bool}}
{\cdot; a : {?}\mathsf{Int}.\mathsf{End}_{?}, b : {!}\mathsf{Bool}.{!}\mathsf{Int}.\mathsf{End}_{!} \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\mathsf{true})}\,\textsc{T-Buffer}
\quad
\dfrac{\Gamma_2, b : {?}\mathsf{Bool}.{?}\mathsf{Int}.\mathsf{End}_{?} \vdash E'[\mathbf{receive}\,b] : A}
{\Gamma_2, b : {?}\mathsf{Bool}.{?}\mathsf{Int}.\mathsf{End}_{?} \vdash^{\bullet} \bullet E'[\mathbf{receive}\,b]}\,\textsc{T-Main}
}
{\Gamma_2; a : {?}\mathsf{Int}.\mathsf{End}_{?}, b : ({?}\mathsf{Bool}.{?}\mathsf{Int}.\mathsf{End}_{?})^{\sharp} \vdash^{\bullet} a(\varepsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\,b]}\,\textsc{T-Compose}_2
}
{
\dfrac{
\Gamma_1, \Gamma_2; a : ({!}\mathsf{Int}.\mathsf{End}_{!})^{\sharp}, b : ({?}\mathsf{Bool}.{?}\mathsf{Int}.\mathsf{End}_{?})^{\sharp} \vdash^{\circ} \circ E[\mathbf{send}\,5\,a] \parallel (a(\varepsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\,b])
}
{
\dfrac{\Gamma_1, \Gamma_2; a : ({!}\mathsf{Int}.\mathsf{End}_{!})^{\sharp} \vdash^{\bullet} (\nu b)(\circ E[\mathbf{send}\,5\,a] \parallel (a(\varepsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\,b]))}
{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} \bullet(\nu a)(\nu b)(\circ E[\mathbf{send}\,5\,a] \parallel (a(\varepsilon) \leftrightsquigarrow b(\mathsf{true}) \parallel \bullet E'[\mathbf{receive}\,b]))}\,\textsc{T-Nu}
}\,\textsc{T-Nu}
}\,\textsc{T-Compose}_1
$$

Figure 8.6: Typing a buffer in AGV

We may now show that the addition of asynchrony does not violate any of the properties enjoyed by SGV.

8.3.2 Preservation

To prove preservation, we begin by adapting the auxiliary lemmas from SGV required to reason about the manipulation of evaluation and configuration contexts in the preservation proof. All are standard, but modified to take the runtime typing environment Δ into account.

Lemma 46 (Substitution)**.** *Suppose* $\Gamma_1, x : B \vdash M : A$ *and* $\Gamma_2 \vdash N : B$*, where* Γ_1, Γ_2 *is defined. Then* $\Gamma_1, \Gamma_2 \vdash M\{N/x\} : A$*.*

Proof. By induction on the derivation of $\Gamma_1, x : B \vdash M : A$. □

Next, we show subterm typeability and replacement. The proofs both follow by induction on the structure of E.

Lemma 47 (Subterm typeability)**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : A$*, then there exists some subderivation* $\mathbf{D}'$ *of* $\mathbf{D}$ *concluding* $\Gamma_2 \vdash M : B$*, where the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* E*.*

Lemma 48 (Subterm replacement)**.** *If:*

- $\mathbf{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : A$
- $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding* $\Gamma_2 \vdash M : B$
- *The position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* E
- $\Gamma_3 \vdash N : B$
- Γ_1, Γ_3 *is defined*

then $\Gamma_1, \Gamma_3 \vdash E[N] : A$*.*

Finally, we can show subconfiguration typeability and replacement.

Lemma 49 (Subconfiguration typeability)**.** *If* $\mathbf{D}$ *is a derivation of* $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$*, then there exist* Γ', Δ', ϕ' *such that* $\mathbf{D}$ *has a subderivation* $\mathbf{D}'$ *that concludes* $\Gamma'; \Delta' \vdash^{\phi'} \mathcal{C}$ *and the position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to the position of the hole in* $\mathcal{G}$*.*

Lemma 50 (Subconfiguration replacement)**.** *If:*

- $\mathbf{D}$ *is a derivation of* $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$
- $\mathbf{D}'$ *is a subderivation of* $\mathbf{D}$ *concluding that* $\Gamma'; \Delta' \vdash^{\phi'} \mathcal{C}$ *for some* Γ', Δ', ϕ'

- *The position of* $\mathbf{D}'$ *in* $\mathbf{D}$ *corresponds to that of the hole in* $\mathcal{G}$
- $\Gamma'';\Delta'' \vdash^{\phi'} \mathcal{D}$ *for some* Γ'',Δ'' *such that* $\Gamma';\Delta' \longrightarrow^{?} \Gamma'';\Delta''$

then there exist some $\Gamma''';\Delta'''$ *such that* $\Gamma;\Delta \longrightarrow^{?} \Gamma''';\Delta'''$ *and* $\Gamma''';\Delta''' \vdash^{\phi} \mathcal{G}[\mathcal{D}]$.

As before, term reduction preserves typing.

Lemma 51 (Preservation (AGV Terms))**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M N$*, then* $\Gamma \vdash N : A$.

Proof. By induction on the derivation of $M \longrightarrow_M N$. □

With the auxiliary results defined, we can see that reduction on configurations preserves configuration typeability.

Theorem 25 (Preservation (AGV Configurations))**.** *If* $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then there exist some* Γ',Δ' *such that* $\Gamma;\Delta \longrightarrow^{?} \Gamma';\Delta'$ *and* $\Gamma';\Delta' \vdash^{\phi} \mathcal{D}$.

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$. The full proof can be found in Appendix C. We show the case for E-Send here, making the choice to prove the case where $\phi = \bullet$. The case where $\phi = \circ$ is similar.

Case E-Send

Assumption:

$$\frac{\dfrac{\Gamma_1,\Gamma_2,a : S \vdash E[\mathbf{send}\,U\,a] : C}{\Gamma_1,\Gamma_2,a : S;\cdot \vdash^{\bullet} \bullet E[\mathbf{send}\,U\,a]} \quad \dfrac{\overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3,\Gamma_4;a : \overline{S},b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : S^{\sharp},b : T \vdash^{\bullet} \bullet E[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma 47:

$$\frac{\Gamma_2 \vdash U : A \quad a : !A.S' \vdash a : !A.S'}{\Gamma_2,a : !A.S' \vdash \mathbf{send}\,U\,a : S'}$$

Thus, $S = !A.S'$, and $\overline{S} = ?A.\overline{S'}$ for some S'. We may therefore refine our original derivation:

$$\frac{\dfrac{\Gamma_1,\Gamma_2,a : !A.S' \vdash E[\mathbf{send}\,U\,a] : C}{\Gamma_1,\Gamma_2,a : !A.S';\cdot \vdash^{\bullet} \bullet E[\mathbf{send}\,U\,a]} \quad \dfrac{?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3,\Gamma_4;a : ?A.\overline{S'},b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : !A.S'^{\sharp},b : T \vdash^{\bullet} \bullet E[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Since $?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}}$ is defined, we have that $\overrightarrow{A} = \varepsilon$. By the definition of slicing, we have that $\overline{T} = \overline{!B_1.\cdots.!B_n.!A.S'}$ for each B_i. It follows that $\overline{S'}/\overrightarrow{A} = \overline{T}/\overrightarrow{B} \cdot A$.

By Lemma 48, we have $\Gamma_1,\Gamma_2,a : S' \vdash E[a] : C$.

Reconstructing:

$$\dfrac{\dfrac{\Gamma_1, a : S' \vdash E[a] : C}{\Gamma_1, a : S'; \cdot \vdash^\bullet \bullet E[a]} \quad \dfrac{\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B} \cdot A} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_2, \Gamma_4 \vdash \overrightarrow{W} \cdot U : \overrightarrow{B} \cdot A}{\Gamma_2, \Gamma_3, \Gamma_4; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U)}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T \vdash^\bullet \bullet E[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U)}$$

Finally, we must show environment reduction:

$$\dfrac{!A.S' \longrightarrow S'}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : !A.S'^\sharp, b : T \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T}$$

as required.

□

Configuration Typing and Equivalence. Similar to SGV, the $\longrightarrow$ relation is *not* defined modulo equivalence, and typeability is *not* preserved by equivalence. The canonical example is of a configuration $\Gamma; \Delta \vdash^\phi (\nu a)(\nu b)(\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}))$, where $a \in \mathsf{fn}(\mathcal{C})$, $b \in \mathsf{fn}(\mathcal{D})$, and $a, b \in \mathsf{fn}(\mathcal{E})$.

As before, only associativity of parallel composition is problematic for typeability, and it is always possible to reassociate parallel composition without breaking typeability either directly, or by firstly commuting a subconfiguration.

Lemma 52. *If $\Gamma; \Delta \vdash^\phi \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{D}$, where the derivation of $\mathcal{C} \equiv \mathcal{D}$ does not contain a use of the axiom for associativity, then $\Gamma; \Delta \vdash^\phi \mathcal{D}$.*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$. The proof is mostly identical to the analogous proof in SGV, but we must consider the additional equivalence axiom which allows us to treat buffers symmetrically.

Case $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \equiv b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V})$

$$\dfrac{S/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_1 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2; a : S, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})} \iff \dfrac{T/\overrightarrow{B} = \overline{S/\overrightarrow{A}} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B} \quad \Gamma_1 \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma_1, \Gamma_2; a : S, b : T \vdash^\circ b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V})}$$

The above holds because $S/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \iff T/\overrightarrow{B} = \overline{S/\overrightarrow{A}}$:

$$\begin{aligned}
& S/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \\
& \quad \iff \text{(duality)} \\
& \overline{S/\overrightarrow{A}} = \overline{\overline{T/\overrightarrow{B}}} \\
& \quad \iff \text{(duality is involutive)} \\
& \overline{S/\overrightarrow{A}} = T/\overrightarrow{B} \\
& \quad \iff \text{(equality is symmetric)} \\
& T/\overrightarrow{B} = \overline{S/\overrightarrow{A}}
\end{aligned}$$

□

Lemma 53 (Associativity)**.**

- *If* $\Gamma;\Delta \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$*, then either* $\Gamma;\Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$ *or* $\Gamma;\Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D}$*.*
- *If* $\Gamma;\Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$*, then either* $\Gamma;\Delta \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$ *or* $\Gamma;\Delta \vdash^{\phi} \mathcal{D} \parallel (\mathcal{C} \parallel \mathcal{E})$*.*

Proof. Similar to the proof of the analogous property in SGV. □

Finally, we may see that any reduction sequence which uses an ill-typed equivalence may be replaced with one which does not.

Theorem 26 (Reduction Modulo Equivalence)**.** *If* $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$*,* $\mathcal{C} \equiv \mathcal{D}$*, and* $\mathcal{D} \longrightarrow \mathcal{D}'$*, then:*

1. *There exists some* $\mathcal{E}$ *such that* $\mathcal{D} \equiv \mathcal{E}$*, and* $\Gamma;\Delta \vdash^{\phi} \mathcal{E}$*, and* $\mathcal{E} \longrightarrow \mathcal{E}'$
2. *There exist some* Γ', Δ' *such that* $\Gamma;\Delta \longrightarrow \Gamma';\Delta'$ *and* $\Gamma';\Delta' \vdash^{\phi} \mathcal{E}$
3. $\mathcal{D}' \equiv \mathcal{E}'$

Proof. Similar to the proof of the analogous property in SGV. □

8.3.3 Deadlock-freedom

Communication takes place through a buffer rather than directly between processes, but AGV still retains an acyclic topology and is therefore deadlock-free. We can therefore construct a similar graph-theoretic proof for deadlock-freedom in AGV as we did for SGV.

Before we begin formalising deadlock-freedom, it is worth emphasising that our notion of deadlock-freedom only classifies configurations which cannot reduce as a result of a cyclic dependency as deadlocked, rather than, for example, trying to send to an external or nonexistent buffer. Such a configuration does not reduce, but not as a result of deadlock. We will examine such cases in more detail in §8.3.4.

We begin by defining the notion of a *blocked* process; in AGV, sending a value is a non-blocking operation, so the only possibly-blocking communication operations are receiving a value, or synchronising on a channel where all communication has completed.

Definition 9. *We say that a term* M *is* blocked on an endpoint a*, written* $\mathit{blocked}(M, a)$*, if* M *is attempting to read from, or wait on endpoint* a*. Formally:*

$$\mathit{blocked}(a, M) \triangleq \exists E.(M = E[\textbf{receive}\, a]) \vee (M = E[\textbf{wait}\, a])$$

If a thread ϕM is blocked on some variable a, then all other free names in M *depend* on the communication over a taking place. We can also extend the notion of dependency to

configurations, taking into account transitive dependencies. For AGV, we also say that each endpoint in a buffer depends on the other endpoint; this is an overapproximation since dependencies need not (indeed, cannot!) exist in both directions, but for our purposes this does not matter.

Note that it is impossible to perform an action on any value contained within a buffer or the payload of a send operation, so we need not characterise them as a part of the dependency predicate. Formally, we may define dependency as follows:

Definition 10. *We say that a* depends on *b* in a configuration $\mathcal{C}$, *written* $\mathit{depends}(a, b, \mathcal{C})$, *in the following cases:*

- $\mathit{depends}(a, b, a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}))$
- $\mathit{depends}(a, b, b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V}))$
- $\mathit{depends}(a, b, \phi M) \triangleq \mathit{blocked}(b, M) \wedge a \in \mathsf{fn}(M)$
- $\mathit{depends}(a, b, \mathcal{C}) \triangleq \exists \mathcal{G}, \mathcal{D}, \mathcal{E}, c. \mathcal{C} \equiv \mathcal{G}[\mathcal{D} \parallel \mathcal{E}] \wedge \mathit{depends}(a, c, \mathcal{D}) \wedge \mathit{depends}(c, b, \mathcal{E})$

As before, deadlocked configurations are configurations with cyclic dependencies.

Definition 11.

$$\mathit{deadlocked}(\mathcal{C}) \triangleq \exists \mathcal{G}, \mathcal{D}, \mathcal{E}, a, b.\ \mathcal{C} \equiv \mathcal{G}[\mathcal{D} \parallel \mathcal{E}] \wedge \mathit{depends}(a, b, \mathcal{D}) \wedge \mathit{depends}(b, a, \mathcal{E})$$

Although we have added asynchrony, no additional constructs in AGV introduce cycles, and the runtime typing rules still enforce the invariant that exactly one name is shared between two configurations.

Lemma 54. *If* $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$ *and* $\exists \mathcal{D}, \mathcal{E}.\ \mathcal{C} = \mathcal{G}[\mathcal{D} \parallel \mathcal{E}]$, *then* $\mathsf{fn}(\mathcal{D}) \cap \mathsf{fn}(\mathcal{E}) = \{a\}$ *for some name* a.

Proof. By induction on the derivation of $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$, due to the partitioning of the type environment and buffer environment in the typing rules for parallel composition. The T-Connect$_1$ and T-Connect$_2$ rules allow exactly one name to be shared. □

It follows that since the configuration typing rules forbid more than one name from being shared between configurations, that well-typed configurations cannot be deadlocked.

Theorem 27. *If* $\Gamma; \Delta \vdash \mathcal{C}$, *then* $\neg \mathit{deadlocked}(\mathcal{C})$.

Proof. By contradiction. By the definition of deadlocked, we know that there must be some cyclic dependency, which would be ill-typed due to Lemma 54. □

8.3.4 Global Progress

Deadlock-freedom is fairly close to the notion of deadlock-freedom for SGV. We now turn our attention to global progress. Recall that $\mathcal{A}$ ranges over *auxiliary threads*; that is, threads which are either a child thread or a buffer. Let $\mathcal{M}$ denote configurations of the following form:

$$\mathcal{M} ::= \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n \parallel \bullet N$$

We may now define the notion of a canonical form for AGV.

Definition 12 (Canonical Form)**.** *A configuration $\mathcal{C}$ is in* canonical form *if there is a sequence of names $a_1, \ldots, a_n$, a sequence of configurations $\mathcal{A}_1, \ldots, \mathcal{A}_n$, and a configuration $\mathcal{M}$, such that:*

$$\mathcal{C} = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}) \ldots))$$

where $a_i \in \mathsf{fn}(\mathcal{A}_i)$ for each a_i and $\mathcal{A}_i$.

All AGV programs with a main thread may be written in canonical form.

Theorem 28 (Canonical Forms (AGV))**.** *If $\Gamma; \Delta \vdash^{\bullet} \mathcal{C}$, then there exists some $\mathcal{C}' \equiv \mathcal{C}$ such that $\Gamma; \Delta \vdash^{\bullet} \mathcal{C}'$ and $\mathcal{C}'$ is in canonical form.*

Proof. Similar to the analogous proof for SGV. □

With the notion of a canonical form defined, and knowing that we may write AGV configurations with a main thread in canonical form, we are well-placed to state some progress results.

Again, let us adopt the convention that Ψ ranges over term typing environments only containing runtime names. Unsurprisingly, the functional fragment of AGV enjoys progress.

Lemma 55. *If $\Psi \vdash M : A$, then either:*

1. *M is a value*
2. *there exists some N such that $M \longrightarrow_M N$*
3. *there exist E, N such that M can be written $E[N]$, where N is a session typing primitive of the form* **fork** *V,* **send** *$V\,W$,* **receive** *V, or* **wait** *V.*

Proof. A standard induction on the derivation of $\Psi \vdash M : A$. □

To reason about progress of configurations, we define the notion of a *ready thread*: that is, a thread which is ready to perform an action on a given channel. We refer to ready threads as opposed to blocked threads since communication is asynchronous and thus sends are non-blocking.

Definition 13 (Ready thread)**.** *We say that a thread M is* ready to perform an action on endpoint a*, written $\mathit{ready}(M,a)$, if M is about to send on, receive on, or wait on a. Formally:*

$$\mathit{ready}(a,M) \triangleq \exists E.(M = E[\mathbf{send}\, V\, a]) \vee (M = E[\mathbf{receive}\, a]) \vee (M = E[\mathbf{wait}\, a])$$

Each auxiliary thread of an open, non-reducing AGV configuration is either a buffer, or a thread ready to perform an action on either a variable in the typing environment or a preceding ν-bound variable. We firstly describe the notion of *open progress*. Intuitively, if a configuration satisfies open progress, it does not "go wrong", but may be ready to perform an action on an external name. We define open progress inductively on canonical forms.

Definition 14 (Open Progress)**.** *Suppose $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$, where $\mathcal{C}$ is in canonical form and $\mathcal{C} \not\Longrightarrow$. We say that $\mathcal{C}$* satisfies open progress *if:*

1. *$\mathcal{C} = (\nu a)(\mathcal{A} \parallel \mathcal{D})$, where $\Psi = \Psi_1, \Psi_2$ and $\Delta = \Delta_1, \Delta_2$ such that either:*
   (a) *$\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{D}$ where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either:*
      i. *A thread $\circ M$ where there exists some $b \in \mathsf{fn}(\Psi_1, a : S)$ such that either $M = b$ or $\mathit{ready}(b, M)$; or*
      ii. *A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \overrightarrow{V}$ or $a \in \overrightarrow{W}$*
   (b) *$\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{D}$, where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$*
2. *$\mathcal{C} = \mathcal{A} \parallel \mathcal{M}$, where $\Psi = \Psi_1, \Psi_2$ and either:*
   (a) *$\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either:*
      i. *A thread $\circ M$ where there exists some $b \in \mathsf{fn}(\Psi_1, a : S)$ such that ether $M = b$ or $\mathit{ready}(b, M)$; or*
      ii. *A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \mathsf{fn}(\overrightarrow{V})$ or $a \in \mathsf{fn}(\overrightarrow{W})$*
   (b) *$\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$*
3. *$\mathcal{C} = \bullet M$, where either $M = V$ for some value V, or $\mathit{ready}(M, a)$ for some $a \in \mathsf{fn}(\Psi)$.*

Every well-typed non-reducing configuration that is in canonical form satisfies open progress; thanks to the compositional definition, the proof is a routine induction on typing derivations.

Lemma 56. *Suppose $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$, where $\mathcal{C}$ is in canonical form and $\mathcal{C} \not\Longrightarrow$. Then $\mathcal{C}$ satisfies open progress.*

Proof. By induction on the derivation of $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$; see Appendix C for details. □

As an immediate corollary, we can show a more global and concise property.

Corollary 7 (Open Progress (AGV))**.** *Suppose $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$ where $\mathcal{C}$ is in canonical form, and $\mathcal{C} \not\Longrightarrow$.*

Let $\mathcal{C} = (\nu a_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M})\cdots)$. Then:

1. *For $1 \leq i \leq n$, each $\mathcal{A}_i$ is either:*
   (a) *a child thread $\circ M$ such that either $M = a_i$, or that there exists $b \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{fn}(\Psi)$ such that ready(b, M)*
   (b) *a buffer*
2. *$\mathcal{M} = \mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_m \parallel \bullet N$ such that*
   (a) *Each $\mathcal{A}'_i$ is either:*
      i. *a child thread $\circ M$ for which there exists some $b \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$ such that either $M = b$ or ready(b, M); or*
      ii. *a buffer*
   (b) *N is either a value, or ready(b, N) for some $b \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$*

We can substantially tighten the result if we only consider closed configurations.

Corollary 8 (Closed Progress (AGV))**.** *Suppose $\cdot;\cdot \vdash^{\bullet} \mathcal{C}$ where $\mathcal{C}$ is in canonical form, and $\mathcal{C} \not\Longrightarrow$.*

Let $\mathcal{C} = (\nu a_1)(\mathcal{A}_1 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M})\cdots)$. Then:

1. *For $1 \leq i \leq n$, each $\mathcal{A}_i$ is either:*
   (a) *A child thread $\circ M$ such that either $M = a_i$, or ready(a_i, M)*
   (b) *A buffer*
2. *$\mathcal{M} = \bullet V$ for some V*

The result follows since each child thread $\circ M_i$ may no longer be ready to perform a communication action on an external endpoint, thus each child thread must be a value (more specifically, a name a_i of type $\mathsf{End}_!$), or ready to perform an action on a_i. Due to the emptiness of typing environments, it cannot be the case that a parallel composition arises in $\mathcal{M}$, since the initial runtime typing environment does not contain any entries of the form $a : S^\sharp$.

Even when considering closed configurations, communication may be blocked if the main thread contains an endpoint name. A conservative way to ensure that communication in a

configuration cannot be blocked is to mandate that the type of its main thread contains no session endpoints nor function types (as functions may capture session endpoints). Thus, we arrive at a yet stronger result for ground configurations: if a configuration is a non-reducing ground configuration, then it must be a main thread fully reduced to a value.

Corollary 9 (Global Progress (AGV)). *If $\mathcal{C}$ is a ground configuration such that $\mathcal{C} \not\Longrightarrow$, then $\mathcal{C} = \bullet V$ for some V.*

8.3.5 Confluence

Like SGV, AGV satisfies the diamond property. Asynchrony requires us to take slightly more care, as it introduces a critical pair when reducing a concurrent read and write on a channel.

Theorem 29 (Diamond Property). *If $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$, and $\mathcal{C} \Longrightarrow \mathcal{D}_1$, and $\mathcal{C} \Longrightarrow \mathcal{D}_2$, then either $\mathcal{D}_1 \equiv \mathcal{D}_2$, or there exists some $\mathcal{D}_3$ such that $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$ and $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$.*

Proof. As in SGV, $\longrightarrow_{\mathsf{M}}$ is deterministic and hence confluent due to the call-by-value, left-to-right ordering imposed by evaluation contexts, and due to linearity, communication actions on different channels may be performed in any order.

Unlike SGV, due to asynchrony, we have a single critical pair when considering concurrent reads and writes to a buffer. Nevertheless, a read and a write may happen in any order, and the pair converges in a single step.

$$\begin{array}{ccc} & \mathcal{F}[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) \parallel \mathcal{F}'[\mathbf{receive}\,b] & \\ \swarrow & & \searrow \\ \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) \parallel \mathcal{F}'[\mathbf{receive}\,b] & & \mathcal{F}[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \parallel \mathcal{F}'[(V,b)] \\ \searrow & & \swarrow \\ & \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \parallel \mathcal{F}'[(V,b)] & \end{array}$$

□

8.3.6 Termination

Again, similarly to SGV, AGV is terminating due by an elementary argument based on linearity. Our asynchronous semantics does not give rise to nontermination.

Theorem 30. *If $\Gamma \vdash^{\phi} \mathcal{C}$, then there are no infinite $\Longrightarrow$ reductions from $\mathcal{C}$.*

Proof. Similar to the proof of termination for SGV. Define the measure of a configuration to be the size of the sums of the ASTs of all threads and values contained in buffers. The measure strictly reduces under reduction and remains invariant under $\equiv$, so no infinite reduction sequences may exist. □

8.4 Related Work

Untyped Asynchronous Process Calculi. Boudol [24] describes the Asynchronous π-calculus, a variation of the π-calculus based on the chemical metaphor introduced by the Chemical Abstract Machine [21]. In contrast to AGV, the Asynchronous π-calculus does not include any primitive notion of a queue. Instead, the key twist is to modify the output process $\bar{x}y.P$ (read as "send name y over name x and continue as P") to $\bar{x}y$, eliminating output as a synchronous prefixing operation. Asynchrony is implemented by creating a message process in parallel, instead of communicating via synchronous rendezvous; the message may then be retrieved from the 'chemical solution' by an input-prefixed process $x(y).P$. The core idea of introducing asynchrony by eliminating output prefixing is introduced independently by Honda and Tokoro [95]. Boudol shows that it is possible to encode synchronous output prefixing using the asynchronous π-calculus using an acknowledgement channel. The asynchronous π-calculus is, however, not powerful enough to encode the full synchronous π-calculus, as it cannot satisfactorily encode mixed choice [168].

Session-Typed Asynchronous Process Calculi. Coppo et al. [48] describe a session-typed object-oriented language AMoose with asynchronous communication. The asynchronous communication semantics is achieved by queues stored in a global heap. The authors also prove a progress property: their approach to proving progress involves introducing an effect system which annotates expressions with channels which may produce deadlock. AGV differs in that it is a functional calculus, queues are first-class processes, and progress is guaranteed by the acyclicity of configurations.

Kouzapas et al. [124] describe a session-typed process calculus with asynchronous communication semantics, and describe its behavioural theory by introducing an *asynchronous session bisimulation* relation and showing that it coincides with reduction-closed congruence [96]. To aid the study of the behavioural theory, asynchrony is formalised by each endpoint being associated with both an input *and* an output queue; communication happens directly between queues as a separate reduction step. Only input queues are present in AGV, and there is a single buffer process per pair of endpoints as opposed to a buffer process for each endpoint. We also consider progress, which is largely inherited from the logical correspondence between GV and CP, whereas Kouzapas et al. do not.

Concurrent Typestate-oriented Programming As discussed in Chapter 2, *typestate* governs which methods are available on an object, and may change as an object evaluates. Aldrich et al. [9] introduce the paradigm of *typestate-oriented programming* (TSOP), implemented in the Plaid programming language. In TSOP, stateful object fields have a *state* which governs which methods may be called. As an example, it is only possible to read from an open file. Methods

on the object may change the state as a result. Stateful objects must be treated linearly, which is handled by Plaid's permissions system.

Crafa and Padovani [50] investigate TSOP in the concurrent setting, by considering a behavioural typing discipline for the Objective Join Calculus [70] based on the chemical metaphor introduced by Berry and Boudol [21]. Object types describe messages that an object must handle and eventually send: an example is an object which must eventually release a lock after it has been acquired. As the Objective Join Calculus is based on the chemical metaphor, communication is asynchronous without the use of explicit buffer processes. Padovani [166] refines this system to ensure deadlock-freedom, which is achieved using an effect typing system which constructs a *dependency graph* forbidding cyclic dependencies.

Session-Typed Asynchronous Functional Languages. Gay and Vasconcelos [77] were first to describe the integration of session types and functional programming through the use of a linear type system. Their calculus, which has obtained the name LAST [135], is a linear λ-calculus which, like SGV and AGV, is defined in terms of a deterministic reduction relation on terms and a nondeterministic reduction relation on configurations. LAST has many additional features including subtyping, access points for more flexible session initiation, a fixpoint combinator for recursion, and additional information for proving that the maximum size of a buffer is bounded by its session type.

LAST satisfies preservation and a weak form of progress, but its access points admit nontermination and nondeterminism. The calculus does not preclude deadlock. In contrast, AGV builds upon SGV to investigate asynchrony while retaining SGV's strong metatheory. More specifically, we retain deadlock-freedom since the **fork** construct coupled with the configuration typing system ensure that configurations are acyclic. In the more modular approach taken by GV, we begin with a well-behaved calculus and can re-add features such as access points and fixpoint combinators, and know which properties remain. In particular, and as we will discuss further in Chapter 9, §9.4, adding a fixpoint combinator retains all properties except termination, whereas adding access points loses all guarantees except preservation and a weak form of progress.

Lindley and Morris [136] consider asynchronous communication semantics for an extension of GV, FST. In particular, we take inspiration from their formulation of buffer processes and buffer typing. Nevertheless, our approach has several fundamental differences: most importantly, our calculus is fully linear and separates $\mathsf{End}_!$ and $\mathsf{End}_?$, whereas Lindley and Morris [136] provide a single, self-dual End type, inhabitants of which may be discarded implicitly. Although treating channels of type End as affine increases the amount of concurrency (as synchronisation is not required to close a channel), it complicates the typing rules. Since the purpose of AGV is to provide a small, well-behaved core language, we also omit the more

advanced features of FST such as subkinding and row typing.

As a result, AGV makes simplifications to the metatheory, and we provide the first full proofs of correctness. As we saw for Synchronous GV, a useful technique to reason about the progress of a configuration is to define a *canonical form*. Lindley and Morris [136] describe a canonical form which relies on the affine nature of names, where buffers may be introduced through the reverse application of a garbage collection equivalence. In contrast, we have an explicitly linear calculus with a more conventional way of constructing canonical forms.

8.5 Conclusion

Asynchrony is vital when considering distributed applications. In this chapter, we have introduced *Asynchronous GV*, an extension of Synchronous GV to incorporate asynchronous, buffered communication.

We have discussed the metatheory of Asynchronous GV, arguing that it satisfies preservation, progress, confluence, and termination, and we have provided the first full proofs that the metatheory holds.

With an asynchronous semantics in hand, we may proceed to describing how to safely integrate session types and exceptions, getting us closer to our goal of extending the Links web programming language with distributed session-typed channels.

Chapter 9

Exceptional GV

9.1 Introduction

So far, we have seen two core functional languages: Synchronous GV and Asynchronous GV, which are logically-grounded and have strong correctness guarantees. Our goal is to bring session types to the domain of tierless web programming, which by nature is distributed, and asynchrony is more suited to implementation in the distributed setting.

In the case of concurrent, multi-threaded applications, it may be reasonable to assume that all interactions succeed. However, this is certainly not the case in the distributed setting where a participant in a session may go offline or crash, and the problem of disconnection is particularly pertinent in web applications where a client may simply close its browser window at any time. Alas, most accounts of session types do not handle failure, which means they are of limited use in distributed applications, where failure is pervasive. Inspired by the work of Mostrous and Vasconcelos [147], in this chapter we present the first account of asynchronous session types with failure handling in a functional programming language. Key to the calculus is the ability to safely discard a session, and its safe integration with exception handling.

9.1.1 Session Types

Recall the earlier example of two-factor authentication described in Chapter 2. A user inputs their credentials. If the login attempt is from a known device, then they are authenticated and may proceed to perform privileged actions. If the login attempt is from an unrecognised device, then the user is sent a challenge code. They enter the challenge code into a hardware key which yields a response code. If the user responds with the correct response code, then they are authenticated.

Figure 9.1 shows the session type for the server which receives a pair of a username and password from the client, then selects whether to authenticate the client, issue a challenge, or reject the credentials. If the server decides to issue a challenge, then it sends the challenge

$$
\begin{aligned}
&\mathsf{TwoFactorServer} \triangleq \\
&\quad ?(\mathsf{Username}, \mathsf{Password}).\oplus\{ \\
&\qquad \mathsf{Authenticated} : \mathsf{ServerBody}, \\
&\qquad \mathsf{Challenge} : !\mathsf{ChallengeKey}.?\mathsf{Response}. \\
&\qquad\quad \oplus\{\mathsf{Authenticated} : \mathsf{ServerBody}, \\
&\qquad\qquad \mathsf{AccessDenied} : \mathsf{End}\}, \\
&\qquad \mathsf{AccessDenied} : \mathsf{End}\}
\end{aligned}
$$

Figure 9.1: Server Session Type

string, awaits the response, and either authenticates or rejects the client. The ServerBody type abstracts over the remainder of the interactions, for example making a deposit or withdrawal.

Recall the implementation of the server, which takes an endpoint of type TwoFactorServer along which it receives the credentials, which are checked using checkDetails. If the check passes, then the server proceeds to the application body (serverBody(s)); if not, then the server notifies the client by selecting the AccessDenied branch.

$$
\begin{aligned}
&\mathsf{twoFactorServer} : \mathsf{TwoFactorServer} \multimap \mathbf{1} \\
&\mathsf{twoFactorServer}(s) \triangleq \mathbf{let}\,((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\,s\,\mathbf{in} \\
&\qquad\qquad \mathbf{if}\,\mathsf{checkDetails}(\mathit{username}, \mathit{password})\,\mathbf{then} \\
&\qquad\qquad\quad \mathbf{let}\,s = \mathbf{select}\,\mathsf{Authenticated}\,s\,\mathbf{in}\,\mathsf{serverBody}(s) \\
&\qquad\qquad \mathbf{else} \\
&\qquad\qquad\quad \mathbf{let}\,s = \mathbf{select}\,\mathsf{AccessDenied}\,s\,\mathbf{in}\,\mathbf{close}\,s
\end{aligned}
$$

9.1.2 Substructural Types

In Chapter 2 we asserted that in order to safely implement session-typed communication, we require a *linear* type system, meaning that each variable is used precisely once. Linearity ensures both that session endpoints are used at most once, preventing duplicate messages from being sent and thus violating the protocol, and also at least once, ensuring that a developer does not accidentally forget to finish the implementation of a protocol.

However, linearity can sometimes be too strong. Certainly, statically checking session types demands some form of substructural type system. In this section, we discuss three options: linear types, affine types, and linear types with explicit cancellation.

9.1.2.1 Linear Types.

Simply providing constructs for sending and receiving values, and for selecting and offering choices, is insufficient for safely implementing session types. Consider the following client:

$$\begin{aligned}&\mathsf{wrongClient} : \mathsf{TwoFactorClient} \multimap \mathbf{1}\\&\mathsf{wrongClient}(s) \triangleq \mathbf{let}\, t = \mathbf{send}\,(\texttt{"Alice"}, \texttt{"hunter2"})\, s\, \mathbf{in}\\&\qquad\qquad \mathbf{let}\, t = \mathbf{send}\,(\texttt{"Bob"}, \texttt{"letmein"})\, s\, \mathbf{in}\ \ldots\end{aligned}$$

Reuse of s allows a (username, password) pair to be sent along the same endpoint twice, violating session fidelity.

Exceptions. In practice, linear session types are unrealistic. Thus far, we have assumed checkDetails always succeeds, which may be plausible if checking against an in-memory store, but not if connecting to a remote database. One option would be for checkDetails to return false on failure, but that would lose information. Instead, suppose we have an exception handling construct. As a first attempt, we might try to write:

$$\begin{aligned}&\mathsf{exnServer1} : \mathsf{TwoFactorClient} \multimap \mathbf{1}\\&\mathsf{exnServer1}(s) \triangleq \mathbf{let}\,((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\, s\, \mathbf{in}\\&\qquad\qquad \mathbf{try}\, \mathbf{if}\, \mathsf{checkDetails}(\mathit{username}, \mathit{password})\, \mathbf{then}\\&\qquad\qquad\qquad \mathbf{let}\, s = \mathbf{select}\, \mathsf{Authenticated}\, s\, \mathbf{in}\, \mathsf{serverBody}(s)\\&\qquad\qquad\quad \mathbf{else}\\&\qquad\qquad\qquad \mathbf{let}\, s = \mathbf{select}\, \mathsf{AccessDenied}\, s\, \mathbf{in}\, \mathbf{close}\, s\\&\qquad\qquad \mathbf{catch}\, \mathsf{log}(\texttt{"Database Error"})\end{aligned}$$

However, the above code does not type-check and is unsafe. Linear endpoint s is not used in the **catch** block and yet is still open if an exception is raised by checkDetails.

As a second attempt, we may decide to localise exception handling to the call to checkDetails. We introduce checkDetailsOpt, which returns Some(result) if the call is successful and None if not.

$$\begin{aligned}&\mathsf{checkDetailsOpt} : (\mathsf{Username} \times \mathsf{Password}) \multimap \mathsf{Option}(\mathsf{Bool})\\&\mathsf{checkDetailsOpt}(\mathit{username}, \mathit{password}) \triangleq \mathbf{try}\, \mathsf{Some}(\mathsf{checkDetails}(\mathit{username}, \mathit{password}))\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{catch}\, \mathsf{None}\end{aligned}$$

$$\begin{aligned}&\mathsf{exnServer2} : \mathsf{TwoFactorServer} \multimap \mathbf{1}\\&\mathsf{exnServer2}(s) \triangleq \mathbf{let}\,((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\, s\, \mathbf{in}\\&\qquad\qquad \mathbf{case}\, \mathsf{checkDetailsOpt}(\mathit{username}, \mathit{password})\, \mathbf{of}\\&\qquad\qquad\quad \mathsf{Some}(\mathit{res}) \mapsto \mathbf{if}\, \mathit{res}\, \mathbf{then}\, \mathbf{let}\, s = \mathbf{select}\, \mathsf{Authenticated}\, s\, \mathbf{in}\, \mathsf{serverBody}(s)\\&\qquad\qquad\qquad\qquad\quad \mathbf{else}\, \mathbf{let}\, s = \mathbf{select}\, \mathsf{AccessDenied}\, s\, \mathbf{in}\, \mathbf{close}\, s\\&\qquad\qquad\quad \mathsf{None} \mapsto \mathsf{log}(\texttt{"Database Error"})\end{aligned}$$

Still the code is unsafe as it does not use s in the None branch of the case-split. However, we do now have more precise information about the type of s, since it is unused in the **try** block in checkDetailsOpt. One solution could be to adapt the protocol by adding an InternalError branch:

$$\begin{aligned}&\mathsf{TwoFactorServerExn} \triangleq ?(\mathsf{Username}, \mathsf{Password}).\oplus\{\\&\quad \mathsf{Authenticated} : \mathsf{ServerBody},\\&\quad \mathsf{Challenge} : !\mathsf{ChallengeKey}.\mathsf{Response}.\oplus\{\mathsf{Authenticated} : \mathsf{ServerBody}, \mathsf{AccessDenied} : \mathsf{End}\},\\&\quad \mathsf{AccessDenied} : \mathsf{End},\\&\quad \mathsf{InternalError} : \mathsf{End}\}\end{aligned}$$

We could use **select** InternalError s in the None branch to yield a type-correct program, but doing so would be unsatisfactory as it clutters the protocol and the implementation with failure points.

Disconnection. The problem of failure is compounded by the possibility of disconnection. On a single machine it may be plausible to assume that communication always succeeds. In a distributed setting this assumption is unrealistic as parties may disconnect without warning. The problem is particularly acute in web applications as a client may close the browser at any point. In order to adequately handle failure we must incorporate some mechanism for detecting disconnection.

9.1.2.2 Affine Types

We began by assuming linear types—each endpoint must be used *exactly* once. One might consider relaxing linear types to *affine types*—meaning that each endpoint must be used *at most* once. Statically checked affine types form the basis of the existing Rust implementation of session types [113] and dynamically checked affine types form the basis of the OCaml FuSe [164] and Scala `lchannels` [190] session type libraries. Affine types present two quandaries, both arising from endpoints being silently discarded. First, a developer receives no feedback if they *accidentally* forget to finish a protocol implementation. Second, if an exception is raised in an evaluation context that captures an open endpoint then the peer may be left waiting forever.

9.1.2.3 Linear Types with Explicit Cancellation

Mostrous and Vasconcelos [147] address the difficulties outlined above through an *explicit* discard (or *cancellation*) operator. (They characterise their sessions as *affine*, but it is important not to confuse their system with affine type systems, as in §9.1.2.2, which allow variables to be discarded *implicitly*.) Their approach boils down to three key principles: endpoints can be explicitly discarded; an exception is thrown if a communication cannot succeed because a

peer endpoint has been cancelled; and endpoint cancellations are propagated when endpoints become inaccessible due to an exception being thrown. They introduce a process calculus including the term $a\lightning$ ("cancel a"), which indicates that endpoint a may no longer be used to perform communications. They provide an exception handling construct which attempts a communication action, running an exception handler if the communication action fails, and show that explicit cancellation is well-behaved: their calculus satisfies preservation and global progress (well-typed processes never get stuck), and is confluent.

Explicit cancellation neatly handles failure while ruling out accidentally incomplete implementations and providing a mechanism for notifying peers when an exception is raised. In this chapter we take advantage of explicit cancellation to formalise and implement asynchronous session types with failure handling in a distributed functional programming language; this is not merely a routine adaptation of the ideas of Mostrous and Vasconcelos for the following reasons:

- They present a *process calculus*, but we work in a *functional programming language*.
- Communication in their system is *synchronous*, depending on a rendezvous between sender and receiver. We require *asynchronous* communication, which is more amenable to implementation in a distributed setting.
- Their exception handling construct is over a single communication action and does not allow nested exception handling. This design is difficult to reconcile with a functional language, as it is inherently *non-compositional*. Our exception handling construct is *compositional*.

In this chapter, we define a core concurrent λ-calculus, *Exceptional GV* (EGV), with asynchronous session-typed communication and exception handling. As with the calculus of Mostrous and Vasconcelos, an exception is raised when a communication action fails. But our compositional exception handling construct can be arbitrarily nested, and allows exception handling over multiple communication actions.

Using EGV, we may implement the two factor authentication server as follows:

```
exnServer3 : TwoFactorServer ⊸ 1
exnServer3(s) ≜ let ((username, password), s) = receive s in
                try checkDetails(username, password) as res in
                  if res then let s = select Authenticated s in serverBody(s)
                  else let s = select AccessDenied s in close s
                otherwise
                  cancel (s); log("Database Error")
```

Following Benton and Kennedy [17], an exception handler **try** L **as** x **in** M **otherwise** N takes an explicit success continuation M as well as the usual failure continuation N. If checkDetails fails with an exception, then s is safely discarded using **cancel**, which takes an endpoint and

returns the unit value. Disconnection is handled by cancelling all endpoints associated with a client. If a peer tries to read along a cancelled endpoint then an exception is thrown.

This chapter concentrates on EGV and its metatheory. It also serves as the basis for our implementation of distributed sessions for web applications, which we describe in more detail in Chapter 10. The remainder of this chapter is structured as follows. §9.2 presents EGV and discusses how it builds on AGV, and §9.3 discusses its metatheory. Finally, §9.4 discusses extensions to Exceptional GV, including exception payloads, unrestricted types, access points, and recursive sessions.

9.2 Exceptional GV

In this section, we introduce Exceptional GV (henceforth EGV). EGV extends AGV with support for failure handling.

Due to GV's close correspondence with classical linear logic, EGV has a strong metatheory, enjoying preservation, global progress, the diamond property, and termination. Much like the simply-typed λ-calculus, this well-behaved core must be extended to be expressive enough to write larger applications. Nonetheless, the core calculus alone is expressive enough to support our two-factor authentication example, and to support server applications which gracefully handle disconnection. In §9.3, we show that cancellation is well-behaved, and does not violate any of the core properties of GV. In §9.4, following Lindley and Morris [132, 135], we extend EGV modularly with standard features of our implementation, some of which provide weaker guarantees. Channel cancellation and exceptions are orthogonal to these features.

9.2.1 Integrating Sessions with Exceptions, by Example

Integrating session types with failure handling into a higher-order functional language requires care. We illustrate three important cases: cancellation and exceptions, delegation, and closures.

Cancellation and Exceptions. Consider the cancelExn function, which forks a thread which immediately cancels its endpoint.

$$
\begin{array}{l}
\mathsf{cancelExn} \triangleq \\
\quad \textbf{try} \\
\quad\quad \textbf{let}\, s = \textbf{fork}\,(\lambda t.\textbf{cancel}\, t)\, \textbf{in} \\
\quad\quad \textbf{let}\,(\mathit{res}, s) = \textbf{receive}\, s\, \textbf{in} \\
\quad\quad \textbf{close}\, s; \mathit{res} \\
\quad \textbf{as}\, \mathit{res}\, \textbf{in} \\
\quad\quad \mathsf{print}\,(\texttt{"Result: "} + \mathit{res}) \\
\quad \textbf{otherwise}\, \mathsf{print}\, \texttt{"Error!"}
\end{array}
$$

The parent attempts to receive, but the message can never arrive so an exception is raised and the **otherwise** clause is invoked.

Note that we could equally move the **fork** out of the **try** block with no difference to the possible reductions, since **fork** can never raise an exception.

Delegation. A central feature of π-calculus is *mobility* of names. In session calculi, sending an endpoint is known as *session delegation*.

$$
\begin{array}{l}
\mathsf{delegation} \triangleq \\
\quad \textbf{let}\, s = \\
\quad\quad \textbf{fork}\, (\lambda t. \\
\quad\quad\quad \textbf{let}\, (res, t) = \textbf{receive}\, t\, \textbf{in} \\
\quad\quad\quad \textbf{close}\, t; res)\, \textbf{in} \\
\quad \textbf{let}\, u = \textbf{fork}\, (\lambda v. \textbf{cancel}\, v)\, \textbf{in} \\
\quad \textbf{let}\, u = \textbf{send}\, s\, u\, \textbf{in} \\
\quad \textbf{close}\, u
\end{array}
$$

The delegation function begins by forking a thread and returning endpoint s. The child is passed endpoint t on which it blocks receiving. Next, the parent forks a second child, yielding endpoint u. The second child is passed endpoint v, which is immediately discarded using **cancel**. Now the parent thread sends endpoint s along u. Endpoint s will never be received as the peer endpoint v of u has been cancelled. In turn, this renders s irretrievable and an exception is thrown in the first child thread, as it can never receive a value.

Closures. It is crucial that cancellation plays nicely with closures.

$$
\begin{array}{l}
\mathsf{closure}(s) \triangleq \\
\quad \textbf{let}\, f = (\lambda x. \textbf{send}\, x\, s)\, \textbf{in} \\
\quad \textbf{raise}; \\
\quad f(5)
\end{array}
$$

The closure function defines a function f which sends its argument x along s. The parent thread then raises an exception. As s appears in the closure bound to f, which appears in the continuation and is thus discarded, s must be cancelled.

9.2.2 Syntax and Typing Rules for Terms

Fig. 9.2 gives the syntax of EGV. Again, constructs which differ to those in AGV are shaded. In contrast to SGV and AGV, we dispense with $\mathsf{End}_!$ and $\mathsf{End}_?$ and instead introduce a self-dual End type; we discuss the reasoning for this in §9.3.1.

Types	$A,B,C ::= \mathbf{1} \mid A \multimap B \mid A + B \mid A \times B \mid S$
Session Types	$S,T ::= !A.S \mid ?A.S \mid \mathsf{End}$
Variables	x,y
Terms	$L,M,N ::= x \mid \lambda x.M \mid M\,N$
	$\mid\ () \mid \mathbf{let}\,() = M\,\mathbf{in}\,N$
	$\mid\ (M,N) \mid \mathbf{let}\,(x,y) = M\,\mathbf{in}\,N$
	$\mid\ \mathbf{inl}\,M \mid \mathbf{inr}\,M \mid \mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\}$
	$\mid\ \mathbf{fork}\,M \mid \mathbf{send}\,M\,N \mid \mathbf{receive}\,M \mid \mathbf{close}\,M$
	$\mid\ \mathbf{cancel}\,M \mid \mathbf{raise} \mid \mathbf{try}\,L\,\mathbf{as}\,x\,\mathbf{in}\,M\,\mathbf{otherwise}\,N$
Type Environments	$\Gamma ::= \cdot \mid \Gamma, x : A$

Figure 9.2: Syntax

We introduce three new term constructs to support session typing with failure handling: **cancel** M explicitly discards session endpoint M; **raise** raises an exception; and **try** L **as** x **in** M **otherwise** N evaluates L, on success binding the result to x in M and on failure evaluating N.

As in SGV and AGV, we omit **select** and **offer** from the core calculus (following Lindley and Morris [132, 135]) as they can be encoded using sums and delegation [54, 118].

Typing. Fig. 9.3 gives the typing rules for EGV. As usual, linearity is enforced by splitting environments when typing subterms, ensuring T-Var takes a singleton environment, and leaf rules T-Unit and T-Raise take an empty environment.

As exceptions do not return values, the rule T-Raise allows an exception to be given any type A. Rule T-Try embraces explicit success continuations as advocated by Benton and Kennedy [17], binding a result in M if L evaluates successfully. The T-Cancel rule explicitly discards an endpoint. Naïvely implemented, cancellation violates progress: a thread could discard an endpoint, and if its peer is blocked receiving, the peer will be left waiting forever. We avoid this pitfall by taking care to cancel discarded endpoints.

9.2.3 Operational Semantics

We now give a small-step operational semantics for EGV.

Runtime Syntax. Fig. 9.4 shows the runtime syntax of EGV, which builds upon that of AGV. As well as program threads (i.e., $\circ M$ and $\bullet M$), configurations include three special forms of thread. A *buffer thread* $(a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}))$, as in AGV models asynchrony. A *zapper thread* $(\lightning\, a)$

Term Typing $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Var}\ \frac{}{x:A \vdash x:A} \qquad \textsc{T-Abs}\ \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M : A \multimap B} \qquad \textsc{T-App}\ \frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M\,N : B}$$

$$\textsc{T-Unit}\ \frac{}{\cdot \vdash () : \mathbf{1}} \qquad \textsc{T-LetUnit}\ \frac{\Gamma_1 \vdash M : \mathbf{1} \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\,() = M\,\mathbf{in}\,N : A} \qquad \textsc{T-Pair}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash (M,N) : A \times B}$$

$$\textsc{T-LetPair}\ \frac{\Gamma_1 \vdash M : A \times B \quad \Gamma_2, x:A, y:B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathbf{let}\,(x,y) = M\,\mathbf{in}\,N : C}$$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}\,M : A + B} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}\,M : A + B}$$

$$\textsc{T-Case}\ \frac{\Gamma_1 \vdash L : A + B \quad \Gamma_2, x:A \vdash M : C \quad \Gamma_2, y:B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} : C}$$

$$\textsc{T-Fork}\ \frac{\Gamma \vdash M : S \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}\,M : \overline{S}} \qquad \textsc{T-Send}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : !A.S}{\Gamma_1, \Gamma_2 \vdash \mathbf{send}\,M\,N : S} \qquad \textsc{T-Recv}\ \frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{receive}\,M : (A \times S)} \qquad \textsc{T-Close}\ \frac{\Gamma \vdash M : \mathsf{End}}{\Gamma \vdash \mathbf{close}\,M : \mathbf{1}}$$

$$\textsc{T-Cancel}\ \frac{\Gamma \vdash M : S}{\Gamma \vdash \mathbf{cancel}\,M : \mathbf{1}} \qquad \textsc{T-Try}\ \frac{\Gamma_1 \vdash L : A \quad \Gamma_2, x:A \vdash M : B \quad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash \mathbf{try}\,L\,\mathbf{as}\,x\,\mathbf{in}\,M\,\mathbf{otherwise}\,N : B} \qquad \textsc{T-Raise}\ \frac{}{\cdot \vdash \mathbf{raise} : A}$$

Duality $\boxed{\overline{S}}$

$$\overline{!A.S} = ?A.\overline{S} \qquad \overline{?A.S} = !A.\overline{S} \qquad \overline{\mathsf{End}} = \mathsf{End}$$

Figure 9.3: Term Typing and Duality

represents an endpoint a that has been cancelled, and is used to propagate failure. A *halted thread* (**halt**) arises when the main thread has crashed due to an uncaught exception.

We sometimes find it useful to distinguish top-level threads $\mathcal{T}$ (main threads and halted threads) from auxiliary threads $\mathcal{A}$ (child threads, zapper threads, and buffer threads).

Environments. We extend type environments Γ to include runtime names of session type S and introduce runtime type environments Δ, which type both buffer endpoints of session type S and channels of type $S^\sharp$ for some S, but not object variables.

Contexts. The definition of evaluation contexts is another departure from AGV. In particular, we have two types of evaluation contexts: standard evaluation contexts E, and pure contexts P. Pure contexts P are those evaluation contexts that include no exception handling frames. Thread contexts $\mathcal{F}$ support reduction in program threads. Configuration contexts $\mathcal{G}$ support reduction under ν-binders and parallel composition.

Runtime Types	R	$::= S \mid S^\sharp$
Names	a, b, c	
Terms	M	$::= \cdots \mid a$
Values	U, V, W	$::= a \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\, V \mid \mathbf{inr}\, V$
Configurations	$\mathcal{C}, \mathcal{D}, \mathcal{E}$	$::= (\nu a)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M \mid \mathbf{halt} \mid \lightning a \mid a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$
Thread Flags	ϕ	$::= \bullet \mid \circ$
Top-level threads	$\mathcal{T}$	$::= \bullet M \mid \mathbf{halt}$
Auxiliary threads	$\mathcal{A}$	$::= \circ M \mid \lightning a \mid a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$
Type Environments	Γ	$::= \cdots \mid \Gamma, a : S$
Runtime Type Environments	Δ	$::= \cdot \mid \Delta, a : R$
Evaluation Contexts	E	$::= [\,] \mid E\, M \mid V\, E$
		$\mid \mathbf{let}\,() = E\,\mathbf{in}\,M \mid (E, M) \mid (V, E) \mid \mathbf{let}\,(x, y) = E\,\mathbf{in}\,M$
		$\mid \mathbf{inl}\, E \mid \mathbf{inr}\, E \mid \mathbf{case}\, E\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}$
		$\mid \mathbf{fork}\, E \mid \mathbf{send}\, E\, M \mid \mathbf{send}\, V\, E \mid \mathbf{receive}\, E \mid \mathbf{close}\, E$
		$\mid \mathbf{cancel}\, E \mid \mathbf{try}\, E\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N$
Pure Contexts	P	$::= [\,] \mid P\, M \mid V\, P$
		$\mid \mathbf{let}\,() = P\,\mathbf{in}\,M \mid \mathbf{let}\,(x, y) = P\,\mathbf{in}\,M \mid (P, M) \mid (V, P)$
		$\mid \mathbf{inl}\, P \mid \mathbf{inr}\, P \mid \mathbf{case}\, P\, \mathbf{of}\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}$
		$\mid \mathbf{fork}\, P \mid \mathbf{send}\, P\, M \mid \mathbf{send}\, V\, P \mid \mathbf{receive}\, P \mid \mathbf{close}\, P$
		$\mid \mathbf{cancel}\, P$
Thread Contexts	$\mathcal{F}$	$::= \phi E$
Configuration Contexts	$\mathcal{G}$	$::= [\,] \mid (\nu a)\mathcal{G} \mid \mathcal{G} \parallel \mathcal{C}$

Syntactic Sugar

$$\mathcal{C} \parallel \lightning V \triangleq \mathcal{C} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n \text{ where } \mathsf{fn}(V) = \{a_i\}_i$$
$$\mathcal{C} \parallel \lightning P \triangleq \mathcal{C} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n \text{ where } \mathsf{fn}(P) = \{a_i\}_i$$
$$\mathcal{C} \parallel \lightning E \triangleq \mathcal{C} \parallel \lightning a_1 \parallel \cdots \parallel \lightning a_n \text{ where } \mathsf{fn}(E) = \{a_i\}_i$$

Figure 9.4: Runtime Syntax

Syntactic Sugar. We write $\lightning V$, $\lightning P$, and $\lightning E$, as shorthand for the parallel composition of zapper threads for each free name in values V, pure contexts P, and evaluation contexts E, respectively. Given $\mathcal{C} \parallel \lightning V$ (and $\lightning P, \lightning E$ respectively), if $\mathsf{fn}(V) = \emptyset$, then $\mathcal{C} \parallel \lightning V = \mathcal{C}$.

Fig. 9.5 presents reduction and equivalence rules for terms and configurations.

Term Reduction. Reduction on terms is standard call-by-value β-reduction.

Configuration Equivalence. Like AGV, we have the standard π-calculus equivalence rules and an additional axiom to allow buffers to be treated symmetrically. EGV adds a further two axioms which function as garbage collection rules, allowing completed child threads and cancelled empty buffers to be discarded.

Communication and Concurrency. The basic communication and concurrency behaviour is identical to that of AGV: the E-Fork rule creates two fresh names for each endpoint of a channel, returning one name and substituting the other in the body of the spawned thread, as well as creating a channel with two empty buffers. The E-Send and E-Receive rules send to and receive from a buffer.

The E-Close rule discards an empty buffer once a session is complete. It differs from E-Wait in SGV and AGV since two threads synchronise and then proceed independently, whereas E-Wait can only synchronise with a process which has finished evaluating, and garbage collects the process as a result.

Cancellation. The E-Cancel rule cancels an endpoint by creating a zapper thread. The E-Zap rule ensures that when an endpoint is cancelled, all other endpoints in the buffer of the cancelled endpoint are also cancelled: it dequeues a value from the head of the buffer and cancels any endpoints contained within the dequeued value. It is applied repeatedly until the buffer is empty.

Raising Exceptions. Following Mostrous and Vasconcelos [147], an exception is raised when it would be otherwise impossible for a communication action to succeed. The E-ReceiveZap rule raises an exception if an attempt is made to receive along an endpoint whose buffer is empty and whose peer endpoint has been cancelled. Similarly, E-CloseZap raises an exception if an attempt is made to close a channel where the peer endpoint has been cancelled. There is no rule for the case where a thread tries to send a value along a cancelled endpoint; the free names in the communicated value must eventually be cancelled, but this is achieved through E-Zap. We choose not to raise an exception in this case since to do so would violate confluence, which we discuss in more detail in §9.3.4. Not raising exceptions on message sends to dead peers is standard behaviour for languages such as Erlang.

Term Reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$

$$
\begin{array}{lr}
\textsc{E-Lam} & (\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\textsc{E-Unit} & \mathbf{let}\,() = ()\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M \\
\textsc{E-Pair} & \mathbf{let}\,(x,y) = (V,W)\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M\{V/x,W/y\} \\
\textsc{E-Inl} & \mathbf{case}\,\mathbf{inl}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\textsc{E-Inr} & \mathbf{case}\,\mathbf{inr}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} N\{V/x\} \\
\textsc{E-Val} & \mathbf{try}\,V\,\mathbf{as}\,x\,\mathbf{in}\,M\,\mathbf{otherwise}\,N \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\textsc{E-Lift} & E[M] \longrightarrow_{\mathsf{M}} E[M'], \quad \text{if } M \longrightarrow_{\mathsf{M}} M'
\end{array}
$$

Configuration Equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad (\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

$$\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}), \quad \text{if } a \notin \mathsf{fn}(\mathcal{C})$$

$$a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \equiv b(\overrightarrow{W}) \leftrightsquigarrow a(\overrightarrow{V}) \qquad \circ() \parallel \mathcal{C} \equiv \mathcal{C} \qquad (\nu a)(\nu b)(\lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)) \parallel \mathcal{C} \equiv \mathcal{C}$$

Configuration Reduction $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

$$
\begin{array}{lr}
\textsc{E-Fork} & \mathcal{F}[\mathbf{fork}\,(\lambda x.M)] \longrightarrow (\nu a)(\nu b)(\mathcal{F}[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)), \quad \text{where } a,b \text{ are fresh} \\
\textsc{E-Send} & \mathcal{F}[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \\
\textsc{E-Receive} & \mathcal{F}[\mathbf{receive}\,a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[(U,a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\
\textsc{E-Close} & (\nu a)(\nu b)(\mathcal{F}[\mathbf{close}\,a] \parallel \mathcal{F}'[\mathbf{close}\,b] \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)) \longrightarrow \mathcal{F}[()] \parallel \mathcal{F}'[()] \\
\textsc{E-Cancel} & \mathcal{F}[\mathbf{cancel}\,a] \longrightarrow \mathcal{F}[()] \parallel \lightning a \\
\textsc{E-Zap} & \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \lightning a \parallel \lightning U \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\
\textsc{E-CloseZap} & \mathcal{F}[\mathbf{close}\,a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon) \longrightarrow \mathcal{F}[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon) \\
\textsc{E-ReceiveZap} & \mathcal{F}[\mathbf{receive}\,a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W}) \\
\textsc{E-Raise} & \mathcal{F}[\mathbf{try}\,P[\mathbf{raise}]\,\mathbf{as}\,x\,\mathbf{in}\,M\,\mathbf{otherwise}\,N] \longrightarrow \mathcal{F}[N] \parallel \lightning P \\
\textsc{E-RaiseChild} & \circ P[\mathbf{raise}] \longrightarrow \lightning P \\
\textsc{E-RaiseMain} & \bullet P[\mathbf{raise}] \longrightarrow \mathbf{halt} \parallel \lightning P \\
\textsc{E-LiftC} & \mathcal{G}[\mathcal{C}] \longrightarrow \mathcal{G}[\mathcal{D}], \quad \text{if } \mathcal{C} \longrightarrow \mathcal{D} \\
\textsc{E-LiftM} & \phi M \longrightarrow \phi N, \quad \text{if } M \longrightarrow_{\mathsf{M}} N
\end{array}
$$

Figure 9.5: Reduction and Equivalence for Terms and Configurations

Remark. *We could implement an alternative reduction rule for E-CloseZap:*

$$\mathcal{F}[\textbf{\textit{close}}\, a] \parallel \lightning\, b \parallel a(\varepsilon) \rightsquigarrow b(\varepsilon) \longrightarrow \mathcal{F}[()]$$

where closing a cancelled channel does not raise an exception. We elected to raise an exception on closing an endpoint where the peer endpoint is cancelled for symmetry with E-ReceiveZap, and also since the above behaviour can already be achieved by cancelling the endpoint:

$$\begin{aligned}
&\mathcal{F}[\textbf{\textit{cancel}}\, a] \parallel \lightning\, b \parallel a(\varepsilon) \rightsquigarrow b(\varepsilon) \\
&\quad\longrightarrow \\
&\mathcal{F}[()] \parallel \lightning\, a \parallel \lightning\, b \parallel a(\varepsilon) \rightsquigarrow b(\varepsilon) \\
&\quad\equiv \\
&\mathcal{F}[()]
\end{aligned}$$

Handling Exceptions. The E-Raise rule invokes the **otherwise** clause if an exception is raised, while also cancelling all endpoints in the enclosing pure context. The structure of handling contexts ensures that exceptions are handled by the *innermost* handler.

If an unhandled exception occurs in a child thread, then all free endpoints in the evaluation context are cancelled and the thread is terminated (E-RaiseChild). If the exception is in the main thread then all free endpoints are cancelled and the main thread reduces to **halt** (E-RaiseMain).

9.3 Metatheory

Even in the presence of channel cancellation and exceptions, EGV retains all of the properties enjoyed by SGV and AGV. As before, we begin by describing a runtime type system for EGV, which differs slightly to that of AGV in order to account for the self-dual End type, as well as halted threads and zapper threads. Next, we show preservation, deadlock-freedom, global progress, confluence, and termination.

9.3.1 Runtime Typing

To state our main results we require typing rules for names and configurations. These are given in Fig. 9.6.

The configuration typing judgement has the shape $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$, which states that under type environment Γ, runtime environment Δ, and thread flag ϕ, configuration $\mathcal{C}$ is well-typed. We additionally require that $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$. Thread flags ensure that there can be at most one top-level thread which can return a value: $\bullet$ denotes a configuration with a top-level thread and $\circ$ denotes a configuration without. The main thread returns the result of running a program. Any configuration $\mathcal{C}$ such that $\Gamma; \Delta \vdash^{\bullet} \mathcal{C}$ has exactly one main thread or halted

Term Typing $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Name}\ \frac{}{a : S \vdash a : S}$$

Session Slicing $\boxed{S/\overrightarrow{A}}$

$$S/\varepsilon = S \qquad !A.S/A \cdot \overrightarrow{A} = S/\overrightarrow{A}$$

Queue Typing $\boxed{\Gamma \vdash \overrightarrow{V} : \overrightarrow{A}}$

$$\frac{}{\cdot \vdash \varepsilon : \varepsilon} \qquad \frac{\Gamma_1 \vdash V : A \quad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma_1, \Gamma_2 \vdash V \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}}$$

Configuration Typing $\boxed{\Gamma; \Delta \vdash^{\phi} \mathcal{C}}$

$$\textsc{T-Nu}\ \frac{\Gamma; \Delta, a : S^{\sharp} \vdash^{\phi} \mathcal{C}}{\Gamma; \Delta \vdash^{\phi} (\nu a)\mathcal{C}} \qquad \textsc{T-Mix}\ \frac{\Gamma_1; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2; \Delta_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}}$$

$$\textsc{T-Connect}_1\ \frac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}} \qquad \textsc{T-Connect}_2\ \frac{\Gamma_1; \Delta_1, a : \overline{S} \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : S; \Delta_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}}$$

$$\textsc{T-Main}\ \frac{\Gamma \vdash M : A}{\Gamma; \cdot \vdash^{\bullet} \bullet M} \qquad \textsc{T-Child}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma; \cdot \vdash^{\circ} \circ M} \qquad \textsc{T-Halt}\ \frac{}{\cdot; \cdot \vdash^{\bullet} \mathbf{halt}} \qquad \textsc{T-Zap}\ \frac{}{a : S; \cdot \vdash^{\circ} \lightning a}$$

$$\textsc{T-Buffer}\ \frac{S/\overrightarrow{A} = \overline{S'/\overrightarrow{B}} \quad \Gamma_1 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2; a : S, b : S' \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Flag Combination $\boxed{\phi_1 + \phi_2 = \phi_3}$

$$\bullet + \circ = \bullet \qquad \circ + \bullet = \bullet \qquad \circ + \circ = \circ \qquad \bullet + \bullet \text{ undefined}$$

Session Type Reduction $\boxed{S \longrightarrow S'}$

$$?A.S \longrightarrow S \qquad !A.S \longrightarrow S$$

Environment Reduction $\boxed{\Gamma; \Delta \longrightarrow \Gamma'; \Delta'}$

$$\frac{S \longrightarrow S'}{\Gamma, a : S; \Delta \longrightarrow \Gamma, a : S'; \Delta} \qquad \frac{S \longrightarrow S'}{\Gamma; \Delta, a : S \longrightarrow \Gamma; \Delta, a : S'} \qquad \frac{S \longrightarrow S'}{\Gamma; \Delta, a : S^{\sharp} \longrightarrow \Gamma; \Delta, a : S'^{\sharp}}$$

Figure 9.6: Runtime Typing

thread as a subconfiguration. We write $\Gamma;\Delta \vdash^\bullet \mathcal{C} : A$ whenever the derivation of $\Gamma;\Delta \vdash^\bullet \mathcal{C}$ contains a subderivation of the form

$$\frac{\Gamma' \vdash M : A}{\Gamma';. \vdash^\bullet \bullet M} \qquad \text{or} \qquad \frac{}{\cdot;\cdot \vdash^\bullet \mathbf{halt}}$$

As before, it is helpful to define a *ground configuration* for EGV.

Definition 15 (Ground Configuration)**.** *We say that a configuration $\mathcal{C}$ is a* ground configuration *if there exists A such that $\cdot;\cdot \vdash^\bullet \mathcal{C} : A$ and A contains no session types or function types.*

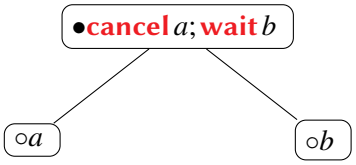
The T-Nu rule introduces a channel name; T-Connect$_1$ and T-Connect$_2$ connect two configurations over a channel; T-Mix composes two configurations that share no channels. The latter three rules use the $+$ operator to combine the flags from subconfigurations. The T-Main and T-Child rules introduce main and child threads. Child threads always return the unit value. The T-Halt rule types the **halt** configuration, which signifies that an unhandled exception has occurred in the main thread. The T-Zap rule types a zapper thread, given a single name in the type environment. The T-Buffer rule ensures that buffers contain values corresponding to the session types of their endpoints. This is the only rule that consumes names from the runtime environment.

Buffers rely on two auxiliary judgements. The queue typing judgement $\Gamma \vdash \overrightarrow{V} : \overrightarrow{A}$ states that under type environment Γ, the sequence of values $\overrightarrow{V}$ have types $\overrightarrow{A}$. The session slicing operator $S/\overrightarrow{A}$ captures reasoning about session types discounting values contained in the buffer. The session types of two buffer endpoints are compatible if they are dual up to values contained in the buffer. The partiality of the slicing operator ensures that at least one queue in a buffer is always empty.

Why T-Mix? A key concept in EGV is that **cancel** severs a communication topology. For the sake of argument, consider SGV (with split ends) extended with **cancel** and zapper threads. Next, consider the following configuration where the main thread cancels name a.

$$(\nu a)(\nu b)(\circ a \parallel \circ b \parallel \bullet(\mathbf{cancel}\, a; \mathbf{wait}\, b))$$

We can visualise this (simple) communication topology as a tree:



We can also show a typing derivation without the use of T-Mix. Let **D** be the following derivation:

$$\dfrac{\dfrac{\dfrac{}{a : \mathsf{End}_? \vdash a : \mathsf{End}_?}\,\textsc{T-Name}}{a : \mathsf{End}_? \vdash \mathbf{cancel}\, a : \mathbf{1}}\,\textsc{T-Cancel} \quad \dfrac{\dfrac{}{b : \mathsf{End}_? \vdash b : \mathsf{End}_?}\,\textsc{T-Name}}{b : \mathsf{End}_? \vdash \mathbf{wait}\, b : \mathbf{1}}\,\textsc{T-Wait}}{a : \mathsf{End}_?, b : \mathsf{End}_? \vdash \mathbf{cancel}\, a; \mathbf{wait}\, b : \mathbf{1}}\,\textsc{T-LetUnit}}{a : \mathsf{End}_?, b : \mathsf{End}_? \vdash^{\bullet} \bullet(\mathbf{cancel}\, a; \mathbf{wait}\, b)}\,\textsc{T-Main}$$

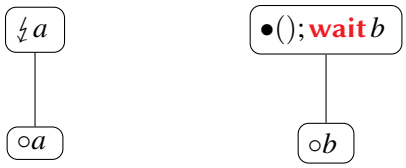
Composing, we can show the typing derivation for the entire configuration:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : \mathsf{End}_! \vdash a : \mathsf{End}_!}\,\textsc{T-Name}}{a : \mathsf{End}_! \vdash^{\circ} \circ a}\,\textsc{T-Thread} \quad \dfrac{\dfrac{\dfrac{}{b : \mathsf{End}_! \vdash b : \mathsf{End}_!}\,\textsc{T-Name}}{b : \mathsf{End}_! \vdash^{\circ} \circ b}\,\textsc{T-Thread} \quad \mathbf{D}}{a : \mathsf{End}_?, b : \mathsf{End}_!^{\sharp} \vdash^{\bullet} \circ b \parallel \bullet(\mathbf{cancel}\, a; \mathbf{wait}\, b)}\,\textsc{T-Connect}_1}{a : \mathsf{End}_!^{\sharp}, b : \mathsf{End}_!^{\sharp} \vdash^{\bullet} \circ a \parallel \circ b \parallel \bullet(\mathbf{cancel}\, a; \mathbf{wait}\, b)}\,\textsc{T-Connect}_1}{a : \mathsf{End}_!^{\sharp} \vdash^{\bullet} (\nu b)(\circ a \parallel \circ b \parallel \bullet(\mathbf{cancel}\, a; \mathbf{wait}\, b))}\,\textsc{T-Nu}}{\cdot \vdash^{\bullet} (\nu a)(\nu b)(\circ a \parallel \circ b \parallel \bullet(\mathbf{cancel}\, a; \mathbf{wait}\, b))}\,\textsc{T-Nu}$$

Now, consider the configuration after a single reduction step (modulo equivalence):

$$\begin{array}{c}(\nu a)(\nu b)(\circ a \parallel \circ b \parallel \bullet(\mathbf{cancel}\, a; \mathbf{wait}\, b)) \\ \Longrightarrow \\ (\nu a)(\nu b)(\circ a \parallel \circ b \parallel \lightning a \parallel \bullet((); \mathbf{wait}\, b))\end{array}$$

Notice now that the communication topology has been severed: we have two configurations which are connected without a channel between them.



We require T-Mix to type this configuration, since the threads $\lightning a$ and $\bullet((); \mathbf{wait}\, b)$ are independent. Let us consider the subderivation proving $a : \mathsf{End}_?, b : \mathsf{End}_?; \cdot \vdash^{\bullet} \lightning a \parallel \bullet((); \mathbf{wait}\, b)$.

$$\dfrac{\dfrac{}{a : \mathsf{End}_? \vdash^{\circ} \lightning a}\,\textsc{T-Zap} \quad \dfrac{\dfrac{\dfrac{}{\cdot \vdash () : \mathbf{1}}\,\textsc{T-Unit} \quad \dfrac{\dfrac{}{b : \mathsf{End}_? \vdash b : \mathsf{End}_?}\,\textsc{T-Name}}{b : \mathsf{End}_? \vdash \mathbf{wait}\, b : \mathbf{1}}\,\textsc{T-Wait}}{b : \mathsf{End}_? \vdash (); \mathbf{wait}\, b) : \mathbf{1}}\,\textsc{T-LetUnit}}{b : \mathsf{End}_? \vdash^{\bullet} \bullet((); \mathbf{wait}\, b)}\,\textsc{T-Main}}{a : \mathsf{End}_?, b : \mathsf{End}_? \vdash^{\bullet} \lightning a \parallel \bullet((); \mathbf{wait}\, b)}\,\textsc{T-Mix}$$

Furthermore, we might envisage a construct which spawns a process without creating a channel:

$$\text{T-Spawn}\ \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \mathbf{spawn}\, M : \mathbf{1}}$$

with the associated reduction rule

$$\text{E-Spawn}\ \mathcal{F}[\mathbf{spawn}\, M] \longrightarrow \mathcal{F}[()] \parallel \circ M$$

Since **cancel** allows a channel to be discarded, it is possible to encode **spawn** in EGV:

$$\mathbf{spawn}\, M \triangleq \mathbf{let}\, s = \mathbf{fork}\, (\lambda x.\mathbf{cancel}\, x; M)\, \mathbf{in}$$
$$\mathbf{cancel}\, s$$

Since **cancel** is asynchronous, both threads may evaluate in parallel.

Our decision to include Mix and a self-dual End type is not without precedent: both Caires and Pérez [26] and Mostrous and Vasconcelos [147] consider exceptional behaviour, and in spite of being logically-inspired, deadlock-free calculi, both choose to have a self-dual End. Furthermore, Caires and Pérez [26] have an explicit Mix rule, and Mostrous and Vasconcelos [147] show that the Mix rule may be derived as they treat endpoints of type End as affine.

We may now explore the properties that EGV enjoys.

9.3.2 Preservation

The lemmas for manipulating term and configuration contexts are identical to those for AGV, and as before, are all established by induction on the structure of the evaluation contexts.

Lemma 57 (Typeability of subterms)**.** *If* $\boldsymbol{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : A$*, then there exists some* B *such that* $\boldsymbol{D}$ *has a subderivation* $\boldsymbol{D}'$ *that concludes* $\Gamma_2 \vdash M : B$*, and the position of* $\boldsymbol{D}'$ *in* $\boldsymbol{D}$ *corresponds to the position of the hole in* E*.*

Lemma 58 (Replacement (evaluation contexts))**.** *If:*

- $\boldsymbol{D}$ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash E[M] : A$
- $\boldsymbol{D}'$ *is a subderivation of* $\boldsymbol{D}$ *concluding* $\Gamma_2 \vdash M : B$
- *The position of* $\boldsymbol{D}'$ *in* $\boldsymbol{D}$ *corresponds to that of the hole in* E
- $\Gamma_3 \vdash N : B$
- Γ_1, Γ_3 *is defined*

then $\Gamma_1, \Gamma_3 \vdash E[N] : A$*.*

Lemma 59 (Typeability of subconfigurations)**.** *If* $\boldsymbol{D}$ *is a derivation of* $\Gamma;\Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$*, then there exist* Γ',Δ',ϕ' *such that* $\boldsymbol{D}$ *has a subderivation* $\boldsymbol{D'}$ *that concludes* $\Gamma';\Delta' \vdash^{\phi'} \mathcal{C}$*, and the position of* $\boldsymbol{D'}$ *in* $\boldsymbol{D}$ *corresponds to the position of the hole in* $\mathcal{G}$*.*

Lemma 60 (Replacement (configurations))**.** *If:*

- $\boldsymbol{D}$ *is a derivation of* $\Gamma;\Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$
- $\boldsymbol{D'}$ *is a subderivation of* $\boldsymbol{D}$ *concluding that* $\Gamma';\Delta' \vdash^{\phi'} \mathcal{C}$ *for some* Γ',Δ',ϕ'
- $\Gamma'';\Delta'' \vdash^{\phi'} \mathcal{C}'$ *for some* Γ'',Δ'' *such that* $\Gamma';\Delta' \longrightarrow^{?} \Gamma'';\Delta''$
- *The position of* $\boldsymbol{D}$ *in* $\boldsymbol{D'}$ *corresponds to that of the hole in* $\mathcal{G}$

then there exist some Γ''',Δ''' *such that* $\Gamma''';\Delta''' \vdash^{\phi} \mathcal{G}[\mathcal{C}']$ *and* $\Gamma;\Delta \longrightarrow^{?} \Gamma''';\Delta'''$*.*

Preservation for the functional fragment of EGV is standard.

Lemma 61 (Preservation (Terms))**.** *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M M'$*, then* $\Gamma \vdash M' : A$*.*

Again, we write Ψ for the restriction of type environments Γ to contain runtime names but no variables:

$$\Psi ::= \cdot \mid \Psi, a : S$$

Unlike in SGV and AGV, preservation of typing by configuration reduction holds only for closed configurations. This is because our semantics safely discards names, but does not account for discarding arbitrary linear variables. One could imagine generalising the theorem by introducing a mechanism for discarding arbitrary linear variables using *destructors* such as in C++ or Rust, but this is out of scope for EGV.

Theorem 31 (Preservation)**.** *If* $\Psi;\Delta \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$*, then there exist* Ψ',Δ' *such that* $\Psi;\Delta \longrightarrow^{?} \Psi';\Delta'$ *and* $\Psi';\Delta' \vdash^{\phi} \mathcal{C}'$*.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{C}'$, making use of Lemmas 57–60, and lemmas for subconfiguration typeability and replacement. Most interesting are T-Zap and T-Raise, which we prove here. The full proof can be found in Appendix D.

Case E-Zap

$$\lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

where $\mathsf{fn}(U) = \{c_i\}_i$.

Assumption:

$$\dfrac{\dfrac{}{a : S; \cdot \vdash^{\circ} \lightning a} \quad \dfrac{\overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_1, \Gamma_2 \vdash U \cdot \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2, \Gamma_3; a : \overline{S}, b : T \vdash^{\circ} a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3; a : S^{\sharp}, b : T \vdash^{\circ} \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By the definition of slicing, we have that there exist some A and S' such that $\overline{S} = !A.\overline{S'}$. Thus, we may refine our judgement:

$$\dfrac{\dfrac{}{a : ?A.S'; \cdot \vdash^{\circ} \lightning a} \quad \dfrac{!A.\overline{S'}/A \cdot \overrightarrow{A'} = \overline{T/\overrightarrow{B}} \quad \Gamma_1, \Gamma_2 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A'} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1, \Gamma_2, \Gamma_3; a : !A.\overline{S'}, b : T \vdash^{\circ} a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3; a : S^{\sharp}, b : T \vdash^{\circ} \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By the definition of buffer typing, we have that $\Gamma_1 \vdash U : A$. By the definition of the reduction rule, $\mathsf{fn}(U) = \{c_i\}_i$, and by assumption, Γ_1 contains only runtime names. Thus, we may conclude that U is closed and therefore that $\Gamma_1 = c_1 : S_1, \ldots c_n : S_n$ for some session types $S_1, \ldots S_n$.

By the definition of slicing, we have that $!A.\overline{S'}/A \cdot \overrightarrow{A'} \iff \overline{S'}/\overrightarrow{A'}$. Correspondingly, by T-Buffer, we may show

$$\dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/\overrightarrow{B}} \quad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A'} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3; a : \overline{S'}, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By repeated applications of T-Zap and T-Mix, we have that

$$\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : \overline{S'}, b : T \vdash^{\circ} \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

Recomposing:

$$\dfrac{\dfrac{}{a : S'; \cdot \vdash^{\circ} \lightning a} \quad \dfrac{\dfrac{}{c_1 : S_1; \cdot \vdash^{\circ} \lightning c_1} \quad \dfrac{\dfrac{}{c_n : S_n; \cdot \vdash^{\circ} \lightning c_n} \quad \dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/\overrightarrow{B}} \quad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A'} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3; a : \overline{S'}, b : T \vdash^{\circ} a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\vdots}}{\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : \overline{S'}, b : T \vdash^{\circ} \lightning c_1 \parallel \ldots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : S'^{\sharp}, b : T \vdash^{\circ} \lightning a \parallel \lightning c_1 \parallel \ldots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Finally, we must show environment reduction:

$$\dfrac{?A.S' \longrightarrow S'}{\Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : (?A.S')^{\sharp}, b : T \longrightarrow \Gamma_2, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : S'^{\sharp}, b : T}$$

as required.

Case E-Raise

$$\bullet E[\textbf{try}\, P[\textbf{raise}]\, \textbf{as}\, x\, \textbf{in}\, M\, \textbf{otherwise}\, N] \longrightarrow E[N] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n$$

where $\mathsf{fn}(P) = \{c_i\}_i$.

Assumption:

$$\frac{\Gamma \vdash E[\textbf{try}\, P[\textbf{raise}]\, \textbf{as}\, x\, \textbf{in}\, M\, \textbf{otherwise}\, N] : A'}{\Gamma; \cdot \vdash^\bullet \bullet E[\textbf{try}\, P[\textbf{raise}]\, \textbf{as}\, x\, \textbf{in}\, M\, \textbf{otherwise}\, N]}$$

By Lemma 57, there exist $\Gamma_1, \Gamma_2, A, B, C$ such that $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ and

$$\frac{\Gamma_2 \vdash P[\textbf{raise}] : A \quad \Gamma_3, x : B \vdash M : C \quad \Gamma_3 \vdash N : C}{\Gamma_2, \Gamma_3 \vdash \textbf{try}\, P[\textbf{raise}]\, \textbf{as}\, x\, \textbf{in}\, M\, \textbf{otherwise}\, N : C}$$

Since Γ contains only runtime names and $\mathsf{fn}(P) = \{c_i\}_i$, we know that $\Gamma_2 = c_1 : S_1, \ldots, c_n : S_n$ for some $\{S_i\}_i$.

By Lemma 58, we have that:

$$\frac{}{\Gamma_1, \Gamma_3 \vdash E[N] : A'}$$

By repeated applications of T-Zap and T-Mix, we have that $\Gamma_2 \vdash \lightning c_1 \parallel \cdots \parallel \lightning c_n$.

Therefore, recomposing:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_3 \vdash E[N] : C}{\Gamma_1, \Gamma_3; \cdot \vdash^\bullet \bullet E[N]} \qquad \dfrac{\dfrac{}{c_1 : S_1; \cdot \vdash^\circ \lightning c_1} \qquad \dfrac{\dfrac{}{c_{n-1} : S_{n-1}; \cdot \vdash^\circ \lightning c_{n-1}} \qquad \dfrac{}{c_n : S_n; \cdot \vdash^\circ \lightning c_n}}{\vdots}}{c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^\circ \lightning c_1 \parallel \cdots \parallel \lightning c_n}}{\Gamma_1, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^\bullet \bullet E[N] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n}$$

as required.

□

Equivalence does not preserve typeability of configurations. Nonetheless, no reductions depend on the ill-typed use of an equivalence. The properties of equivalence (i.e., that equivalence preserves typing as long as the associativity axiom is not used, and that it is always possible to safely re-associate configurations either directly or by first commuting two threads) are unchanged from SGV and AGV.

Theorem 32 (Preservation Modulo Equivalence). *If $\Psi; \Delta \vdash^\phi \mathcal{C}$, $\mathcal{C} \equiv \mathcal{D}$, and $\mathcal{D} \longrightarrow \mathcal{D}'$, then:*

1. *There exists some $\mathcal{E} \equiv \mathcal{D}$ and some $\mathcal{E}'$ such that $\Psi; \Delta \vdash^\phi \mathcal{E}$ and $\mathcal{E} \longrightarrow \mathcal{E}'$*
2. *There exist Ψ', Δ' such that $\Psi; \Delta \longrightarrow^? \Psi'; \Delta'$ and $\Psi'; \Delta' \vdash^\phi \mathcal{E}'$*
3. *$\mathcal{D}' \equiv \mathcal{E}'$*

The proof of Theorem 32 is identical to that of SGV and AGV.

9.3.3 Global Progress

To prove that EGV enjoys a strong notion of progress we identify a *canonical form* for configurations. We prove that every well-typed configuration is equivalent to a well-typed configuration in canonical form, and that irreducible ground configurations are equivalent to either a value or **halt**. Due to the inclusion of zapper threads, halted threads, and T-Mix, the canonical form and progress proofs require some additional care over the previously-defined results for SGV and AGV.

The functional fragment of EGV enjoys progress.

Lemma 62 (Progress: Open Terms)**.** *If* $\Psi \vdash M : A$*, then either:*

- M *is a value;*
- *there exists some* M' *such that* $M \longrightarrow_M M'$*; or*
- *there exist* E, N *such that* M *can be written* $E[N]$*, where* N *is either* **raise** *or a communication / concurrency primitive of the form:* **fork** V*,* **send** $V\,W$*,* **receive** V*,* **close** V*, or* **cancel** V*.*

Proof. By induction on the derivation of $\Psi \vdash M : A$. □

To reason about progress of configurations, we characterise *canonical forms*, which now make explicit the property that *at most* one name is shared between threads. Recall that $\mathcal{A}$ ranges over auxiliary threads (i.e., child threads, buffer threads, and zapper threads), and $\mathcal{T}$ over top-level threads (Fig. 9.4). Let $\mathcal{M}$ range over configurations of the form:

$$\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_m \parallel \mathcal{T}$$

Definition 16 (Canonical Form)**.** *A configuration* $\mathcal{C}$ *is in* canonical form *if there is a sequence of names* $a_1, \ldots, a_n$*, a sequence of configurations* $\mathcal{A}_1, \ldots, \mathcal{A}_n$*, and a configuration* $\mathcal{M}$*, such that:*

$$\mathcal{C} = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}) \ldots))$$

where $a_i \in \mathsf{fn}(\mathcal{A}_i)$ *for each* $i \in 1..n$*.*

The following lemma implies that communication topologies are always acyclic.

Lemma 63. *If* $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} = \mathcal{G}[\mathcal{D} \parallel \mathcal{E}]$*, then* $\mathsf{fn}(\mathcal{D}) \cap \mathsf{fn}(\mathcal{E})$ *is either* $\emptyset$ *or* $\{a\}$ *for some* a*.*

Proof. By induction on the derivation of $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$; the only interesting rules are those for parallel composition. Due to the global condition that $\mathsf{fn}(\Gamma) \cap \mathsf{fn}(\Delta) = \emptyset$, T-Connect$_1$ and T-Connect$_2$ allow exactly one name to be shared, whereas T-Mix forbids sharing of names. □

All well-typed configurations with a main thread can be written in canonical form.

Theorem 33 (Canonical Forms)**.** *Given $\mathcal{C}$ such that $\Gamma;\Delta \vdash^{\bullet} \mathcal{C}$, there exists some $\mathcal{D} \equiv \mathcal{C}$ such that $\Gamma;\Delta \vdash^{\bullet} \mathcal{D}$ and $\mathcal{D}$ is in canonical form.*

Proof. By induction on the count of ν-bound variables, as with SGV and EGV. The additional features of EGV do not change the essential argument. □

Definition 17. *We say that term M is* ready to perform an action on name *a if M is about to send on, receive on, close, or cancel a. Formally:*

$$\mathit{ready}(a,M) \triangleq \exists E.(M = E[\mathbf{send}\, V\, a]) \vee (M = E[\mathbf{receive}\, a]) \vee (M = E[\mathbf{close}\, a]) \vee (M = E[\mathbf{cancel}\, a])$$

Using the notion of a ready thread, we may classify non-reducing open configurations. We define a notion of *open progress*, this time taking into account zapper threads, halted configurations, and the fact that child threads now have type $\mathbf{1}$.

Definition 18 (Open Progress)**.** *Suppose $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$, where $\mathcal{C}$ is in canonical form and $\mathcal{C} \not\Longrightarrow$. We say that $\mathcal{C}$* satisfies open progress *if:*

1. *$\mathcal{C} = (\nu a)(\mathcal{A} \parallel \mathcal{D})$, where $\Psi = \Psi_1, \Psi_2$ and $\Delta = \Delta_1, \Delta_2$ such that either:*
   (a) *$\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{D}$ where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either:*
      i. *A thread $\circ M$ where $\mathit{ready}(b,M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$; or*
      ii. *A zapper thread $\lightning\, a$; or*
      iii. *A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \overrightarrow{V}$ or $a \in \overrightarrow{W}$*
   (b) *$\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{D}$, where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$*
2. *$\mathcal{C} = \mathcal{A} \parallel \mathcal{M}$, where $\Psi = \Psi_1, \Psi_2$ and either:*
   (a) *$\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either:*
      i. *A thread $\circ M$ where $\mathit{ready}(b,M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$; or*
      ii. *A zapper thread $\lightning\, a$; or*
      iii. *A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \mathsf{fn}(\overrightarrow{V})$ or $a \in \mathsf{fn}(\overrightarrow{W})$*
   (b) *$\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$*
   (c) *$\Delta = \Delta_1, \Delta_2$, where $\Psi_1; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2 \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either:*
      i. *A thread $\circ M$ where either $M = ()$, or $\mathit{ready}(a,M)$ for some $a \in \mathsf{fn}(\Psi_1)$; or*

ii. *A zapper thread* $\lightning a$ *for some* $a \in \mathsf{fn}(\Psi_1)$*; or*

iii. *A buffer* $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ *for some* $a, b \in \mathsf{fn}(\Delta_1)$

3. $\mathcal{C} = \mathcal{T}$*, where either:*

   (a) $\mathcal{T} = \bullet N$*, where* N *is either a value or* $\mathit{ready}(b, N)$ *for some* $b \in \mathsf{fn}(\Psi)$

   (b) $\mathcal{T} = \mathbf{halt}$

All well-typed, non-reducing threads satisfy open progress.

Lemma 64. *Suppose* $\Psi; \Delta \vdash^{\bullet} \mathcal{C}$*, where* $\mathcal{C}$ *is in canonical form and* $\mathcal{C} \not\Longrightarrow$*. Then* $\mathcal{C}$ *satisfies open progress.*

Proof. By induction on the derivation of $\Psi; \Delta \vdash^{\bullet} \mathcal{C}$. The full proof can be found in Appendix D. □

As an immediate corollary, we obtain a more global view of non-reducing, open configurations in canonical form.

Corollary 10 (Progress: Open)**.** *Suppose* $\Psi; \Delta \vdash^{\bullet} \mathcal{C}$ *where* $\mathcal{C}$ *is in canonical form and* $\mathcal{C} \Longrightarrow$*. Let* $\mathcal{C} = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}))\ldots))$*.*

1. *For* $1 \leq i \leq n$*, each thread in* $\mathcal{A}_i$ *is either:*

   (a) *a child thread* $\circ M$ *for which there exists* $b \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{fn}(\Psi)$ *such that* $\mathit{ready}(b, M)$*;*

   (b) *a zapper thread* $\lightning a_i$*; or*

   (c) *a buffer.*

2. $\mathcal{M} = \mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_m \parallel \mathcal{T}$ *such that for* $1 \leq j \leq m$*:*

   (a) $\mathcal{A}'_j$ *is either:*

      i. *a child thread* $\circ N$ *such that* $N = ()$ *or* $\mathit{ready}(b, N)$ *for some* $b \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$*;*

      ii. *a zapper thread* $\lightning b$ *for some* $b \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$*; or*

      iii. *a buffer.*

   (b) *Either* $\mathcal{T} = \bullet N$*, where* N *is either a value or* $\mathit{ready}(b, N)$ *for some* $b \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$*; or* $\mathcal{T} = \mathbf{halt}$*.*

Corollary 10 tells us that open reduction cannot "go wrong". By restricting attention to closed environments, we immediately obtain a tighter progress property.

Corollary 11 (Progress: Closed). *Suppose* $\cdot;\cdot \vdash^{\bullet} \mathcal{C}$ *where* $\mathcal{C}$ *is in canonical form and* $\mathcal{C} \not\Longrightarrow$. *Let* $\mathcal{C} = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M})\ldots))$. *Then:*

1. *For* $1 \leq i \leq n$, *every thread in* $\mathcal{A}_i$ *is either:*
   (a) *a child thread* $\circ M$ *for some* M *such that* $\mathit{ready}(a_i, M)$; *or*
   (b) *a zapper thread* $\lightning a_i$; *or*
   (c) *a buffer.*
2. $\mathcal{M} \equiv \mathcal{T}$ *where either* $\mathcal{T} = \bullet W$ *for some value* W, *or* $\mathcal{T} = \mathbf{halt}$.

The above progress results do not specifically mention deadlock. However, Lemma 63 ensures deadlock-freedom. Nevertheless, communication can still be blocked if an endpoint appears in the value returned by the main thread. A conservative way of disallowing endpoints in the result is to insist that the return type of the program be free of session types and function types (closures may capture endpoints). All configurations of such a program are ground configurations.

Theorem 34 (Global Progress). *Let* $\mathcal{C}$ *be a ground configuration such that* $\mathcal{C} \not\Longrightarrow$. *Then either* $\mathcal{C} \equiv \bullet V$, *for some* V, *or* $\mathcal{C} \equiv \mathbf{halt}$.

Proof. As $\mathcal{C}$ is ground and irreducible, by Lemma 63, we have that no thread can be ready to perform an action, and thus each $\mathcal{A}_i$ in Corollary 11 must be either $\circ()$, a zapper thread, or an empty buffer. The result follows by the garbage collection congruences of Fig. 9.5, which clean up fully-reduced child threads and cancelled buffers and name restrictions. □

9.3.4 Confluence

Like SGV and AGV, EGV enjoys the diamond property.

Theorem 35 (Diamond Property). *If* $\Psi;\Delta \vdash^{\phi} \mathcal{C}$, *and* $\mathcal{C} \Longrightarrow \mathcal{D}_1$, *and* $\mathcal{C} \Longrightarrow \mathcal{D}_2$, *then either* $\mathcal{D}_1 \equiv \mathcal{D}_2$, *or there exists some* $\mathcal{D}_3$ *such that* $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$ *and* $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$.

Proof. Like SGV and AGV, term reduction is deterministic, and reduction on different channels may be performed in any order.

In addition to the critical pair introduced in AGV, a second critical pair arises when sending to a buffer where the peer endpoint has a non-empty buffer and has been cancelled. There is a choice as to whether the value at the head of the queue is cancelled before or after the send takes place. Again, both cases reduce to the same configuration after a single further step.

$$\begin{array}{ccc} & \mathcal{F}[\mathbf{send}\,U\,a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) & \\ \swarrow & & \searrow \\ \mathcal{F}[a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) & & \mathcal{F}[\mathbf{send}\,U\,a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\ \searrow & & \swarrow \\ & \mathcal{F}[a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) & \end{array}$$

□

Remark. *The system becomes non-confluent if we choose to raise an exception when sending to a cancelled buffer. If instead of the current semantics, we were to replace E-Send with the following two rules:*

$$(\nu b)(\mathcal{F}[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \parallel \phi M) \longrightarrow (\nu b)(\mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \parallel \phi M)$$
$$\mathcal{F}[\mathbf{send}\,U\,a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \mathcal{F}[\mathbf{raise}] \parallel \lightning b \parallel \lightning U \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

Then, sending and cancelling peer endpoints of a buffer results in a non-convergent critical pair:

$$\begin{array}{ccc} & (\nu b)(\mathcal{F}[\mathbf{send}\,U\,a] \parallel \mathcal{F}'[\mathbf{cancel}\,b] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})) & \\ \swarrow & & \searrow \\ (\nu b)(\mathcal{F}[a] \parallel \mathcal{F}'[\mathbf{cancel}\,b] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U)) & & (\nu b)(\mathcal{F}[\mathbf{send}\,U\,a] \parallel \mathcal{F}'[()] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})) \\ \Downarrow & & \Downarrow \\ (\nu b)(\mathcal{F}[a] \parallel \mathcal{F}'[()] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U)) & & (\nu b)(\mathcal{F}[\mathbf{raise}] \parallel \mathcal{F}'[()] \parallel \lightning b \parallel \lightning U \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})) \end{array}$$

In either case, the endpoints contained in U will still eventually be cancelled, thus preservation and global progress still hold. However, the lack of confluence affects exactly when *the exception is raised in context $\mathcal{F}$. This decision has practical significance, in that it characterises the race between sending a message and propagating a cancellation notification.*

9.3.5 Termination

As EGV is linear, it has an elementary strong normalisation proof.

Theorem 36 (Strong Normalisation)**.** *If $\Psi;\Delta \vdash^{\phi} \mathcal{C}$, then there are no infinite $\Longrightarrow$ reduction sequences from $\mathcal{C}$.*

Proof. Let the size of a configuration be the sum of the sizes of the abstract syntax trees of all of the terms contained in its main threads, child threads, and buffers, modulo exhaustively applying the garbage collection equivalence rules from left-to-right. The size of a configuration is invariant under $\equiv$ and strictly decreases under $\longrightarrow$, hence $\Longrightarrow$ reduction must always terminate. □

9.4 Extensions

9.4.1 User-defined Exceptions with Payloads

In order to focus on the interplay between exceptions and session types we have thus far considered handling a single kind of exception. In practice it can be useful to distinguish between multiple kinds of user-defined exception, each of which may carry a payload.

Consider again handling the exception in checkDetails. An exception may be raised if the database is corrupt, or if there are too many connections. We might like to handle each case

Syntax

Types	$A, B ::= \cdots \mid \mathsf{Exn}$
Terms	$L, M, N ::= \cdots \mid X(M) \mid \mathbf{raise}\, M \mid \mathbf{try}\, L\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{unless}\, H$
Exception Handlers	$H ::= \{X_i(x_i) \mapsto N_i\}_i$

Runtime Syntax

$$\text{Evaluation Contexts } E ::= \cdots \mid \mathbf{raise}\, E \mid \mathbf{try}\, E\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{unless}\, H$$

Term typing $\boxed{\Gamma \vdash M : A}$

$$\text{TP-Exn}\ \frac{\Sigma(X) = A \quad \Gamma \vdash M : A}{\Gamma \vdash X(M) : \mathsf{Exn}} \qquad \text{TP-Raise}\ \frac{\Gamma \vdash M : \mathsf{Exn}}{\Gamma \vdash \mathbf{raise}\, M : A} \qquad \text{TP-Try}\ \frac{\Gamma_1 \vdash L : A \quad \Gamma_2, x : A \vdash M : B \quad (\Gamma_2, y_i : \Sigma(X_i) \vdash N_i : B)_i}{\Gamma_1, \Gamma_2 \vdash \mathbf{try}\, L\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{unless}\, \{X_i(y_i) \mapsto N_i\}_i : B}$$

Term and Configuration Reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$ $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

EP-Val	$\mathbf{try}\, V\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{unless}\, H$	$\longrightarrow_{\mathsf{M}}$	$M\{V/x\}$
EP-Raise	$\mathcal{F}[\mathbf{try}\, E[\mathbf{raise}\, X(V)]\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{unless}\, H]$	$\longrightarrow$	$\mathcal{F}[N\{V/y\}] \parallel \lightning E$ where $X \notin \mathsf{handled}(E)$, $(X(y) \mapsto N) \in H$
EP-RaiseChild	$\circ E[\mathbf{raise}\, X(V)]$	$\longrightarrow$	$\lightning E \parallel \lightning V$ where $X \notin \mathsf{handled}(E)$
EP-RaiseMain	$\bullet E[\mathbf{raise}\, X(V)]$	$\longrightarrow$	$\mathbf{halt} \parallel \lightning E \parallel \lightning V$ where $X \notin \mathsf{handled}(E)$

Figure 9.7: User-defined Exceptions with Payloads

separately:

$\mathsf{exnServer4}(s) \triangleq$
  let $((username, password), s) =$ **receive** s **in**
  try checkDetails$(username, password)$ **as** res **in**
    if res **then let** $s =$ **select** Authenticated s **in** serverBody(s)
    else let $s =$ **select** AccessDenied s **in close** s
  unless
    DBCorrupt(y) $\mapsto$ **cancel** s; log(`"Database Corrupt: " + y`)
    TooManyConnections(y) $\mapsto$ **cancel** s; log(`"Too many connections: " + y`)

An exception in checkDetails might be raised by the term **raise** DatabaseCorrupt$(filename)$, for example. Our approach generalises straightforwardly to handle this example.

Syntax. Figure 9.7 shows extensions to EGV for exceptions with payloads. We introduce a type of exceptions, Exn. We let X range over a countably infinite set $\mathbb{E}$ of exception names, and assume a type schema function $\Sigma(X) = A$ mapping exception names to payload types. We extend **raise** to take a term of type Exn as its argument. Finally, we generalise **try** L **as** x **in** M **otherwise** N to **try** L **as** x **in** M **unless** H, where H is an exception handler with clauses $\{X_i(y_i) \mapsto N_i\}_i$, such that X_i is an exception name; y_i binds the payload; and N_i is the clause to be evaluated when the exception is raised.

Typing Rules. The TP-Exn rule ensures that an exception's payload matches its expected type. The TP-Raise and TP-Try rules are the natural extensions of T-Raise and T-Try.

Semantics. Our presentation is similar to operational accounts of effect handlers; the formulation here is inspired by that of Hillerström et al. [91]. To define the semantics of the generalised exception handling construct, we first introduce the auxiliary function $\mathsf{handled}(E)$, which defines the exceptions handled in a given evaluation context:

$$\mathsf{handled}(P) = \emptyset \qquad \mathsf{handled}(\textbf{try}\, E\, \textbf{as}\, x\, \textbf{in}\, M\, \textbf{unless}\, H) = \mathsf{handled}(E) \cup \mathsf{dom}(H)$$
$$\mathsf{handled}(E) = \mathsf{handled}(E'), \quad \text{if } E \text{ is not a } \textbf{try} \text{ and } E' \text{ is the immediate subcontext of } E$$

The EP-Raise rule handles an exception. The side conditions ensure that the exception is caught by the nearest matching handler and is handled by the appropriate clause. As with plain EGV, all free names are safely discarded. The EP-RaiseChild and EP-RaiseMain rules cover the cases where an exception is unhandled. Due to the use of the handled function we no longer require pure contexts. All of EGV's metatheoretic properties (preservation, global progress, confluence, and termination) adapt straightforwardly to this extension.

9.4.2 Unrestricted Types and Access Points

Unrestricted (intuitionistic) types allow some values to be used in a non-linear fashion. Access points [77] provide a more flexible method of session initiation than **fork**, allowing two threads to dynamically establish a session. Both features are useful in practice: unrestricted types because some data is naturally multi-use, and access points because they admit cyclic communication topologies supporting racey stateful servers such as chat servers. *Access points* decouple spawning a thread from establishing a session. An access point has the unrestricted type $\mathsf{AP}(S)$; we write $\mathsf{un}(A)$ to mean that A is unrestricted and $\mathsf{un}(\Gamma)$ if $\mathsf{un}(A_i)$ for all $x_i : A_i \in \Gamma$. Figure 9.8 shows the syntax, typing rules, and reduction rules for EGV extended with access points.

Unrestricted Types. To support unrestricted types, we use a standard splitting judgement ($\Gamma = \Gamma_1 + \Gamma_2$), which allows variables of unrestricted type to be shared across sub-environments, but requires linear variables to be used only in a single sub-environment. We relax rule T-Var

Syntax

Types	$A ::= \cdots \mid \mathsf{AP}(S)$
Access Point Names	z
Terms	$M ::= \cdots \mid z \mid \mathbf{spawn}\, M \mid \mathbf{new}_S \mid \mathbf{request}\, M \mid \mathbf{accept}\, M$
Configurations	$\mathcal{C} ::= \cdots \mid (\nu z)\mathcal{C} \mid z(\mathcal{X}, \mathcal{Y})$
Runtime typing environments	$\Delta ::= \cdots \mid \Delta, z : S$
Evaluation contexts	$E ::= \cdots \mid \mathbf{request}\, E \mid \mathbf{accept}\, E$
Pure contexts	$P ::= \cdots \mid \mathbf{request}\, P \mid \mathbf{accept}\, P$

Splitting $\boxed{\Gamma = \Gamma_1 + \Gamma_2}$

$$\frac{}{\cdot = \cdot + \cdot} \qquad \frac{\mathsf{un}(A)}{\Gamma, x : A = (\Gamma_1, x : A) + (\Gamma_2, x : A)} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x : A = (\Gamma_1, x : A) + \Gamma_2}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, x : A = \Gamma_1 + (\Gamma_2, x : A)}$$

Typing $\boxed{\Gamma \vdash M : A}$

T-Var
$$\frac{x : A \in \Gamma \quad \mathsf{un}(\Gamma)}{\Gamma \vdash x : A}$$

T-App
$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma \vdash MN : B}$$

...

TA-Spawn
$$\frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash \mathbf{spawn}\, M : \mathbf{1}}$$

TA-New
$$\frac{}{\Gamma \vdash \mathbf{new}_S : \mathsf{AP}(S)}$$

TA-Request
$$\frac{\Gamma \vdash M : \mathsf{AP}(S)}{\Gamma \vdash \mathbf{request}\, M : \overline{S}}$$

TA-Accept
$$\frac{\Gamma \vdash M : \mathsf{AP}(S)}{\Gamma \vdash \mathbf{accept}\, M : S}$$

Reduction $\boxed{\mathcal{C} \longrightarrow \mathcal{D}}$

E-Spawn	$\mathcal{F}[\mathbf{spawn}\, M] \longrightarrow \mathcal{F}[()] \parallel \circ M$	
E-New	$\mathcal{F}[\mathbf{new}_S] \longrightarrow (\nu z)(\mathcal{F}[z] \parallel z(\varepsilon, \varepsilon))$	z is fresh
E-Accept	$\mathcal{F}[\mathbf{accept}\, z] \parallel z(\mathcal{X}, \mathcal{Y}) \longrightarrow (\nu a)(\mathcal{F}[a] \parallel z(\{a\} \cup \mathcal{X}, \mathcal{Y}))$	a is fresh
E-Request	$\mathcal{F}[\mathbf{request}\, z] \parallel z(\mathcal{X}, \mathcal{Y}) \longrightarrow (\nu a)(\mathcal{F}[a] \parallel z(\mathcal{X}, \{a\} \cup \mathcal{Y}))$	a is fresh
E-Match	$z(\{a\} \cup \mathcal{X}, \{b\} \cup \mathcal{Y}) \longrightarrow z(\mathcal{X}, \mathcal{Y}) \parallel a(\varepsilon) \rightsquigarrow b(\varepsilon)$	

Configuration Typing $\boxed{\Gamma; \Delta \vdash^{\phi} \mathcal{C}}$

TA-ApName
$$\frac{\Gamma, z : \mathsf{AP}(S); \Delta, z : S \vdash^{\phi} \mathcal{C}}{\Gamma; \Delta \vdash^{\phi} (\nu z)\mathcal{C}}$$

TA-Ap
$$\frac{\mathsf{un}(\Gamma)}{\Gamma, z : \mathsf{AP}(S); \mathcal{X} : \overline{S}, \mathcal{Y} : S, z : S \vdash^{\circ} z(\mathcal{X}, \mathcal{Y})}$$

TA-ConnectN
$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \quad \Gamma_1, \overrightarrow{a : S}; \Delta_1, \overrightarrow{b : T} \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, \overrightarrow{b : T}; \Delta_2, \overrightarrow{a : S} \vdash^{\phi_2} \mathcal{D}}{\Gamma; \Delta_1, \Delta_2, \overrightarrow{a : S^{\sharp}}, \overrightarrow{b : T^{\sharp}} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}}$$

Figure 9.8: Access Points

to allow the use of unrestricted environments, and adapt all rules containing multiple subterms to use the splitting judgement; we detail T-App in the figure; the adaptations of other rules are similar. While unrestricted types are useful in general, we show the specific case of unrestricted access points.

Access points. The **spawn** M construct spawns M as a new thread, $\mathbf{new}_S$ creates a fresh access point, and **request** M and **accept** M generate fresh endpoints that are matched up nondeterministically to form channels. With access points we can macro-express **fork**:

$$\mathbf{fork}\, M \triangleq \mathbf{let}\, ap = \mathbf{new}_S\, \mathbf{in}\, \mathbf{spawn}\, (M\, (\mathbf{accept}\, ap)); \mathbf{request}\, ap$$

Reduction rules. We let z range over access point names. Configuration $(\nu z)\mathcal{C}$ denotes binding access point name z in $\mathcal{C}$, and $z(\mathcal{X}, \mathcal{Y})$ is an access point with name z and two sets $\mathcal{X}$ and $\mathcal{Y}$ containing endpoints to be matched.

Rule E-Spawn creates a new child thread but, unlike **fork**, returns the unit value instead of creating a channel and returning an endpoint. Rule E-New creates a new access point with fresh name z. Rules E-Accept and E-Request create a fresh name a, returning the newly-created name to the thread, and adding the name to sets $\mathcal{X}$ and $\mathcal{Y}$ respectively. Rule E-Match matches two endpoints a and b contained in $\mathcal{X}$ and $\mathcal{Y}$, and creates an empty buffer $a(\varepsilon) \leftrightsquigarrow b(\varepsilon)$.

Configuration typing. Configuration typing judgements again have the shape $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$. Whereas Γ may contain unrestricted variables, Δ remains entirely linear.

Read bottom-up, rule TA-ApName adds an unrestricted reference $z : \mathsf{AP}(S)$ to Γ, and a linear entry $z : S$ to Δ. Rule TA-Ap types an access point configuration. We write $\mathcal{X} : S$ for $a_1 : S, \ldots, a_n : S$, where $\mathcal{X} = \{a_1, \ldots, a_n\}$. For an access point $z(\mathcal{X}, \mathcal{Y})$ to be well-typed, Δ must contain $z : S$, along with the names in $\mathcal{X}$ having type $\overline{S}$ and the names in $\mathcal{Y}$ having type S. Rule T-ConnectN generalises T-Connect$_1$ and T-Connect$_2$ to allow any number of channels to communicate across a configuration; this therefore introduces the possibility of deadlock.

Interaction with cancellation. We need no additional reduction rules to account for interaction between access points and channel cancellation. Should an endpoint waiting to be matched be cancelled, it is paired as usual, and interaction with its associated buffer raises an exception:

$$\begin{aligned} \lightning a \parallel \mathcal{F}\,[\mathbf{receive}\, b] \parallel z(\{a\}, \{b\}) &\Longrightarrow \lightning a \parallel \mathcal{F}\,[\mathbf{receive}\, b] \parallel z(\varepsilon, \varepsilon) \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon) \\ &\Longrightarrow \lightning a \parallel \mathcal{F}\,[\mathbf{raise}] \parallel \lightning b \parallel z(\varepsilon, \varepsilon) \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon) \end{aligned}$$

We might replace E-Match with the following three rules.

$$(\nu a)(\nu b)(\phi_1 M \parallel \phi_2 N \parallel z(\{a\} \cup \mathcal{X}, \{b\} \cup \mathcal{Y})) \longrightarrow (\nu a)(\nu b)(\phi_1 M \parallel \phi_2 N \parallel z(\mathcal{X}, \mathcal{Y}) \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))$$
$$(\nu a)(\lightning a \parallel z(\{a\} \cup \mathcal{X}, \mathcal{Z}) \longrightarrow z(\mathcal{X}, \mathcal{Y})$$
$$(\nu a)(\lightning a \parallel z(\mathcal{X}, \{a\} \cup \mathcal{Z}) \longrightarrow z(\mathcal{X}, \mathcal{Y})$$

The first matches only non-cancelled endpoints, whereas the second and third clean up cancelled endpoints which are present in the access point. These rules are an optimisation and not required to retain preservation or the weaker notion of progress that holds in the presence of access points. These rules introduce a further non-convergent critical pair:

$$\begin{array}{ccc} & (\nu a)(\nu b)(\mathcal{F}[\textbf{cancel}\, a] \parallel \phi M \parallel z(\{a\} \cup \mathcal{X}, \{b\} \cup \mathcal{Y})) & \\ \swarrow & & \searrow \\ (\nu a)(\nu b)(\mathcal{F}[()] \parallel \lightning a \parallel \phi M \parallel z(\{a\} \cup \mathcal{X}, \{b\} \cup \mathcal{Y}) & & (\nu a)(\nu b)(\mathcal{F}[\textbf{cancel}\, a] \parallel \phi M \parallel a(\mathcal{X}, \mathcal{Y}) \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)) \\ \downarrow & & \downarrow \\ (\nu b)(\mathcal{F}[()] \parallel \phi M \parallel z(\mathcal{X}, \{b\} \cup \mathcal{Y})) & & (\nu a)(\nu b)(\mathcal{F}[()] \parallel \lightning a \parallel \phi M \parallel z(\mathcal{X}, \mathcal{Y}) \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)) \end{array}$$

Metatheory. By decoupling process and channel creation we lose the guarantee that the communication topology is acyclic, and therefore introduce the possibility of deadlock. Preservation continues to hold—in fact, we gain a stronger preservation result since the use of TA-ConnectN allows typeability to be preserved by equivalence.

Theorem 37 (Preservation Modulo Equivalence (EGV with Access Points))**.**
If $\Psi; \Delta \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \Longrightarrow \mathcal{D}$*, then there exist* Ψ', Δ' *such that* $\Psi; \Delta \longrightarrow \Psi'; \Delta'$ *and* $\Psi'; \Delta' \vdash^{\phi} \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$ and preservation by $\equiv$; see Appendix D. □

Alas, the introduction of cyclic topologies and therefore the loss of deadlock-freedom necessarily violates global progress. Nevertheless, a weaker form of progress still holds: if a configuration does not reduce, then it is due to deadlock rather than cancellation.

Let us extend the definition of Ψ to take into account references to access points:

$$\Psi ::= \cdot \mid \Psi, a : S \mid \Psi, z : \mathsf{AP}(S)$$

We first extend the term progress lemma to take into account the additional communication and concurrency constructs:

Lemma 65 (Progress: Terms (EGV with Access Points))**.** *If* $\Psi \vdash M : A$*, then either:*

- M *is a value;*
- *there exists some* N *such that* $M \longrightarrow_M N$*; or*
- *there exist* E, N *such that* M *can be written* $E[N]$*, where* N *is either* **raise** *or a session typing primitive of the form:* **send** $V\, W$*,* **receive** V*,* **close** V*,* **cancel** V*,* **spawn** M*,* **new**$_S$*,* **request** V*, or* **accept** V*.*

Next, we must classify terms which are *waiting* to perform an action on an access point.

Definition 19. *We say that a thread is* waiting to perform an action on an access point z *if M is about to request or accept on z. Formally:*

$$\mathit{waiting}(M, z) \triangleq \exists E.(M = E[\mathbf{request}\, z]) \vee (M = E[\mathbf{accept}\, z])$$

We may now classify the nature of non-reducing configurations in the presence of access points. Due to the possibility of deadlock, the result is necessarily weaker.

Lemma 66 (Open Progress (Access Points))**.** *Suppose $\Psi; \Delta \vdash^{\phi} \mathcal{C}$ and $\mathcal{C} \not\Longrightarrow$. Each thread in $\mathcal{C}$ is either:*

1. *a buffer*
2. *a zapper thread*
3. *an access point*
4. *a thread ϕM such that either M is a value V; $\mathit{ready}(M, a)$ for some channel name a, or $\mathit{waiting}(M, z)$ for some access point name z*

Proof. By induction on the derivation of $\Psi; \Delta \vdash^{\phi} \mathcal{C}$, Lemma 65, and inspection of the reduction rules. Note that $\mathbf{spawn}\, M$ and $\mathbf{new}_S$, like $\mathbf{fork}\, M$, may always reduce. □

As a corollary, we have that the threads in non-reducing configurations are either values, buffers, zapper threads, access points, ready on a name a; or waiting on an access point z. Additionally, if a thread is ready to perform an action on some name a, then the configuration does not contain any zapper thread $\lightning a$.

Corollary 12 (Closed Progress (Access Points))**.** *Suppose $\cdot; \cdot \vdash^{\phi} \mathcal{C}$ and $\mathcal{C} \not\Longrightarrow$. Then each thread in $\mathcal{C}$ is either a value; a buffer; a zapper thread; an access point; ready to perform a communication action on some name a; or waiting on an access point z.*

If $\mathcal{C}$ contains a thread ϕM and $\mathit{ready}(a, M)$ for some name a, then $\mathcal{C}$ contains some buffer $a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})$, and $\mathcal{C}$ does not contain a zapper thread $\lightning b$.

In the presence of access points confluence and termination no longer hold: access points are nondeterministic and can encode higher-order state and hence fixpoints via Landin's knot [135].

Syntax

$$S ::= \cdots \mid \mu t.S \mid t \mid \bar{t}$$
$$M ::= \mathbf{rec}\, f(x).M$$

Equirecursive Session Type Unrolling $\boxed{S = S'}$

$$\mu t.S = S\{\mu t.S/t\}$$

Typing Rules $\boxed{\Gamma \vdash M : A}$

$$\frac{\Gamma, f : A \multimap B, x : A \vdash M : B}{\Gamma \vdash \mathbf{rec}\, f(x).M : A \multimap B}\ \text{T-Rec}$$

Duality $\boxed{\overline{S}}$

$$\overline{\mu t.S} = \mu t.\overline{S\{\bar{t}/t\}} \qquad \bar{\bar{t}} = t$$

Reduction Rule $\boxed{M \longrightarrow_{\mathrm{M}} N}$

$$(\mathbf{rec}\, f(x).M)\, V \longrightarrow M\{\mathbf{rec}\, f(x).M/f, V/x\}$$

Figure 9.9: Recursive Session Types

9.4.3 Recursive Session Types

Thus far, we have only considered finite, terminating protocols. In this section, we show how recursive types may be integrated into the system.

We may want, for example, to extend our two-factor authentication session to allow the user to retry after failing to log in, instead of terminating the session.

We can write the recursive two factor authentication server session type as follows:

$$\begin{aligned}
&\mathsf{RecursiveTwoFactorServer} \triangleq \mu t.\\
&\quad ?(\mathsf{Username}, \mathsf{Password}).\oplus\{\\
&\qquad \mathsf{Authenticated} : \mathsf{ServerBody},\\
&\qquad \mathsf{Challenge} : !\mathsf{ChallengeKey}.?\mathsf{Response}.\\
&\qquad\quad \oplus\{\mathsf{Authenticated} : \mathsf{ServerBody},\\
&\qquad\qquad \mathsf{AccessDenied} : t\},\\
&\qquad \mathsf{AccessDenied} : t\\
&\qquad \mathsf{Quit} : \mathsf{End}\}
\end{aligned}$$

Here, μt binds a recursive type variable t such that it can be used in the remainder of the session type. We replace End with t to allow the protocol to repeat, and add a Quit branch to

allow the server to terminate the session.

Now, let us see the updated server implementation:

$$
\begin{array}{l}
\mathsf{recursiveTwoFactorServer} : \mathsf{RecursiveTwoFactorServer} \multimap \mathbf{1} \\
\mathsf{recursiveTwoFactorServer} \triangleq \mathbf{rec}\, f(s)\,. \\
\quad \mathbf{let}\,((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\, s\, \mathbf{in} \\
\quad \mathbf{if}\, \mathsf{checkDetails}(\mathit{username}, \mathit{password})\, \mathbf{then} \\
\quad\quad \mathbf{let}\, s = \mathbf{select}\, \mathsf{Authenticated}\, s\, \mathbf{in}\, \mathsf{serverBody}(s) \\
\quad \mathbf{else} \\
\quad\quad \mathbf{let}\, s = \mathbf{select}\, \mathsf{AccessDenied}\, s\, \mathbf{in}\, f\, s
\end{array}
$$

We choose not to use the Quit branch, although one could imagine it being selected after the number of login attempts exceeds a given number, in the presence of unrestricted integers. The $\mathbf{rec}\, f(x)\,.\, M$ construct, like with the extended versions of λ_{ch} and λ_{act}, introduces an anonymous recursive function where f may be used in the body of the function. The recursive function takes the session channel endpoint as its argument.

We introduce three additional session type constructs: recursive type variables t, dual recursive type variables $\bar{t}$, and recursive type binders $\mu t\,.S$ which bind a recursive type variable t in session type S. We take an *equi-recursive* view of session types, identifying each session type with its unfolding.

Duality in the presence of recursive types has posed problems in the past [19, 52]. We choose to use the view of recursive session types advocated by Lindley and Morris [133], which stems from the initial algebra semantics of recursion, where recursive type variables may be dualised.

With recursive anonymous functions, it becomes possible to write non-terminating programs. The remaining properties of EGV continue to hold.

9.5 Related Work

Carbone et al. [29] provide the first formal basis for exceptions in a session-typed process calculus. Our approach provides significant simplifications: zapper threads provide a simpler semantics and remove the need for their queue levels, meta-reduction relation, and liveness protocol.

Their calculus includes a *service*, which has two associated processes: a default process, and an exception handling process, as well as a throw construct. Their calculus, like ours, allows an arbitrary nesting of exception handlers, and is asynchronous. A difference is that try-catch blocks manifest themselves in the session type, whereas we take the different view of writing protocols with the assumption that failure is either explicitly encoded in the session via branching and selection, or implementation-dependent and therefore handled

using cancellation. Instead, in our setting, the failed session can be cancelled and a session can be re-established using an access point.

Whereas Carbone et al. use a first-order process calculus, EGV is based on a linear λ-calculus and therefore works in the presence of closures. Additionally, our adoption of channel cancellation processes simplifies the semantics, removing the need for queue levels, and an explicit meta-reduction relation. Due to GV's roots in linear logic, we have a direct method for proving progress, therefore eliminating the need for a liveness protocol. Capecchi et al. [28] extend the work of Carbone et al. [29] to the multiparty setting.

Our work draws on that of Mostrous and Vasconcelos [147], who introduce the idea of explicit cancellation. Our work differs from theirs in several key ways. Their system is a process calculus; ours is a λ-calculus. Their channels are synchronous; ours are asynchronous. Their exception handling construct scopes over a single action; ours scopes over an arbitrary computation.

Caires and Pérez [26] describe a core, logically-inspired process calculus supporting nondeterminism and abortable behaviours encoded via a nondeterminism modality. Processes may either provide or not provide a prescribed behaviour; if a process attempts to consume a behaviour that is not provided, then its linear continuation is safely discarded by propagating the failure of sessions contained within the continuation. Their approach is similar in spirit to our zapper threads. Additionally, they give a core λ-calculus with abortable behaviours and exception handling, and define a type-preserving translation into their core process calculus.

Our approach differs in several important ways. First, our semantics is asynchronous, handling the intricacies involved with cancelling values contained in message queues. Second, we give a direct semantics to EGV, whereas Caires and Pérez rely on a translation into their underlying process calculus. Third, to handle the possibility of disconnection, our calculus allows *any* endpoint to be discarded (including the ability to handle uncaught exceptions), whereas the authors opt for an approach more closely resembling checked exceptions, aided by a monadic presentation.

These works are all theoretical; backed by our theoretical development, our implementation (Chapter 10) integrates session types and exceptions, extending Links.

Multiparty Session Types In previous work, I describe an Erlang implementation [71] of the Multiparty Session Actor framework proposed by Neykova and Yoshida [150] with a limited form of failure recovery; Neykova and Yoshida [151] present a more comprehensive approach, based on refining existing Erlang supervision strategies. Chen et al. [36] introduce a formalism based on multiparty session types [100] that handles partial failures by transforming programs to detect possible failures at a set of statically determined synchronisation points. These approaches rely on a fixed communication topology, using mechanisms such as dependency

graphs or synchronisation points to determine which participants are affected when one participant fails. For binary channels, delegation implies location transparency, thus we must consider dynamic topologies.

In the setting of binary channels, delegation implies location transparency and thus failure detection must be at the level of a *channel* as opposed to a *participant*. However, our treatment of exceptions could lend itself very well to incorporating the 'let it crash' methodology embraced by Erlang: an exception raised by attempting to receive on a channel where the partner endpoint is cancelled, for example, would raise an uncaught exception and cancel all affected channels, allowing the process to be restarted and the exception to propagate.

Adameit et al. [4] describe a synchronous multiparty session calculus to handle *link failures* in distributed systems. They introduce *optional* blocks, inspired by subsessions [58]; progress is maintained by specifying a set of default values to use should the subsession fail.

9.6 Conclusion

In this chapter, we have introduced *Exceptional GV*, a core linear functional language which integrates session types, asynchronous communication, and exception handling, and we have seen that EGV enjoys all of GV's metatheory. Additionally, we have shown extensions such as exception payloads, unrestricted types, access points, and recursive session types, and how they are all orthogonal to exception handling.

In the next chapter, we describe the implementation of distributed tierless web applications in the Links programming language, including an implementation of EGV's exception handling semantics.

Chapter 10

Implementation

10.1 Introduction

In Chapter 9, we introduced Exceptional GV, which provides the theoretical underpinnings for a functional programming language which integrates session types and exception handling. In this chapter, we put this theoretical framework into practice, by describing an extension of the Links tierless web programming language with distributed session-typed communication. In doing so, we provide the first application of session types to web programming, showing that EGV may be implemented via a small translation to effect handlers. We illustrate the implementation through the larger case study of a distributed, web-based chat application.

In §10.2, we give an introduction to the tierless web programming paradigm as pioneered by Links. In §10.2.2, we give an overview of the existing design and implementation of session types in Links. We illustrate the concrete example of two-factor authentication, and give a high-level overview of the FST calculus [135] which underlies the integration of session types and linearity in Links. In §10.4, we showcase the high-level points of our implementation by describing an extended example of a distributed chat server making use of session types. In §10.5, we describe our implementation of distributed session types in Links, and in §10.5.2 we describe our implementation strategy for exception handling in the presence of session types. In §10.6 we discuss related work, and §10.7 concludes.

10.2 Background

10.2.1 Tierless Web Programming

Traditional web programming (as shown in Figure 10.1) involves writing client, server and database code in different languages, and manually mediating between them. Not only must a programmer know multiple languages, but they must also be sure that the data transformations between them are correct.

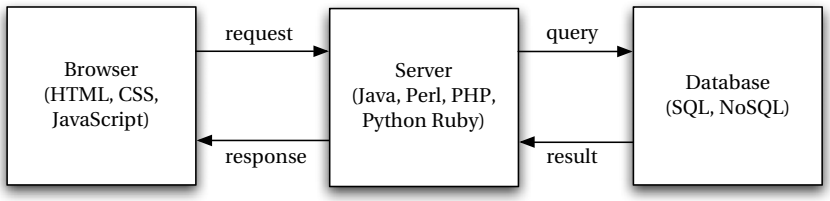


Figure 10.1: Tiers of Web Programming

The Links [46] programming language introduces the notion of *tierless* (also known as *multi-tier* or *cross-tier*) web programming. Links is a statically-typed, ML-inspired, impure functional language which can compile client code to JavaScript, interpret server code, and compile database queries to SQL. Consequently, developers may write web applications in a single, uniform language, without the need to explicitly marshal and unmarshal data. Functions may be annotated with *locations* specifying whether they are located on the client or server, and communication with the server is possible through the use of remote function calls. Links also implements lightweight statically-typed message-passing concurrency in the style of actor-based languages such as Erlang.

On top of basic web functionality, Links has remained an active research language, and now supports a multitude of features such as handlers for algebraic effects [90, 91]; provenance tracking [65]; and query shredding [37].

10.2.2 Session Types in Links

Before proceeding to the distributed extension of Links, we give an overview of the Links implementation of session types as introduced and implemented by Lindley and Morris [135].

Consider again the example of the two-factor authentication server from Chapter 2.

$$
\begin{aligned}
&\mathsf{TwoFactorServer} \triangleq \\
&\quad ?(\mathsf{Username}, \mathsf{Password}).\oplus\{ \\
&\qquad \mathsf{Authenticated} : \mathsf{ServerBody}, \\
&\qquad \mathsf{Challenge} : !\mathsf{ChallengeKey}.?\mathsf{Response}. \\
&\qquad\quad \oplus\{\mathsf{Authenticated} : \mathsf{ServerBody}, \\
&\qquad\qquad \mathsf{AccessDenied} : \mathsf{End}\} \\
&\qquad \mathsf{AccessDenied} : \mathsf{End}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{twoFactorServer} : \mathsf{TwoFactorServer} \multimap \mathbf{1} \\
&\mathsf{twoFactorServer}(s) \triangleq \mathbf{let}\,((\mathit{username}, \mathit{password}), s) = \mathbf{receive}\,s\,\mathbf{in} \\
&\qquad\qquad \mathbf{if}\,\mathsf{checkDetails}(\mathit{username}, \mathit{password})\,\mathbf{then} \\
&\qquad\qquad\quad \mathbf{let}\,s = \mathbf{select}\,\mathsf{Authenticated}\,s\,\mathbf{in}\,\mathsf{serverBody}(s) \\
&\qquad\qquad \mathbf{else} \\
&\qquad\qquad\quad \mathbf{let}\,s = \mathbf{select}\,\mathsf{AccessDenied}\,s\,\mathbf{in}\,\mathbf{close}\,s
\end{aligned}
$$

In Links, this example would be written as follows:

```
typename TwoFactorServer =
  ?(Username, Password).[+|
    Authenticated: ServerBody,
    Challenge: !ChallengeKey.?Response.
      [+| Authenticated: ServerBody,
          AccessDenied: End |+],
    AccessDenied: End |+];

sig twoFactorServer : (TwoFactorServer) ~%~> ()
fun twoFactorServer(s) {
  var ((username, password), s) = receive(s);
  if (checkDetails(username, password)) {
    var s = select Authenticated s;
    serverBody(s);
  } else {
    var s = select AccessDenied s;
    close(s)
  }
}
```

Here, `sig` describes a type signature, in this case denoting a function taking a channel endpoint of type `TwoFactorServer` and returning the unit value. The `~%~>` arrow is Links syntax for a function which may perform arbitrary effects, including recursion. Type aliases are introduced by `typename`, and selection is denoted by `[+| ... |+]`. Although not appearing in the example, offering a choice is denoted by `[&| ... |&]`. Duality is denoted by a tilde (~). The remainder of the syntax is similar to that of EGV. An important difference is that channel endpoints of type `End` are affine, so `close` is a dummy function defined as

```
fun close(s) {
  ()
}
```

Communication mismatch. Links catches session type violations, including linearity violations, statically. As an example, if we were to try to send prior to receiving along the channel:

```
sig twoFactorServer : (TwoFactorServer) ~%~> ()
fun twoFactorServer(s) {
  var s = send("incorrect message", s);
  var ((username, password), s) = receive(s);
  if (checkDetails(username, password)) {
    var s = select Authenticated s;
    serverBody(s);
  } else {
    var s = select AccessDenied s;
    close(s)
  }
}
```

we obtain the following type error:

```
error.links:28: Type error: The function
    `send'
has type
    `(String, !(String).a::Session) ~b~> a::Session'
while the arguments passed to it have types
    `String'
and
    `TwoFactorServer'
and the currently allowed effects are
    `|wild|c'
In expression: send("incorrect message", s).
```

From the error, we can infer that that the session type was expected to be of the form !String.S, whereas it is in fact of type TwoFactorServer.

Erroneously reusing endpoints. Now, let us see how Links can catch linearity violations. The following erroneous code snippet receives along the channel endpoint `s`, binding the continuation to `t`, which is immediately cancelled. Next, `s` is used again to receive along the channel, in contravention of the session type, and the protocol is followed as before.

```
sig twoFactorServer : (TwoFactorServer) ~%~> ()
fun twoFactorServer(s) {
  var ((username, password), t) = receive(s);
  cancel(t);
  var ((username, password), s) = receive(s);
  if (checkDetails(username, password)) {
    var s = select Authenticated s;
    serverBody(s);
  } else {
    var s = select AccessDenied s;
    close(s)
  }
}
```

Links statically detects the linearity violation, and reports the following error.

```
error.links:34: Type error: Variable s has linear type
    `TwoFactorServer'
but is used 2 times.
In expression: fun twoFactorServer(s) {
  var ((username, password), t) = receive(s);
  cancel(t);
  var ((username, password), s) = receive(s);
  if (checkDetails(username, password)) {
    var s = select Authenticated s;
    serverBody(s);
  } else {
    var s = select AccessDenied s;
    close(s)
  }
}
```

Erroneously discarding session endpoints. Finally, we can show that Links ensures that the entire protocol is followed, preventing session endpoints from being dropped prematurely. The following code takes a session endpoint `s` of type `TwoFactorServer` as its parameter, but ignores it and returns the unit value.

```
sig twoFactorServer : (TwoFactorServer) ~%~> ()
fun twoFactorServer(s) {
  ()
}
```

Links statically catches the linearity violation, reporting the following error:

```
error.links:34: Type error: Variable s has linear type
    `TwoFactorServer'
```

```
but is used 0 times.
In expression: fun twoFactorServer(s) {
  ()
}.
```

10.2.3 FST

Integrating session types with a realistic programming language is nontrivial. The Links implementation of session types is underpinned by a calculus, FST [135], which smoothly integrates session types, linearity, unrestricted types, polymorphism, and row types. Session subtyping, as originally described by Gay and Hole [76], is handled through row polymorphism [184]. In this section, we describe the key features of FST by example. A full technical exposition can be found in [135].

Types and Kinds. In FST, types have *kinds* $K(Y, Z)$, such that:

- K is a *primary kind*: Type, Row, or Presence
- Y describes *linearity*: either *linear* ($\circ$) or *unrestricted* ($\bullet$)
- Z describes *restriction*: either *session-typed* (π) or *unconstrained* ($\star$)

We will concentrate on types with kind Type, omitting the primary kind annotation.

Subkinding allows an unrestricted type to be used linearly and a session type to be used as an unconstrained type, but not vice-versa.

Session-typed communication primitives. Let us now look at the types of session-typed communication primitives in FST.

$$\mathsf{send} : \forall \alpha^{\circ,\star}.\forall \beta^{\circ,\pi}.(\alpha \times !\alpha.\beta) \to^{\bullet} \beta$$

We can deconstruct the type piece-by-piece. The send primitive is polymorphic in two type variables, α and β: here, α is the payload type, and β is the session continuation. The type variable α is defined as linear and unconstrained, but by subkinding may also be treated as unrestricted or session-typed. In contrast, β may be either linear or unrestricted, but *must* be a session type. The send function is unrestricted ($\to^{\bullet}$) as we want to be able to use it more than once, and takes a pair of a payload α and output session type $!\alpha.\beta$, returning an endpoint of session type β.

The receive construct is similar:

$$\mathsf{receive} : \forall \alpha^{\circ,\star}.\forall \beta^{\circ,\pi}.?\alpha.\beta \to^{\bullet} (\alpha \times \beta)$$

As before, the primitive is polymorphic in α and β, where α is linear and unconstrained, and β is linear and session-typed. This time, the primitive takes an endpoint of type $?\alpha.\beta$, returning a pair of the payload of type α and the continuation endpoint of type β.

The fork construct is slightly different:

$$\mathsf{fork} : \forall \alpha^{\circ,\pi}.\forall \beta^{\bullet,\star}.(\alpha \to^{\circ} \beta) \to^{\bullet} \overline{\alpha}$$

The fork construct is again polymorphic in type variables α and β. This time, α is a linear session type, and β is a unrestricted, unconstrained type; the function passed to fork must be linear as it contains a linear variable, and β must be unrestricted in order to ensure that the linear session has been fully used. The function returns the dualised type variable $\overline{\alpha}$.

Type and kind system examples. The main typing judgement in FST is $\Delta;\Gamma \vdash M : A$, which can be read as "under kind environment Δ and type environment Γ, M has type A".

Let us now see some illustrative examples of the type system.

As we might expect, we can straightforwardly type values of base type.

$$\Delta;\Gamma \vdash 5 : \mathsf{Int}$$

(where Γ contains no linear variables).

Next, let us see how the kinding environment can be used. Here, we type an unrestricted const function.

$$\alpha :: \mathsf{Type}(\bullet,\star);\Gamma \vdash \Lambda\beta^{\bullet,\star}.\lambda^{\bullet}x^{\alpha}.\lambda^{\bullet}y^{\beta}.x : \forall \beta^{\bullet,\star}.\alpha \to^{\bullet} \beta \to^{\bullet} \alpha$$

(again, where Γ contains no linear variables).

The kinding environment initially contains a mapping stating that type variable α has kind $\mathsf{Type}(\bullet,\star)$. The term consists of a type variable binder for type variable β, ascribing it kind $\mathsf{Type}(\bullet,\star)$, followed by binders for variables x and y of types α and β respectively. Note that each λ binder is annotated with whether the function is linear or unrestricted. The whole term thus has type $\forall \beta^{\bullet,\star}.\alpha \to^{\bullet} \beta \to^{\bullet} \alpha$.

Now, let us see a term which fails to type under a given environment.

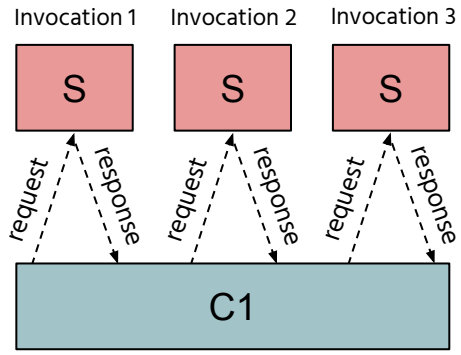
$$\alpha :: \mathsf{Type}(\bullet,\star), \beta :: \mathsf{Type}(\circ,\star);\Gamma \nvdash .\lambda^{\bullet}x^{\alpha}.\lambda^{\bullet}y^{\beta}.x$$

Here, we have that α has kind $\mathsf{Type}(\bullet,\star)$, and β has kind $\mathsf{Type}(\circ,\star)$. The term is not typeable since y has type β which is linear, but does not appear in the body of the function.

10.3 Cross-tier Communication and Concurrency

To better appreciate the challenges of cross-tier communication and concurrency, it is useful to investigate how the design of the Links server and concurrency runtimes have evolved.

Pre-v0.6: CGI and RPC

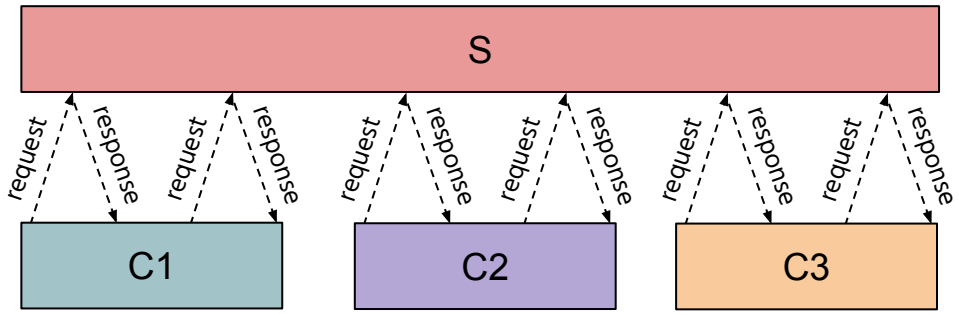


The original version of Links allows cross-tier communication through the use of RPC calls. Functions are annotated with `client` and `server` annotations; the client can call the server by making an AJAX request, and the server can call the client by serialising its continuation in its response to the client. The RPC mechanism used by Links is formalised by Cooper and Wadler [47].

The server in the original version of Links is a CGI script that is invoked upon every request. While this offers a degree of scalability, it does not allow persistent server processes which can communicate with multiple clients.

The original version of Links allows actor-style message passing concurrency on the client only, which is implemented by translating client code to continuation-passing style and implementing a scheduler.

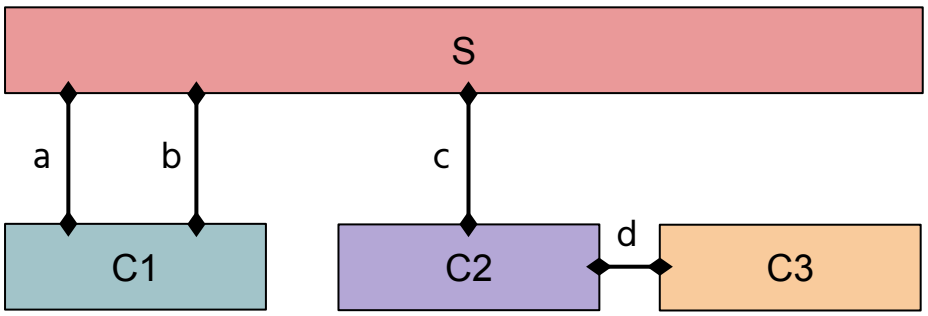
v0.6: Persistent Server, Server-side Concurrency, and Session Types



Version v0.6 of Links was the first to include session types, and was first to introduce an application server model where the web server persists as opposed to being invoked upon every request. Additionally, v0.6 was first to support server-side concurrency.

Whereas both clients and the server could spawn processes and use both actor-style messaging as well as session-typed communication channels, communication was only possible within a concurrency runtime.

v0.7: Distributed Communication and Concurrency



Version v0.7 of Links introduced distributed communication and concurrency, allowing both actor-style and session-typed communication both between a client and the server, but also between two clients. We describe the design and implementation of distributed session-typed concurrency in §10.5.

v0.7.2: Session Types with Exceptions

Version v0.7.2 introduced the exception handling mechanism described in Chapter 9, allowing applications to handle the case where a user goes offline in the middle of a session. We describe the implementation of the exception mechanism in Links in §10.5.2.

The changes in v0.7 and v0.7.2, namely distribution and session types with exceptions, are the contributions described in this chapter.

10.4 Example: A Chat Application

In this section we outline the design and implementation of a web-based chat application in Links making use of distributed session-typed channels. Informally, we write the following specification:

- To initialise, a client must:
  - connect to the chat server; then
  - send a nickname; then
  - receive the current topic and list of nicknames.
- After initialisation the client is connected and can:
  - send a chat message to the room; or
  - change the room's topic; or
  - receive messages from other users; or
  - receive changes of topic from other users.

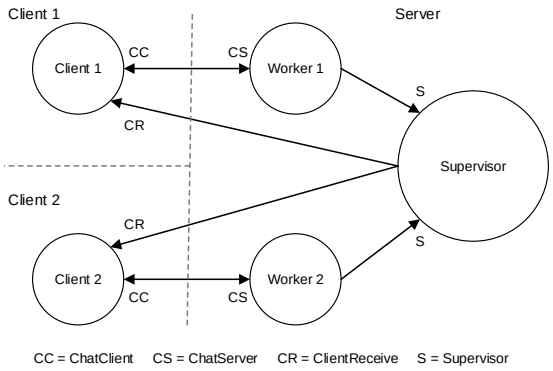
```
typename ChatClient = !Nickname.
  [&| Join: ?(Topic, [Nickname], ClientReceive).ClientSend,
      Nope:End |&];

typename ClientReceive =
  [&| Join     : ?Nickname           .ClientReceive,
      Chat     : ?(Nickname, Message).ClientReceive,
      NewTopic : ?Topic              .ClientReceive,
      Leave    : ?Nickname           .ClientReceive
  |&];

typename ClientSend =
  [+| Chat  : ?Message.ClientSend,
      Topic : ?Topic  .ClientSend |+];

typename ChatServer = ~ChatClient;
typename WorkerSend = ~ClientReceive;
typename WorkerReceive = ~ClientSend;
```

Figure 10.2: Chat Application Session Types

- Clients cannot connect with a nickname that is already in use in the room.
- All participants should be notified whenever a participant joins or leaves the room.

Session Types. We can encode much of the specification more precisely as a session type, as shown in Figure 10.2. The client begins by sending a nickname, and then offers the server a choice of a `Join` message or a `Nope` message. In the former case, the client then receives a triple containing the current topic, a list of existing nicknames, and an endpoint (of type `ClientReceive`) for receiving further updates from the server; and may then continue to send messages to the server as a connected client endpoint (of type `ClientSend`). (Observe the essential use of session delegation.) In the latter case, communication is terminated. The intention is that the server will respond with `Nope` if a client with the supplied nickname is already in the chat room (the details of this check are part of the implementation, not part of the communication protocol).

The `ClientReceive` endpoint allows the client to offer a choice of four different messages: `Join`, `Chat`, `NewTopic`, or `Leave`. In each case the client then receives a payload (depending on the choice, a nickname, pair of nickname and chat message, or topic change) before offering another choice. The `ClientSend` endpoint allows the client to select between two different messages: `Chat` and `NewTopic`. In each case the client subsequently sends a payload (a chat message or a new topic) before selecting another choice. The chat server communicates with the client along endpoints with dual types.



(a) Architecture

```
sig worker : (Nickname, WorkerReceive) ~> ()
fun worker(nick, c) {
  try {
    offer(c) {
      case Chat(c) ->
        var (msg, c) = receive(c);
        chat(nick, msg);
        c
      case NewTopic(c) ->
        var (topic, c) = receive(c);
        newTopic(topic);
        c
    }
  } as (c) in {
    worker(nick, c)
  } otherwise {
    leave(nick)
  }
}
```

(b) Worker Implementation

Figure 10.3: Chat Application Architecture and Worker Implementation

Architecture. Figure 10.3a depicts the architecture of the chat server application. Each client has a process which sends messages over a distributed session channel of type `ClientSend` to its own worker process on the server, which in turn sends internal messages to a supervisor process containing the state of the chat room. In turn, these messages trigger the supervisor process to broadcast a message to all chat clients over a channel of type `~ClientReceive`. As is evident from the figure, the communication topology is cyclic; in order to construct this topology, the code makes essential use of access points.

Disconnection. Figure 10.3b shows the implementation of a worker process which receives messages from a client. The worker takes the nickname of the client, as well as a channel endpoint of type `WorkerReceive` (which is the dual of `ClientSend`). The server offers the client a choice of sending a message (`Chat`), or changing topic (`NewTopic`); in each case, the associated data is received and an appropriate message dispatched to the supervisor process by calling `chat` or `newTopic`. The client may leave the chat room at any time by closing the browser window. All other participants are notified when a participant joins or leaves. When a client closes its connection to the server, all associated endpoints are cancelled. Consequently, an exception will be raised when evaluating the `offer` or `receive` expressions where the user has closed their browser window. To handle disconnection, we wrap the function in an exception handler, which recursively calls `worker` if the interaction is successful, and notifies the server that the user has left via a call to `leave` if an exception is raised.

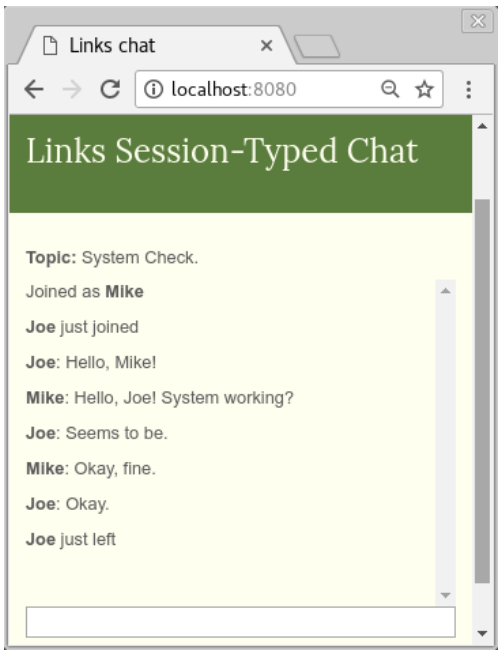


Figure 10.4: Chat server application, handling disconnection

Figure 10.4 shows a screenshot of the chat application running, in particular showing a user leaving the room and the disconnection being handled gracefully.

10.5 Implementation

In this section we describe our extensions to the Links programming language to incorporate the exception handling functionality of EGV as well as extensions to the Links concurrency runtimes to support distribution.

10.5.1 Concurrency

Links provides typed, actor-style concurrency, where processes have a single incoming message queue and can send asynchronous messages. Lindley and Morris [135] extended Links with session-typed channels, using Links' process-based model but replacing actor mailboxes with session-typed channels. We extend their implementation to support distribution and failure handling.

The client relies on continuation-passing style (CPS), trampolining, and co-operative threading. Client code is compiled to CPS, and explicit `yield` instructions are inserted at every function application. When a process has yielded a given number of times, the continuation is pushed to the back of a queue, and the next process is pulled from the front of the queue. While modern browsers are beginning to integrate tail-recursion, and the Links client runtime has been updated to support it, adoption is not yet widespread. Thus, we periodically discard

the call stack using a trampoline. Cooper [45] discusses the Links client concurrency model in depth. The server implements concurrency on top of the OCaml `lwt` library [211], which provides lightweight co-operative threading. At runtime, a channel is represented as a pair of endpoint identifiers:

$$(\texttt{Peer endpoint}, \texttt{Local endpoint})$$

Endpoint identifiers are unique. If a channel (a, b) exists at a given location, then that location should contain a buffer for b.

10.5.2 Exceptions

Effect Handlers. Algebraic effects [175] and their handlers [178] are a modular abstraction for programming with user-defined effects. Exception handlers are in fact a special case of effect handlers. Consequently, we leverage the existing implementation of effect handlers in Links [90, 91]. In Chapter 9 we generalised $\textbf{try} - \textbf{as} - \textbf{in} - \textbf{otherwise}$ to accommodate user defined exceptions. Effect handlers generalise further to support what amounts to *resumable exceptions* in which the handler not only has access to a payload, but also to the delimited continuation (i.e. evaluation context) from the point at which the exception was raised up to the handler, allowing effect handlers to implement arbitrary side-effects; not just exceptions.

Adopting the syntax of Hillerström and Lindley's $\lambda^{\rho}_{\text{eff}}$ calculus [90], we translate exception handling as follows:

$$\begin{aligned}
[\![\textbf{raise}]\!] &= \textbf{do}\,\mathsf{raise} \\
[\![\textbf{try}\,L\,\textbf{as}\,x\,\textbf{in}\,M\,\textbf{otherwise}\,N]\!] &= \textbf{handle}\,[\![L]\!]\,\textbf{with} \\
&\qquad\qquad \textbf{return}\,x \mapsto [\![M]\!] \\
&\qquad\qquad \mathsf{raise}\,r \quad \mapsto \textbf{cancel}\,r; [\![N]\!]
\end{aligned}$$

The introduction form $\textbf{do}\,\mathsf{op}$ invokes an operation op (which may represent raising an exception or some other effect). The elimination form $\textbf{handle}\,M\,\textbf{with}\,H$ runs effect handler H on the computation M. In general an effect handler H consists of a *return clause* of the form $\textbf{return}\,x \mapsto N$, which behaves just like the success continuation $(x\,\textbf{in}\,N)$ of an exception handler, and a collection of *operation clauses*, each of the form $\mathsf{op}\,\vec{p}\,r \mapsto N$. Each clause specifies how to handle each operation analogously to how exception handler clauses specify how to handle each exception, except that as well as binding payload parameters $\vec{p}$, each operation clause also binds a *resumption* parameter r. The resumption r binds a closure that reifies the continuation up to the nearest enclosing effect handler, allowing control to pass back to the program after handling the effect.

In the case of our translation, the raise operation has no payload, and rather than invoking the resumption r we cancel it, assuming the natural extension of cancellation to arbitrary

linear values, whereby all free names in the value are cancelled (r being bound to the current evaluation context reified as a value).

As a preprocessing step, before translating to effect handlers, we insert a dummy exception handler around each forked thread

$$(\!|\mathbf{fork}\, M|\!)^\star = \mathbf{fork}\,(\mathbf{try}\,(\!|M|\!)^\star\,\mathbf{as}\,x\,\mathbf{in}\,()\,\mathbf{otherwise}\,())$$

which has the effect of simulating the E-RaiseChild rule, ensuring that unhandled exceptions are trapped and all endpoints in the context are cancelled if an exception is raised.

As we are targeting *linear* effect handlers, the sharing of linear variables between the success and failure continuations of an exception handler is problematic since there is no reason, *a priori*, to assume that operations should not be handled more than once. The issue can be resolved by restricting the typing rule for **try** in order to disallow any free variables in the continuations:

$$\text{T-TryRestricted}\quad \frac{\Gamma \vdash L : A \quad x : A \vdash M : B \quad \cdot \vdash N : B}{\Gamma \vdash \mathbf{try}\,L\,\mathbf{as}\,x\,\mathbf{in}\,M\,\mathbf{otherwise}\,N : B}$$

This rule may look overly restrictive, but in fact it still allows us to simulate the unrestricted rule via a simple macro translation using a Option type:

$$\begin{aligned}(\!|\mathbf{try}\,L\,\mathbf{as}\,x\,\mathbf{in}\,M\,\mathbf{otherwise}\,N|\!)^\dagger = \mathbf{case}\,\mathbf{try}\,(\!|L|\!)^\dagger\,\mathbf{as}\,x\,\mathbf{in}\,\mathsf{Some}\,x\,\mathbf{otherwise}\,\mathsf{None}\,\mathbf{of}\\ \mathsf{Some}\,x \mapsto (\!|M|\!)^\dagger\\ \mathsf{None} \mapsto (\!|N|\!)^\dagger\end{aligned}$$

Links performs this translation as another preprocessing step.

Raising Exceptions. An exception may be raised either explicitly through an invocation of **raise** (desugared to **do** raise), or through a blocked **receive** call where the partner endpoint has been cancelled. Thus, we know statically where any exceptions may be raised.

In order to support cancellation of closures on the client, we adorn closures with an explicit environment field that can be directly inspected. Currently, Links does not closure convert continuations on the client, so we use a workaround in order to simulate cancelling a resumption (as required by the translation $[\![-]\!]$). When compiling client code, for each occurrence of **do** raise, we compile a function that inspects all affected variables and cancels any affected endpoints in the continuation. For each occurrence of **receive**, we compile a continuation to cancel affected endpoints to be invoked by the runtime system if the receive operation fails.

10.5.3 Distributed Communication

In order to support bidirectional communication between client and server we use WebSockets [67]. A WebSocket connection is established by a client. When a request is made and a web page is generated, each client is assigned a unique identifier, which it uses to establish a WebSocket connection. Any messages the server attempts to send prior to a WebSocket connection being established are buffered and delivered once the connection is established. Once the WebSocket connection is established, a JSON protocol is used to communicate messages such as access point operations, remote session messages, and endpoint cancellation notifications.

Client-to-Client Communication. Due to delegation and access points, it is possible that one client will hold one endpoint of a channel, and another client will hold the other endpoint. In order to provide the illusion of client-to-client communication, we route the communication between the two clients via the server. The server maintains a map

$$\texttt{Endpoint ID} \mapsto \texttt{Location}$$

where `Location` is either `Server` or `Client(ID)`, where `ID` identifies a particular client. The map is updated if a connection is established using `fork` or an access point; an endpoint is sent as part of a message; or a client disconnects.

Handling Disconnection. The server also maintains a map

$$\texttt{Client ID} \mapsto [\texttt{Channel}]$$

associating each client with the publicly-facing channels residing on that client, where `Channel` is a pair of endpoints (a, b) such that b is the endpoint residing on the client. Much like TCP connections, WebSocket connections raise an event when a connection is disconnected. Upon receiving such an event, all channels associated with the client are cancelled, and exceptions are invoked as per the exception handling mechanism described further in Chapter 9 and §10.5.2.

10.5.4 Distributed Exceptions

Our implementation fully supports the exception handling semantics defined by EGV. The concurrency runtime at each location maintains a set of cancelled endpoints.

Cancellation. Suppose endpoint a is connected to peer endpoint b. If a is cancelled, then all endpoints in the queue for a are also cancelled according to the E-Zap rule. If a and b are at the same location, then a is added to the set of cancelled endpoints. If they are at different locations, then a cancellation notification for a is routed to b's location. Zapper threads are

modelled in the implementation by recording sets of cancelled endpoints and propagating cancellation messages.

Failed communications. Again, suppose endpoint a is connected to peer endpoint b. Should a process attempt to read from a when the buffer for a is empty, then the runtime will check to see whether b is in the set of cancelled endpoints. If so, then a is cancelled and an exception is raised in the blocked process; if not, the process is suspended until a message is ready. Should the runtime later add b to the set of cancelled endpoints, then again a is cancelled and an exception raised. These actions implement the E-ReceiveZap rule.

Disconnection. To handle disconnection, the server maintains a map from client IDs to the list of endpoints at the associated client. WebSockets—much like TCP sockets—raise a *closed* event on disconnection. Consequently, when a connection is closed, the runtime looks up the endpoints owned by the terminated client and notifies all other clients containing the peer endpoints.

10.5.5 Distributed delegation.

It is possible to send endpoints as part of a message. Session delegation in the presence of distributed communication has intricacies in ensuring that messages are delivered to the correct participant; our implementation adapts the algorithms described by Hu et al. [106]. More details can be found in Appendix E.

10.6 Related Work

10.6.1 Concurrent Functional Web Programming

Early systems. One of the earliest systems for functional web programming is due to Meijer [139], who implements a Haskell library for interfacing with the Common Gateway Interface (CGI) [187]. Meijer's library handles the intricacies of decoding requests, parsing arguments, and generating a response to the user. The library also provides combinators for generating HTML responses.

Thiemann [201] describes a more sophisticated Haskell library for programming with CGI, called WASH/CGI, which as well as providing the raw CGI functionality of Meijer's library, additionally provides a monadic interface for constructing HTML pages and type-safe web forms. CGI scripts must be re-invoked whenever users submit a web form, however WASH/CGI allows programs to be written as though interaction with the user happens in a single persistent application. This is implemented by storing a log of user input, which is

serialised as a hidden field and resubmitted whenever a user submits a form; the key insight is to only prompt the user for additional input when all data has been consumed from the log.

SMLServer [64] is a webserver written in Standard ML, and designed as a module to be loaded in production web servers such as Apache (and originally AOLServer). SMLServer was amongst the first functional web frameworks to provide direct integration with relational databases. Additionally, SMLServer provides an interface for caching, and allows the use of quotations and antiquotations when generating HTML.

Ur/Web Ur/Web [39] leverages the Ur [38] programming language to provide a tierless model of programming where client, server, and database code are written in a single, statically-typed language. Ur/Web provides simply-typed unidirectional channels from the server to the client. Session typed channels allow more structure to be added to the channel types, specifying a sequence of actions which may involve exchanging values of different types.

Elm Elm [51] is a language for declaratively designing web interfaces, which has received much industry adoption. Originally designed as a form of functional reactive programming forbidding higher-order signals, the Elm Architecture has evolved to become a message-based incarnation of the model-view-controller pattern. Elm's focus is on client-side code, as opposed to tierless programming or distributed, multi-user web applications.

iTasks Plasmeijer et al. [174] introduce *task-oriented programming* (TOP): a task is a unit of work with an observable value describing its progress. TOP is implemented in the iTask system [173], allowing the creation of distributed, multi-user web applications. iTask works at a higher level of abstraction, whereas we focus on the case where communication along a channel may follow complex protocols.

Eliom Eliom [183] is an extension of OCaml which allows tierless web programming. A key feature of Eliom is its ability to explicitly assign *locations* using *section annotations*. Annotations can be added to any expression—for example, a server function with a body annotated as evaluating on the client will force evaluation of an expression on the client. This powerful mechanism of distribution is formalised as a core language, along with compilation to core client and server languages.

Haste.App Haste.App [63] is a Haskell DSL for implementing client-centric web applications. Like Links, Haste.App allows both client and server code to be written in a single language, namely Haskell. The primary mechanism of communication between the client and the server is RPC requests, as opposed to the more liberal session-typed channel model we present in this work. Ekblad [62] extends Haste.App to communicate with multiple different servers, as well

as providing an abstraction for *sandboxing* external code; we leave integration with multiple servers to future work.

Hop.js Hop.js [195] (a JavaScript version of the Hop tierless web programming language [196]) allows web applications to be described in a superset of JavaScript. Hop.js is compiled to a persistent web server and to client JavaScript code; a useful feature of Hop is its *service* mechanism, which allows a server-side computation to seamlessly connect to remote servers. Communication is via AJAX requests as opposed to bidirectional channels.

Phoenix Phoenix [138] is a web framework designed for the Elixir [203] programming language. Elixir is an actor-based concurrent functional programming language which uses the BEAM virtual machine designed for Erlang. A key feature of the Elixir programming language is its use of publish-subscribe channels which allow messages to be sent to- and from clients via WebSockets. Unlike our session-typed channels, channels are untyped.

10.6.2 Distributed Session Types

Hu et al. [106] introduce Session Java (SJ), which allows distributed session-based communication in the Java programming language, making use of the Polyglot framework [158] to statically check session types. Hu et al. are the first to present the challenges of distributed delegation along with distributed algorithms which address those challenges. We adapt their algorithms to web applications. SJ restricts communication to a fixed set of simple types; Links allows arbitrary values to be sent. SJ provides statically scoped exception handling, propagating exceptions to ensure liveness, but this feature is not formalised.

10.7 Conclusion

In this chapter, we have seen how the theoretical ideas from EGV may be applied in practice, in particular providing the first implementation of session types and tierless web programming. Our implementation allows fully-distributed communication across session channels, supplementing the existing RPC mechanism in Links.

The exception handling mechanism introduced in EGV is crucial in allowing a program to handle the case where a user goes offline, as we saw in the larger example of a web-based, session-typed chat server. Additionally, the implementation shows that the ideas from EGV may be implemented, and in turn shows the first development of exception handling and session types that is both formalised and implemented in practice.

Part IV

Conclusion

Chapter 11

Conclusion

Communication-centric programming languages use explicit message passing to co-ordinate between concurrent processes, as opposed to relying on co-ordination through shared memory. In doing so, communication-centric programming languages avoid many of the pitfalls of shared-memory approaches by avoiding the non-compositionality of lock mechanisms.

Static type systems provide a lightweight verification mechanism which catches errors, such as passing an unsupported argument to a function, before a program can compile.

11.1 Research Challenges Revisited

In Chapter 1, we posed two research challenges. We summarise the work in the thesis by revisiting the questions:

What is the relationship between typed channel- and actor-based programming, and why have typed actor mailboxes seen limited uptake?
Channel-based languages provide anonymous processes which communicate using shared names known as *channels*, whereas actor-based languages are based on the actor model and provide *named* processes which communicate point-to-point using local message queues known as mailboxes.

To better understand the intricacies of typed mailboxes and the relation of typed channels and actors, we defined two minimal concurrent λ-calculi: λ_{ch}, in which an anonymous process sends messages along typed channels, and λ_{act}, in which an actor sends a message directly to the mailbox of another actor.

To make the relationship between the two models precise, we showed type- and semantics-preserving translations between λ_{ch} and λ_{act}, in particular noting that the translation from λ_{act} into λ_{ch} was simple but global (in the sense of Felleisen [66]), whereas more surprisingly the translation from λ_{act} into λ_{ch} was more involved but local. The translation from channels

into actors is complicated by the *type pollution* problem described by He et al. [86], where the type of a mailbox must reflect the types of all messages sent by other actors, in turn requiring every actor to expose every message it may be sent in order for communication partners to determine the correct sum or variant injection to use when sending a message. Naïvely, type pollution leads to a complete loss of modularity. In the setting of the translation between λ_{ch} and λ_{act}, it may be addressed by either ensuring that each channel in the system has the same type, or by introducing synchronisation primitives.

Finally, we showed that λ_{act} can encode the more involved *selective receive* construct as found in Erlang [11], and in doing so, we gave a formal grounding to the 'stashing' implementation strategy proposed by Haller [83].

How can session types be adapted to support exceptions in a functional language where communication is asynchronous?
To answer this question, we investigated core session-typed functional languages, providing the basis for tierless web programming with session types. The web-based setting made it crucial to provide a formal account of failure handling and the integration of session types and exceptions.

We began by reprising a core, synchronous version of the GV session-typed linear functional language, Synchronous GV (SGV). Next, we introduced Asynchronous GV (AGV), showing how SGV could be extended to allow buffered, asynchronous communication, while still preserving all of SGV's strong metatheory. Modelling asynchrony is important when considering distributed communication.

We then introduced Exceptional GV (EGV), a session-typed linear λ-calculus which extends AGV with support for the ability to explicitly *cancel* an endpoint, following previous work by Mostrous and Vasconcelos [147]. In spite of the ability to discard endpoints, EGV retains global progress by raising exceptions when a communication cannot succeed, and propagating cancellation whenever an endpoint becomes unreachable due to evaluation of a continuation being aborted due to an exception.

Finally, we showed how the ideas from EGV can be implemented in practice, by extending the Links tierless web programming language. In doing so, we provided the first application of session types to web programming, where disconnection (for example, a client closing their web browser), can be handled gracefully. Our implementation makes use of effect handlers, and paves the way for future study of linear effect handlers.

11.2 Future Work

The field of typed, concurrent functional programming is ripe for further study. Here, we describe some interesting future directions.

Relating Actor Variations. In Chapter 5, we related channel- and actor-based programming languages by relating concurrent λ-calculi. The actor-based calculus, λ_{act}, is *process-based* according to the taxonomy defined by De Koster et al. [56]. It would be interesting to formally relate different actor incarnations, for example active objects, by distilling each to a minimal core calculus and showing translations between them.

Exceptional CP. The **cancel** construct in EGV explicitly discards an endpoint, and an exception discards the free names contained in the aborted continuation of a process. It would be interesting to relate a synchronous version of EGV to an analogous extension of CP.

Hypersequents. Recent work by Kokke et al. [122, 123], building on work by Montesi and Peressotti [146] and Carbone et al. [32], addresses the syntactic restrictions present in CP by considering *hypersequents* to register parallelism in linear logic typing judgements. Hypersequents generalise from a single typing environment to multiple typing environments. Consequently, the authors are able to relax the syntactic restrictions required by CP: as an example, the combined $(\nu x)(P \mid Q)$ construct can be expressed as two separate constructs $(\nu x)P$ and $P \mid Q$, with the resulting system enjoying the same strong metatheory as CP.

GV does not suffer from the same severe syntactic restrictions as CP. Nevertheless, the mechanism in the configuration typing system to ensure acyclicity, namely the $S^\sharp$ type in tandem with the T-Connect$_i$ rules, violates preservation of typing under configuration equivalence. While this does not matter for reduction, the issue stems from registering parallelism syntactically. Early exploratory work has shown that the use of hypersequents solves this issue and thus simplifies the metatheory. It would be interesting to fully define GV using hypersequents, and to show type- and semantics-preserving translations to- and from Hypersequent CP.

Multiparty GV. In this thesis, we have considered *binary* session types in concurrent linear λ-calculi. *Multiparty* session types, as described by Honda et al. [100], generalise binary session types to encode interactions between multiple participants in a system.

Multiparty session types are typically described in the context of process calculi, and unlike binary session types, have not been explored in the functional setting. A logically-grounded linear functional language with multiparty session types would give a clean design for integrating multiparty session types into a functional languages, and would give a firm theoretical grounding to extensions such as multiparty access points. Additionally, a logically-grounded calculus would provide global progress by design, without requiring a separate interaction typing system as used in the work of Coppo et al. [49].

A promising initial direction would be to investigate a calculus into which it is possible to translate the MCP calculus described by Carbone et al. [30]. Early exploratory work has

shown that again, hypersequents prove useful; in contrast, generalising the $S^\sharp$ type to the multiparty setting proves to be an exercise in masochism.

Linear Effect Handlers. In Chapter 10, we showed how EGV's exception handling mechanism could be encoded using handlers for algebraic effects. Hillerström [89] discusses an implementation of the Links actor-style message passing concurrency model using only handlers for algebraic effects, with communication and concurrency implemented using handlers without built-in concurrency primitives.

It would be interesting to encode a version of EGV in a language such as $\lambda^{\rho}_{\text{eff}}$, to formally show how the integration of session-typed communication and exception handling may be encoded using linear effect handlers.

Choice and Session Types. In GV, and indeed other session calculi, the **receive** construct receives a value from a single channel. In languages such as Concurrent ML or Go, for example, it is possible to wait on multiple channels, and synchronise on the first channel which is ready with a message. This notion of *choice* stems from work on process calculi, in particular CSP [92].

Choice has not been well-studied in the static, session-typed setting: the closest proposal considers event-driven session-types [107] and relies on session set types and a session typecase primitive. An interesting direction of work would be integrating guarded choice and fully static session types. This would likely require more expressive type systems such as dependent types.

Bibliography

[1] Samson Abramsky. Proofs as processes. *Theor. Comput. Sci.*, 135(1):5–9, 1994. doi: 10.1016/0304-3975(94)00103-0.

[2] Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. Specification structures and propositions-as-types for concurrency. In *Banff Higher Order Workshop*, volume 1043 of *Lecture Notes in Computer Science*, pages 5–40. Springer, 1995.

[3] Actors and Hopac. `https://www.github.com/Hopac/Hopac/blob/master/Docs/Actors.md`, 2016.

[4] Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In *FORTE*, volume 10321 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2017.

[5] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[6] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[7] Akka Typed. `http://doc.akka.io/docs/akka/current/scala/typed.html`, 2016.

[8] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Testing of Concurrent and Imperative Software Using CLP. In *PPDP*, pages 1–8. ACM, 2016. ISBN 978-1-4503-4148-6.

[9] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 1015–1022. ACM, 2009. doi: 10.1145/1639950.1640073.

[10] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[11] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[12] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.

[13] Robert Atkey, Sam Lindley, and J. Garrett Morris. *Conflation confers concurrency*, volume 9600 of *Lecture Notes in Computer Science*, pages 32–55. Springer, 4 2016. ISBN 9783319309354. doi: 10.1007/978-3-319-30936-1_2.

[14] Andrew Barber. *Dual Intuitionistic Linear Logic*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1996.

[15] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103. North Holland, 1984.

[16] Gianluigi Bellin and Philip J. Scott. On the pi-calculus and linear logic. *Theor. Comput. Sci.*, 135(1):11–65, 1994. doi: 10.1016/0304-3975(94)00104-9. URL `https://doi.org/10.1016/0304-3975(94)00104-9`.

[17] Nick Benton and Andrew Kennedy. Exceptional Syntax. *Journal of Functional Programming*, 11(4):395–410, 2001.

[18] Jan A. Bergstra and Jan Willem Klop. Process theory based on bisimulation semantics. In *REX Workshop*, volume 354 of *Lecture Notes in Computer Science*, pages 50–122. Springer, 1988.

[19] Giovanni Bernardi, Ornela Dardha, Simon J. Gay, and Dimitrios Kouzapas. On duality relations for session types. In *TGC*, pages 51–66, 2014. doi: 10.1007/978-3-662-45917-1_4. URL `https://doi.org/10.1007/978-3-662-45917-1_4`.

[20] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *PACMPL*, 2(POPL):5:1–5:29, 2018. doi: 10.1145/3158093. URL `http://doi.acm.org/10.1145/3158093`.

[21] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, 1992. doi: 10.1016/0304-3975(92)90185-I. URL `https://doi.org/10.1016/0304-3975(92)90185-I`.

[22] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2008. doi: 10.1007/978-3-540-85361-9_33.

[23] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *FORTE*, pages 50–65. Springer, 2013.

[24] Gérard Boudol. Asynchrony and the Pi-Calculus. Research Report RR-1702, INRIA, 1992.

[25] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30 (2):323–342, 1983. doi: 10.1145/322374.322380. URL `http://doi.acm.org/10.1145/322374.322380`.

[26] Luís Caires and Jorge A Pérez. Linearity, control effects, and behavioral types. In *ESOP*, pages 229–259. Springer, 2017.

[27] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 10, pages 222–236. Springer, 2010.

[28] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):156–205, 2016.

[29] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured interactional exceptions in session types. In *CONCUR*, pages 402–417. Springer, 2008.

[30] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[31] Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Inf.*, 54(3):243–269, 2017. doi: 10.1007/s00236-016-0285-y.

[32] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distributed Computing*, 31(1):51–67, 2018. doi: 10.1007/s00446-017-0295-1.

[33] Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang/OTP*. O'Reilly Media, Inc., 2016.

[34] Avik Chaudhuri. A Concurrent ML library in Concurrent Haskell. In *ICFP*, pages 269–280, New York, NY, USA, 2009. ACM.

[35] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 11, pages 25–45. Springer, 2011.

[36] Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. A type theory for robust failure handling in distributed systems. In *FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2016.

[37] James Cheney, Sam Lindley, and Philip Wadler. Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD Conference*, pages 1027–1038. ACM, 2014.

[38] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *PLDI*, pages 122–133, New York, NY, USA, 2010. ACM.

[39] Adam Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, volume 50, pages 153–165. ACM, 2015.

[40] Søren Christensen, Hans Hüttel, and Colin Stirling. Bisimulation equivalence is decidable for all context-free processes. *Inf. Comput.*, 121(2):143–148, 1995.

[41] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing*, 10(4):361–380, Apr 1998. ISSN 1433-299X. doi: 10.1007/s001650050022.

[42] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, 2001. doi: 10.1007/3-540-44577-3_12. URL `https://doi.org/10.1007/3-540-44577-3_12`.

[43] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, pages 1–12, 2015. doi: 10.1145/2824815.2824816. URL `http://doi.acm.org/10.1145/2824815.2824816`.

[44] William D Clinger. *Foundations of actor semantics*. PhD thesis, Massachusetts Institute of Technology, 1981.

[45] Ezra Cooper. *Programming Language Features for Web Application Development*. PhD thesis, University of Edinburgh, 2009.

[46] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO*, pages 266–296. Springer, 2007.

[47] Ezra EK Cooper and Philip Wadler. The RPC calculus. In *PPDP*, pages 231–242. ACM, 2009.

[48] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous session types and progress for object oriented languages. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2007. doi: 10.1007/978-3-540-72952-5_1.

[49] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016. doi: 10.1017/S0960129514000188. URL `https://doi.org/10.1017/S0960129514000188`.

[50] Silvia Crafa and Luca Padovani. The chemical approach to typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 39(3):13:1–13:45, 2017.

[51] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *PLDI*, pages 411–422. ACM, 2013.

[52] Ornela Dardha. Recursive session types revisited. In *BEAT*, pages 27–34, 2014. doi: 10.4204/EPTCS.162.4.

[53] Ornela Dardha and Simon J. Gay. A new linear logic for deadlock-free session-typed processes. In *FoSSaCS*, pages 91–109, 2018. doi: 10.1007/978-3-319-89366-2_5.

[54] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017.

[55] Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330. Springer, 2007.

[56] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: A taxonomy of actor models and their key properties. In *AGERE*. ACM, 2016.

[57] Ugo de'Liguoro and Luca Padovani. Mailbox types for unordered interactions. In *ECOOP*, volume 109 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[58] Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, pages 272–286, 2012. doi: 10.1007/978-3-642-32940-1_20.

[59] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, 46(3):197–225, 2015.

[60] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 12, pages 194–213. Springer, 2012.

[61] Alan A.A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 1st edition, 2015. ISBN 0134190440, 9780134190440.

[62] Anton Ekblad. A meta-EDSL for distributed web applications. In *Haskell Symposium*, pages 75–85. ACM, 2017.

[63] Anton Ekblad and Koen Claessen. A seamless, client-centric programming model for type safe web applications. In *Haskell Symposium*. ACM, 2014.

[64] Martin Elsman and Niels Hallenberg. Web programming with SMLserver. In *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 74–91. Springer, 2003.

[65] Stefan Fehrenbach and James Cheney. Language-integrated provenance. *Sci. Comput. Program.*, 155:103–145, 2018. doi: 10.1016/j.scico.2017.08.009.

[66] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1-3):35–75, 1991.

[67] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. URL `http://www.rfc-editor.org/rfc/rfc6455.txt`.

[68] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, 1993.

[69] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *POPL*, pages 372–385, 1996.

[70] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join calculus. *J. Log. Algebr. Program.*, 57(1-2):23–69, 2003. doi: 10.1016/S1567-8326(03)00040-7.

[71] Simon Fowler. An Erlang implementation of multiparty session actors. In *ICE*, volume 223 of *EPTCS*, pages 36–50, 2016.

[72] Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In Peter Müller, editor, *ECOOP*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:28, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi: 10.4230/LIPIcs.ECOOP.2017.11.

[73] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, Jan 2019. doi: 10.1145/3290341.

[74] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(1):27–45, 2009.

[75] Lars-Åke Fredlund. *A framework for reasoning about Erlang code*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2001.

[76] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2):191–225, 2005.

[77] Simon J Gay and Vasco T Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[78] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, pages 28–38. ACM, 1986.

[79] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4. URL `https://doi.org/10.1016/0304-3975(87)90045-4`.

[80] Peter R Gluck and Gerard J Holzmann. Using SPIN model checking for flight software verification. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 1, pages 1–1. IEEE, 2002.

[81] Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[82] Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg. The formal specification language mCRL2. In *MMOSS*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[83] Philipp Haller. On the integration of the actor model in mainstream technologies: the Scala perspective. In *AGERE*, pages 1–6. ACM, 2012.

[84] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[85] Paul Harvey. *A linguistic approach to concurrent, distributed, and adaptive programming across heterogeneous platforms*. PhD thesis, University of Glasgow, 2015.

[86] Jiansen He, Philip Wadler, and Philip W. Trinder. Typecasting actors: from Akka to Takka. In *Proceedings of the Fifth Annual Scala Workshop, SCALA@ECOOP 2014, Uppsala, Sweden, July 28-29, 2014*, pages 23–33, 2014. doi: 10.1145/2637647.2637651. URL `http://doi.acm.org/10.1145/2637647.2637651`.

[87] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[88] Rich Hickey. Clojure core.async Channels. `http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html`, 2013.

[89] Daniel Hillerström. Compilation of effect handlers and their applications in concurrency. Master's thesis, University of Edinburgh, 2016.

[90] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *TyDe@ICFP*, pages 15–27. ACM, 2016.

[91] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. Continuation passing style for effect handlers. In *FSCD*, volume 84 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[92] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21 (8):666–677, August 1978. ISSN 0001-0782.

[93] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997. doi: 10.1109/32.588521. URL `https://doi.org/10.1109/32.588521`.

[94] Kohei Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523. Springer, 1993.

[95] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *ECOOP*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 1991. doi: 10.1007/BFb0057019.

[96] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theor. Comput. Sci.*, 151(2):437–486, 1995.

[97] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, pages 122–138. Springer, 1998.

[98] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284, 2008. doi: 10.1145/1328438.1328472.

[99] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, pages 55–75, 2011.

[100] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM (JACM)*, 63(1):9, 2016.

[101] Hopac. `http://www.github.com/Hopac/hopac`, 2016.

[102] How are Akka actors different from Go channels? `https://www.quora.com/How-are-Akka-actors-different-from-Go-channels`, 2013.

[103] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[104] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In Perdita Stevens and Andrzej Wąsowski, editors, *FASE*, pages 401–418, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-49665-7.

[105] Raymond Hu and Nobuko Yoshida. Explicit Connection Actions in Multiparty Session Types. In *FASE*, pages 116–133, 2017. doi: 10.1007/978-3-662-54494-5_7.

[106] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *ECOOP*, pages 516–541. Springer, 2008.

[107] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *ECOOP*, volume 10, pages 329–353. Springer, 2010.

[108] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[109] Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *PACMPL*, 1(ICFP):38:1–38:28, 2017.

[110] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. Session-ocaml: A session-based library with polarities and lenses. In *COORDINATION*, pages 99–118, 2017. doi: 10.1007/978-3-319-59746-1_6.

[111] Shams M Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *AGERE*, pages 67–80. ACM, 2014.

[112] Is Scala's actors similar to Go's coroutines? `http://stackoverflow.com/questions/22621514/is-scalas-actors-similar-to-gos-coroutines`, 2014.

[113] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *WGP*, pages 13–22. ACM, 2015.

[114] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, pages 142–164. Springer, 2010.

[115] Simon Peyton Jones. Beautiful concurrency. *Beautiful Code: Leading Programmers Explain How They Think*, pages 385–406, 2007.

[116] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *LICS*, pages 128–139, 1997. doi: 10.1109/LICS.1997.614941.

[117] Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002. doi: 10.1006/inco.2002.3171.

[118] Naoki Kobayashi. Type systems for concurrent programs. In *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers*, pages 439–453. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[119] Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, pages 233–247, 2006. doi: 10.1007/11817949_16.

[120] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371, 1996. doi: 10.1145/237721.237804.

[121] Naoki Kobayashi, Shin Saito, and Eijiro Sumii. An implicitly-typed deadlock-free process calculus. In *CONCUR*, pages 489–503, 2000. doi: 10.1007/3-540-44618-4_35.

[122] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Taking linear logic apart. Extended abstract, presented at Linearity/TLLA, 2018.

[123] Wen Kokke, Fabrizio Montesi, and Marco Peressotti. Better late than never: A fully abstract semantics for classical processes. *Proc. ACM Program. Lang.*, 3(POPL):24:1–24:29, Jan 2019. doi: 10.1145/3290337.

[124] Dimitrios Kouzapas, Nobuko Yoshida, and Kohei Honda. On asynchronous session semantics. In Roberto Bruni and Jürgen Dingel, editors, *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2011.

[125] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.*, 155:52–75, 2018. doi: 10.1016/j.scico.2017.10.006. URL `https://doi.org/10.1016/j.scico.2017.10.006`.

[126] Pierre Krafft. Singly typed actors in Agda. Master's thesis, Chalmers University of Technology, 2018.

[127] Yves Lafont. The linear abstract machine. *Theor. Comput. Sci.*, 59:157–180, 1988.

[128] R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[129] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system release 3.11. *Documentation and user's manual. INRIA*, 2008.

[130] Jean-Jacques Lévy and Luc Maranget. Explicit substitutions and programming languages. In *FSTTCS*, volume 1738 of *Lecture Notes in Computer Science*, pages 181–200. Springer, 1999.

[131] Paul B. Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

[132] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015.

[133] Sam Lindley and J Garrett Morris. Talking bananas: structural recursion for session types. In *ICFP*, pages 434–447. ACM, 2016.

[134] Sam Lindley and J Garrett Morris. Embedding session types in Haskell. In *Haskell Symposium*, pages 133–145. ACM, 2016.

[135] Sam Lindley and J Garrett Morris. Lightweight functional session types. In *Behavioural Types: from Theory to Tools*, pages 265–286. River Publishers, 2017.

[136] Sam Lindley and J Garrett Morris. Lightweight functional session types (extended version), 2017. URL `http://homepages.inf.ed.ac.uk/slindley/papers/fst-extended.pdf`.

[137] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *TLDI*, pages 77–88. ACM, 2010.

[138] Chris Mccord, Bruce Tate, and Jose Valim. *Programming Phoenix 1.4*. O'Reilly Media, Inc., 2019.

[139] Erik Meijer. Server side web scripting in Haskell. *J. Funct. Program.*, 10(1):1–18, 2000.

[140] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[141] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. Concurrency among strangers. In *TGC*, pages 195–229, 2005. doi: 10.1007/11580850_12.

[142] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. ISBN 978-0-13-115007-2.

[143] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2): 119–141, 1992.

[144] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1st edition, June 1999.

[145] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990. ISBN 978-0-262-63132-7.

[146] Fabrizio Montesi and Marco Peressotti. Classical transitions. *CoRR*, abs/1803.01049, 2018. URL `http://arxiv.org/abs/1803.01049`.

[147] Dimitris Mostrous and Vasco Thudichum Vasconcelos. Affine sessions. In *COORDINATION*, pages 115–130. Springer, 2014.

[148] Maurice Naftalin. *Mastering Lambdas: Java Programming in a Multicore World*. McGraw-Hill Education Group, 2014.

[149] Matthias Neubauer and Peter Thiemann. An implementation of session types. In *PADL*, volume 4, pages 56–70. Springer, 2004.

[150] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1), 2017.

[151] Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *CC*, pages 98–108. ACM, 2017.

[152] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: local verification of global protocols. In *RV*, pages 358–363. Springer, 2013.

[153] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *CC*, pages 128–138, 2018. doi: 10.1145/3178372.3179495.

[154] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: Safe parallel programming with message optimisation. *TOOLS*, pages 202–218, 2012.

[155] Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by default - safe MPI code generation based on session types. In *CC*, pages 212–232, 2015. doi: 10.1007/978-3-662-46663-6_11.

[156] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, 2006.

[157] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Institute of Technology, 2007.

[158] Nathaniel Nystrom, Michael Clarkson, and Andrew Myers. Polyglot: An extensible compiler framework for Java. In *CC*, pages 138–152. Springer, 2003.

[159] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

[160] Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *POPL*, volume 51, pages 568–581. ACM, 2016.

[161] Dominic Orchard and Nobuko Yoshida. Session types with linearity in Haskell. *Behavioural Types: from Theory to Tools*, page 219, 2017.

[162] Luca Padovani. From lock freedom to progress using session types. In *PLACES*, pages 3–19, 2013. doi: 10.4204/EPTCS.137.2.

[163] Luca Padovani. Deadlock and lock freedom in the linear π-calculus. In *CSL-LICS*, pages 72:1–72:10, 2014. doi: 10.1145/2603088.2603116.

[164] Luca Padovani. A simple library implementation of binary sessions. *Journal of Functional Programming*, 27:e4, 2017.

[165] Luca Padovani. Context-free session type inference. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 804–830. Springer, 2017.

[166] Luca Padovani. Deadlock-free typestate-oriented programming. *Programming Journal*, 2(3):15, 2018. doi: 10.22152/programming-journal.org/2018/2/15.

[167] Luca Padovani and Luca Novara. Types for deadlock-free higher-order programs. In Susanne Graf and Mahesh Viswanathan, editors, *FORTE*, pages 3–18. Springer International Publishing, 2015.

[168] Catuscia Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. doi: 10.1017/S0960129503004043.

[169] Jennifer Paykin, Antal Spector-Zabusky, and Kenneth Foner. `choose` your own derivative. In *TyDe*, pages 58–59. ACM, 2016.

[170] Jorge A Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In *ESOP*, pages 539–558. Springer, 2012.

[171] Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: making structured data first-class citizens in F#. In *PLDI*, pages 477–490, 2016. doi: 10.1145/2908080.2908115.

[172] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[173] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *ICFP*, pages 141–152, New York, NY, USA, 2007. ACM.

[174] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-oriented programming in a pure functional language. In *PPDP*, pages 195–206. ACM, 2012.

[175] Gordon Plotkin and John Power. Adequacy for algebraic effects. In *FoSSaCS*, pages 1–24. Springer, 2001.

[176] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *ESOP*, pages 80–94. Springer, 2009.

[177] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5 (3):223–255, 1977. doi: 10.1016/0304-3975(77)90044-5.

[178] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[179] Jeff Polakow. Embedding a full linear lambda calculus in Haskell. *Haskell Symposium*, 50(12):177–188, 2016.

[180] Swarn Priya. λir: A language with intensional receive. Master's thesis, Iowa State University, 2017.

[181] Proto.Actor. `http://www.proto.actor`, 2016.

[182] Riccardo Pucella and Jesse A Tov. Haskell session types with (almost) no class. In *Haskell Symposium*, volume 44, pages 25–36. ACM, 2008.

[183] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In *APLAS*, pages 377–397. Springer, 2016.

[184] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming*, pages 67–95. MIT Press, Cambridge, MA, 1994.

[185] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. URL `http://www.oreilly.com/webops-perf/free/kubernetes.csp`.

[186] J.H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[187] David Robinson and Ken A. L. Coar. The common gateway interface (CGI) version 1.1. *RFC*, 3875:1–36, 2004.

[188] Matthew Sackman and Susan Eisenbach. Session types in Haskell: Updating message passing for the 21st century. 2008.

[189] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2003.

[190] Alceste Scalas and Nobuko Yoshida. Lightweight session programming in Scala. In *ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

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

[192] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[193] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP*, pages 275–299, 2010. doi: 10.1007/978-3-642-14107-2_13.

[194] Bastian Schlich. Model checking of software for microcontrollers. *ACM Trans. Embedded Comput. Syst.*, 9(4):36:1–36:27, 2010. doi: 10.1145/1721695.1721702. URL `http://doi.acm.org/10.1145/1721695.1721702`.

[195] Manuel Serrano and Vincent Prunet. A glimpse of Hopjs. In *ICFP*, pages 180–192. ACM, 2016. ISBN 978-1-4503-4219-3.

[196] Nauel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *DLS*, pages 975–985. ACM, 2006.

[197] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming*, pages 81–92, 2006.

[198] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1):157–171, 1986.

[199] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[200] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do Scala developers mix the actor model with other concurrency models? In *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer, 2013.

[201] Peter Thiemann. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In *PADL*, volume 2257 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2002.

[202] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *ICFP*, pages 462–475. ACM, 2016.

[203] Dave Thomas. *Programming Elixir: Functional, Concurrent, Pragmatic, Fun*. Pragmatic Bookshelf, 1st edition, 2014. ISBN 1937785580, 9781937785581.

[204] Bernardo Toninho and Nobuko Yoshida. Interconnectability of session-based logical processes. *ACM Trans. Program. Lang. Syst.*, 40(4):17:1–17:42, 2018.

[205] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, pages 350–369, 2013. doi: 10.1007/978-3-642-37036-6_20.

[206] Typed Actors. `https://github.com/knutwalker/typed-actors`, 2016.

[207] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

[208] Vasco Vasconcelos, António Ravara, and Simon Gay. Session types for functional multithreading. In *CONCUR*, pages 497–511. Springer, 2004.

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

[210] Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2): 64–87, 2006. doi: 10.1016/j.tcs.2006.06.028. URL `https://doi.org/10.1016/j.tcs.2006.06.028`.

[211] Jérôme Vouillon. Lwt: a cooperative thread library. In *ML*, pages 3–12. ACM, 2008.

[212] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.

[213] Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3): 384–418, 2014.

[214] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015. doi: 10.1145/2699407. URL `http://doi.acm.org/10.1145/2699407`.

[215] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009.

[216] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[217] Derek Wyatt. *Akka Concurrency*. Artima Incorporation, USA, 2013. ISBN 0981531660, 9780981531663.

[218] Nobuko Yoshida and Vasco T Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electronic Notes in Theoretical Computer Science*, 171(4):73–93, 2007.

[219] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010.

[220] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble protocol language. In *TGC*, pages 22–41, 2013. doi: 10.1007/978-3-319-05119-2_3.

Appendix A

Proofs for Chapter 3 (Synchronous GV)

A.1 Preservation

A.1.1 Reduction

Theorem 1 *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then there exists some* $\Gamma \longrightarrow^{?} \Gamma'$ *such that* $\Gamma' \vdash^{\phi} \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$, making use of Lemmas 1–5. Where there is a choice of flags, we make the decision to prove the case where the first flag is $\bullet$ and the second is $\circ$; the proofs for other cases are similar.

Case E-Fork

$$\mathcal{F}[\mathsf{fork}\,\lambda x.M] \longrightarrow (\nu a)(\mathcal{F}[a] \parallel \circ M\{a/x\}) \quad (a \text{ is fresh})$$

We let $\mathcal{F}[\mathsf{fork}\,\lambda x.M] = \bullet E[\mathsf{fork}\,\lambda x.M]$. The case where $\mathcal{F}[\mathsf{fork}\,\lambda x.M] = \circ E[\mathsf{fork}\,\lambda x.M]$ is similar (using T-Main instead of T-Thread).

Assumption:

$$\frac{\Gamma_1, \Gamma_2 \vdash E[\mathsf{fork}\,\lambda x.M] : A}{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} \bullet E[\mathsf{fork}\,\lambda x.M]}$$

By Lemma 2:

$$\frac{\dfrac{\Gamma_2, x : S \vdash M : \mathsf{End}_!}{\Gamma_2 \vdash \lambda x.M : S \multimap \mathsf{End}_!}}{\Gamma_2 \vdash \mathsf{fork}\,\lambda x.M : \overline{S}}$$

By Lemma 3, $\Gamma_1, a : \overline{S} \vdash E[a] : A$.

By Lemma 1, $\Gamma_2, a : S \vdash M\{a/x\} : \mathsf{End}_!$.

Recomposing:

$$\dfrac{\dfrac{\Gamma_1; a : \overline{S} \vdash E[a] : A}{\Gamma_1; a : \overline{S}; \cdot \vdash^\bullet E[a]} \quad \dfrac{\Gamma_2, a : S \vdash M\{a/x\} : \mathsf{End}_!}{\Gamma_2, a : S \vdash^\circ \circ M\{a/x\}}}{\Gamma_1, \Gamma_2, a : \overline{S}^\sharp \vdash^\bullet \bullet E[a] \parallel M\{a/x\}}$$
$$\Gamma_1, \Gamma_2 \vdash^\bullet (\nu a)(\bullet E[a] \parallel M\{a/x\})$$

(noting that $\overline{\overline{S}} = S$), as required.

Case E-Comm

$$\bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a] \quad \longrightarrow \quad \bullet E[a] \parallel \circ E'[(V, a)]$$

Assumption:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : S \vdash E[\mathbf{send}\, V\, a] : C}{\Gamma_1, \Gamma_2, a : S \vdash^\bullet \bullet E[\mathbf{send}\, V\, a]} \quad \dfrac{\Gamma_3, a : \overline{S} \vdash E'[\mathbf{receive}\, a] : \mathsf{End}_!}{\Gamma_3, a : \overline{S} \vdash^\circ \circ E'[\mathbf{receive}\, a]}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S^\sharp \vdash^\bullet \bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a]}$$

By Lemma 2, we have that:

$$\dfrac{\Gamma_2 \vdash V : A \quad a : !A.S' \vdash a : !A.S'}{\Gamma_2, a : !A.S' \vdash \mathbf{send}\, V\, a : S'}$$

Also by Lemma 2, we have that:

$$\dfrac{a : ?A.\overline{S'} \vdash a : ?A.\overline{S'}}{a : ?A.\overline{S'} \vdash \mathbf{receive}\, a : (A \times \overline{S'})}$$

This reasoning allows us to refine our original derivation:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : !A.S' \vdash E[\mathbf{send}\, V\, a] : C}{\Gamma_1, \Gamma_2, a : !A.S' \vdash^\bullet \bullet E[\mathbf{send}\, V\, a]} \quad \dfrac{\Gamma_3, a : ?A.\overline{S'} \vdash E'[\mathbf{receive}\, a] : \mathsf{End}_!}{\Gamma_3, a : ?A.\overline{S'} \vdash^\circ \circ E'[\mathbf{receive}\, a]}}{\Gamma_1, \Gamma_2, \Gamma_3, a : !A.S'^\sharp \vdash^\bullet \bullet E[\mathbf{send}\, V\, a] \parallel \circ E'[\mathbf{receive}\, a]}$$

By Lemma 3, $\Gamma_1, a : S' \vdash E[a] : C$, and $\Gamma_2, \Gamma_3, a : \overline{S'} \vdash E'[(V, a)] : \mathsf{End}_!$ (that Γ_2, Γ_3 is well-defined follows from the fact that the two environments are disjoint).

Recomposing:

$$\dfrac{\dfrac{\Gamma_1, a : S' \vdash E[a] : C}{\Gamma_1, a : S' \vdash^\bullet \bullet E[a]} \quad \dfrac{\Gamma_2, \Gamma_3, a : \overline{S'} \vdash E'[(V, a)] : \mathsf{End}_!}{\Gamma_2, \Gamma_3, a : \overline{S'} \vdash^\circ \circ E'[(V, a)]}}{\Gamma_1, \Gamma_2, \Gamma_3, a : S'^\sharp \vdash^\bullet \bullet E[a] \parallel \circ E'[(V, a)]}$$

Finally, we can show that the environment in the first derivation reduces to the environment in the final derivation:

$$\frac{!A.S' \longrightarrow S'}{\Gamma_1, \Gamma_2, \Gamma_3, a : !A.S'^\sharp \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3 : a : S'^\sharp}$$

as required.

Case E-Wait

$$(\nu a)(\mathcal{F}[\mathbf{wait}\, a] \parallel \circ a) \longrightarrow \mathcal{F}[()]$$

We prove the case where $\mathcal{F}[\mathbf{wait}\, a] = \bullet E[\mathbf{wait}\, a]$ (for some E); the case where $\mathcal{F}[\mathbf{wait}\, a] = \circ E[\mathbf{wait}\, a]$ is similar.

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma, a : \mathsf{End}_? \vdash E[\mathbf{wait}\, a] : A}{\Gamma, a : \mathsf{End}_? \vdash^\bullet \bullet E[\mathbf{wait}\, a]} \quad \dfrac{\dfrac{}{a : \mathsf{End}_! \vdash a : \mathsf{End}_!}}{a : \mathsf{End}_! \vdash^\circ \circ a}}{\Gamma, a : \mathsf{End}_?^\sharp \vdash^\bullet \bullet E[\mathbf{wait}\, a] \parallel \circ a}}{\Gamma \vdash^\bullet (\nu a)(\bullet E[\mathbf{wait}\, a] \parallel \circ a)}$$

By Lemma 2:

$$\dfrac{\dfrac{}{a : \mathsf{End}_? \vdash a : \mathsf{End}_?}}{a : \mathsf{End}_? \vdash \mathbf{wait}\, a : \mathbf{1}}$$

By Lemma 3, $\Gamma \vdash E[()] : B$. Recomposing:

$$\frac{\Gamma \vdash E[()] : A}{\Gamma \vdash^\bullet \bullet E[()]}$$

as required.

Case E-LiftM

Immediate by Lemmas 2, 6, and 3.

Case E-Lift

Immediate by Lemma 4, the induction hypothesis, and Lemma 5.

□

A.1.2 Equivalence

Lemma 7 *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, where the derivation of* $\mathcal{C} \equiv \mathcal{D}$ *does not contain a use of the axiom for associativity of parallel composition, then* $\Gamma \vdash^{\phi} \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$, examining the equivalence in both directions to account for symmetry. We show that a typing derivation of the left-hand side of an equivalence rule implies the existence of the right-hand side, and vice versa.

That reflexivity, transitivity, and symmetry of the equivalence relation respect typing follows immediately because equality of typing derivations is an equivalence relation.

We make implicit use of the induction hypothesis.

Congruence rules

Case Name restriction

$$\frac{\mathcal{C} \equiv \mathcal{D}}{(\nu a)\mathcal{C} \equiv (\nu a)\mathcal{D}}$$

$$\frac{\Gamma, a : S^{\sharp} \vdash^{\phi} \mathcal{C}}{\Gamma \vdash^{\phi} (\nu a)\mathcal{C}} \iff \frac{\Gamma, a : S^{\sharp} \vdash^{\phi} \mathcal{D}}{\Gamma \vdash^{\phi} (\nu a)\mathcal{D}}$$

Case Parallel Composition

$$\frac{\mathcal{C} \equiv \mathcal{D}}{\mathcal{C} \parallel \mathcal{E} \equiv \mathcal{D} \parallel \mathcal{E}}$$

There are two subcases, based on whether the parallel composition arises from T-Connect$_1$ or T-Connect$_2$.

Subcase T-Connect$_1$

$$\frac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : \overline{S} \vdash^{\phi_2} \mathcal{E}}{\Gamma_1, \Gamma_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{E}} \iff \frac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{D} \quad \Gamma_2, a : \overline{S} \vdash^{\phi_2} \mathcal{E}}{\Gamma_1, \Gamma_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{D} \parallel \mathcal{E}}$$

Subcase T-Connect$_2$

$$\frac{\Gamma_1, a : \overline{S} \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : S \vdash^{\phi_2} \mathcal{E}}{\Gamma_1, \Gamma_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{E}} \iff \frac{\Gamma_1, a : \overline{S} \vdash^{\phi_1} \mathcal{D} \quad \Gamma_2, a : S \vdash^{\phi_2} \mathcal{E}}{\Gamma_1, \Gamma_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{D} \parallel \mathcal{E}}$$

Equivalence Axioms

Case $\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C}$

There are two subcases, based on which rule is used for parallel composition.

Subcase T-Connect$_1$

$$\frac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^\sharp \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}} \iff \frac{\Gamma_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \quad \Gamma_1, a : S \vdash^{\phi_1} \mathcal{C}}{\Gamma_1, \Gamma_2, a : S^\sharp \vdash^{\phi_2 + \phi_1} \mathcal{D} \parallel \mathcal{C}}$$

Subcase T-Connect$_2$

$$\frac{\Gamma_1, a : \overline{S} \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : S \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^\sharp \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}} \iff \frac{\Gamma_2, a : S \vdash^{\phi_2} \mathcal{D} \quad \Gamma_1, a : \overline{S} \vdash^{\phi_1} \mathcal{C}}{\Gamma_1, \Gamma_2, a : S^\sharp \vdash^{\phi_2 + \phi_1} \mathcal{D} \parallel \mathcal{C}}$$

Case $\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D})$ if $a \notin \mathsf{fn}(\mathcal{C})$

There are again two subcases based on which parallel composition rule is used. The exact rule does not affect the discussion, so we show T-Connect$_1$.

$$\frac{\Gamma_1, b : T \vdash^{\phi_1} \mathcal{C} \quad \dfrac{\Gamma_2, a : S^\sharp, b : T \vdash^{\phi_2} \mathcal{D}}{\Gamma_2, b : \overline{T} \vdash^{\phi_2} (\nu a)\mathcal{D}}}{\Gamma_1, \Gamma_2, b : T^\sharp \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel (\nu a)\mathcal{D}} \iff \frac{\dfrac{\Gamma_1, b : T \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : S^\sharp, b : \overline{T} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^\sharp, b : T^\sharp \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}}}{\Gamma_1, \Gamma_2, b : T^\sharp \vdash^{\phi_1 + \phi_2} (\nu a)(\mathcal{C} \parallel \mathcal{D})}$$

In the left-to-right direction, that $\Gamma_1, \Gamma_2, a : S^\sharp$ is well-defined follows because $a \notin \mathsf{fn}(\mathcal{C})$.

Case $(\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$

$$\frac{\dfrac{\Gamma, a : S^\sharp, b : T^\sharp \vdash^{\phi} \mathcal{C}}{\Gamma, a : S^\sharp \vdash^{\phi} (\nu b)\mathcal{C}}}{\Gamma \vdash^{\phi} (\nu a)(\nu b)\mathcal{C}} \iff \frac{\dfrac{\Gamma, b : T^\sharp, a : S^\sharp \vdash^{\phi} \mathcal{C}}{\Gamma, b : T^\sharp \vdash^{\phi} (\nu a)\mathcal{C}}}{\Gamma \vdash^{\phi} (\nu b)(\nu a)\mathcal{C}}$$

□

Appendix B

Proofs for Chapters 4–6 (Mixing Metaphors)

B.1 Preservation ($\lambda_{\mathbf{ch}}$)

Theorem 9: Preservation ($\lambda_{\mathbf{ch}}$ configurations) *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$ *then* $\Gamma;\Delta \vdash \mathcal{C}_2$.

Proof. By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$.

Case E-Give

$$E[\mathbf{give}\, W\, a] \parallel a(\overrightarrow{V}) \quad \longrightarrow \quad E[\mathbf{return}\, ()] \parallel a(\overrightarrow{V} \cdot W)$$

Assumption:

$$\frac{\dfrac{\Gamma \vdash E[\mathbf{give}\, W\, a] : B}{\Gamma;\cdot \vdash E[\mathbf{give}\, W\, a]} \qquad \dfrac{(\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash a(\overrightarrow{V})}}{\Gamma; a : A \vdash E[\mathbf{give}\, W\, a] \parallel a(\overrightarrow{V})}$$

Note that by T-Chan, Γ must contain $a : \mathsf{ChanRef}(A)$.

By Lemma 12, we have:

$$\frac{\Gamma \vdash W : A \quad \Gamma \vdash a : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathbf{give}\, W\, a : \mathbf{1}}$$

By Lemma 13, we have that $\Gamma \vdash E[\mathbf{return}\, ()] : B$.

By T-Buf:

$$\frac{(\Gamma \vdash V_i : A)_i \quad \Gamma \vdash W : A}{\Gamma; a : A \vdash a(\overrightarrow{V} \cdot W)}$$

Recomposing, we have:

$$\frac{\dfrac{\Gamma \vdash E[\mathbf{return}\ ()] : B}{\Gamma; \cdot \vdash E[\mathbf{return}\ ()]} \quad \dfrac{(\Gamma \vdash V_i : A)_i \quad \Gamma \vdash W : A}{\Gamma; a : A \vdash a(\overrightarrow{V} \cdot W)}}{\Gamma; a : A \vdash E[\mathbf{return}\ ()] \parallel a(\overrightarrow{V} \cdot W)}$$

Case E-Take

$$E[\mathbf{take}\, a] \parallel a(W \cdot \overrightarrow{V}) \quad \longrightarrow \quad E[\mathbf{return}\, W] \parallel a(\overrightarrow{V})$$

Assumption:

$$\frac{\dfrac{\Gamma \vdash E[\mathbf{take}\, a] : B}{\Gamma; \cdot \vdash E[\mathbf{take}\, a]} \quad \dfrac{\Gamma \vdash W : A \quad (\Gamma \vdash V_i : A)_i}{\Gamma, a : A \vdash a(W \cdot \overrightarrow{V})}}{\Gamma; a : A \vdash E[\mathbf{take}\, a] \parallel a(W \cdot \overrightarrow{V})}$$

where $\Gamma = \Gamma', a : \mathsf{ChanRef}(A)$, due to rule T-Buf.

By Lemma 12:

$$\frac{\Gamma \vdash a : \mathsf{ChanRef}(A)}{\Gamma \vdash \mathbf{take}\, a : A}$$

By Lemma 13, $\Gamma \vdash E[\mathbf{return}\, W] : B$.

Recomposing:

$$\frac{\dfrac{\Gamma \vdash E[\mathbf{return}\, W] : C}{\Gamma; \cdot \vdash E[\mathbf{return}\, W]} \quad \dfrac{(\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash a(\overrightarrow{V})}}{\Gamma; a : A \vdash E[\mathbf{return}\, W] \parallel a(\overrightarrow{V})}$$

as required.

Case E-Fork

$$E[\mathbf{fork}\, M] \quad \longrightarrow \quad E[\mathbf{return}\ ()] \parallel M$$

Assumption:

$$\frac{\Gamma \vdash E[\mathbf{fork}\, M] : A}{\Gamma; \cdot \vdash E[\mathbf{fork}\, M]}$$

By Lemma 12:

$$\frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{fork}\, M : \mathbf{1}}$$

By Lemma 13, $\Gamma \vdash E[\mathbf{return}\ ()] : A$.

Recomposing:

$$\frac{\dfrac{\Gamma \vdash E[\mathbf{return}\ ()] : A}{\Gamma;\cdot \vdash E[\mathbf{return}\ ()]} \qquad \dfrac{\Gamma \vdash M : B}{\Gamma;\cdot \vdash M}}{\Gamma;\cdot \vdash E[\mathbf{return}\ ()] \parallel M}$$

as required.

Case E-NewCh

$$E[\mathbf{newCh}] \quad \longrightarrow \quad (\nu a)(E[\mathbf{return}\, a] \parallel a(\varepsilon)) \qquad (a \text{ is fresh})$$

Assumption:

$$\frac{\Gamma \vdash E[\mathbf{newCh}] : B}{\Gamma;\cdot \vdash E[\mathbf{newCh}]}$$

By Lemma 12:

$$\frac{}{\Gamma \vdash \mathbf{newCh} : \mathsf{ChanRef}(A)}$$

By T-Name, $\Gamma, a : \mathsf{ChanRef}(A) \vdash a : \mathsf{ChanRef}(A)$. By Lemma 13, $\Gamma, a : \mathsf{ChanRef}(A) \vdash E[\mathbf{return}\, a] : B$.

Recomposing:

$$\frac{\dfrac{\dfrac{\Gamma, a : \mathsf{ChanRef}(A) \vdash E[\mathbf{return}\, a] : B}{\Gamma, a : \mathsf{ChanRef}(A);\cdot \vdash E[\mathbf{return}\, a]} \qquad \dfrac{}{\Gamma, a : \mathsf{ChanRef}(A); a : A \vdash \varepsilon()}}{\Gamma, a : \mathsf{ChanRef}(A); a : A \vdash E[\mathbf{return}\, a] \parallel a(\varepsilon)}}{\Gamma;\cdot \vdash (\nu a)(E[\mathbf{return}\, a] \parallel a(\varepsilon))}$$

as required.

Case E-LiftM

$$M_1 \quad \longrightarrow \quad M_2 \qquad (\text{if } M_1 \longrightarrow_{\mathsf{M}} M_2)$$

Assumption:

$$\frac{\Gamma \vdash M_1 : A}{\Gamma;\cdot \vdash M_1}$$

and $M_1 \longrightarrow_{\mathrm{M}} M_2$.

By Lemma 14, $\Gamma \vdash M_2 : A$. Thus:

$$\frac{\Gamma \vdash M_2 : A}{\Gamma; \cdot \vdash M_2}$$

as required.

Case E-Lift

$$\mathcal{G}[\mathcal{C}_1] \longrightarrow \mathcal{G}[\mathcal{C}_2] \qquad \text{and } \mathcal{C}_1 \longrightarrow \mathcal{C}_2$$

Assumption: $\Gamma; \Delta \vdash \mathcal{G}[\mathcal{C}_1]$.

By Lemma 16, we have that there exist Γ', Δ' such that $\Gamma'; \Delta' \vdash \mathcal{C}_1$.

By the induction hypothesis, we have that $\Gamma'; \Delta' \vdash \mathcal{C}_2$.

By Lemma 17, we have that $\Gamma; \Delta \vdash \mathcal{G}[\mathcal{C}_2]$ as required.

□

B.2 Preservation ($\lambda_{\mathbf{act}}$)

Theorem 11: Preservation ($\lambda_{\mathbf{act}}$ configurations) *If* $\Gamma; \Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, *then* $\Gamma; \Delta \vdash \mathcal{C}_2$.

Proof. By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$.

Case E-Spawn

$$\langle a, E[\mathbf{spawn}\, M], \overrightarrow{V} \rangle \longrightarrow (\nu b)(\langle a, E[\mathbf{return}\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \varepsilon \rangle)$$

where b is fresh.

Assumption:

$$\frac{\Gamma \mid A \vdash E[\mathbf{spawn}\, M] : A' \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash \langle a, E[\mathbf{spawn}\, M], \overrightarrow{V} \rangle}$$

By Lemma 20:

$$\frac{\Gamma \mid B \vdash M : B'}{\Gamma \mid A \vdash \mathbf{spawn}\, M : \mathsf{ActorRef}(B)}$$

By Lemma 21, $\Gamma, b : \mathsf{ActorRef}(B) \mid A \vdash E[\mathbf{return}\, b] : A'$. Let $\Gamma' = \Gamma, b : \mathsf{ActorRef}(B)$.

Recomposing:

$$\dfrac{\dfrac{\Gamma' \mid A \vdash E[\mathbf{return}\, b] : A' \quad (\Gamma' \vdash V_i : A)_i}{\Gamma'; a : A \vdash \langle a, E[\mathbf{return}\, b], \overrightarrow{V} \rangle} \quad \dfrac{\Gamma' \mid B \vdash M : B'}{\Gamma'; b : B \vdash \langle b, M, \varepsilon \rangle}}{\dfrac{\Gamma'; a : A, b : B \vdash \langle a, E[\mathbf{return}\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \varepsilon \rangle}{\Gamma; a : A \vdash (\nu b)(\langle a, E[\mathbf{return}\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \varepsilon \rangle)}}$$

as required.

Case E-Send

$$\text{E-Send} \quad \langle a, E[\mathbf{send}\, V'\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \rangle \longrightarrow \langle a, E[\mathbf{return}\, ()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot V' \rangle$$

Let $\Gamma = \Gamma', a : \mathrm{ActorRef}(A), b : \mathrm{ActorRef}(B)$ for some Γ'.

Assumption:

$$\dfrac{\dfrac{\Gamma \mid A \vdash E[\mathbf{send}\, V'\, b] : C \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash \langle a, E[\mathbf{send}\, V'\, b], \overrightarrow{V} \rangle} \quad \dfrac{\Gamma \mid B \vdash M : C' \quad (\Gamma \vdash W_i : B)_i}{\Gamma; b : B \vdash \langle b, M, \overrightarrow{W} \rangle}}{\Gamma; a : A, b : B \vdash \langle a, E[\mathbf{send}\, V'\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \rangle}$$

Note that $a : \mathrm{ActorRef}(A)$ and $b : \mathrm{ActorRef}(B)$ must be in Γ due to rule T-Actor.

By Lemma 20:

$$\dfrac{\Gamma \vdash V' : B \quad \Gamma \vdash b : \mathrm{ActorRef}(B)}{\Gamma \mid B \vdash \mathbf{send}\, V'\, b : \mathbf{1}}$$

By Lemma 21, we have that $\Gamma \mid B \vdash E[\mathbf{return}\, ()] : C$.

By T-Actor:

$$\dfrac{\begin{array}{c}\Gamma \mid B \vdash M : C' \\ (\Gamma \vdash W_i : B)_i \quad \Gamma \vdash V' : B\end{array}}{\Gamma; b : B \vdash \langle b, M, \overrightarrow{W} \cdot V' \rangle}$$

Recomposing:

$$\dfrac{\dfrac{\Gamma \mid A \vdash E[\mathbf{return}\, ()] : C \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash \langle a, E[\mathbf{return}\, ()], \overrightarrow{V} \rangle} \quad \dfrac{\begin{array}{c}\Gamma \mid B \vdash M : C' \\ (\Gamma \vdash W_i : B)_i \quad \Gamma \vdash V' : B\end{array}}{\Gamma; b : B \vdash \langle b, M, \overrightarrow{W} \cdot V' \rangle}}{\Gamma; a : A, b : B \vdash \langle a, E[\mathbf{return}\, ()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot V' \rangle}$$

as required.

Case E-SendSelf

$$\langle a, E[\mathbf{send}\, V'\, a], \overrightarrow{V}\rangle \longrightarrow \langle a, E[\mathbf{return}\, ()], \overrightarrow{V} \cdot V'\rangle$$

Assumption:

$$\frac{\Gamma \mid A \vdash E[\mathbf{send}\, V'\, a] : A' \quad (\Gamma \vdash V_i : A)_i}{\Gamma; a : A \vdash \langle a, E[\mathbf{send}\, V'], \overrightarrow{V}\rangle}$$

where (due to T-Process), $\Gamma = \Gamma', a : \mathsf{ActorRef}(A)$.

By Lemma 20:

$$\frac{\Gamma \vdash V' : A \quad \Gamma \vdash a : \mathsf{ActorRef}(A)}{\Gamma \mid A \vdash \mathbf{send}\, V'\, a : \mathbf{1}}$$

By Lemma 21, $\Gamma \mid A \vdash E[\mathbf{return}\, ()] : A'$.

Recomposing:

$$\frac{\Gamma \mid A \vdash E[\mathbf{return}\, ()] : A' \quad (\Gamma \vdash V_i : A)_i \quad \Gamma \vdash V' : A}{\Gamma, a : A \vdash \langle a, E[\mathbf{return}\, ()], \overrightarrow{V} \cdot V'\rangle}$$

Case E-Self

$$\langle a, E[\mathbf{self}], \overrightarrow{V}\rangle \longrightarrow \langle a, E[\mathbf{return}\, a], \overrightarrow{V}\rangle$$

Assumption:

$$\frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash E[\mathbf{self}] : B \quad (\Gamma \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, E[\mathbf{self}], \overrightarrow{V}\rangle}$$

By Lemma 20:

$$\frac{}{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash \mathbf{self} : \mathsf{ActorRef}(A)}$$

By T-Name, $\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash \mathbf{return}\, a : \mathsf{ActorRef}(A)$.

By Lemma 21: $\Gamma, a : \mathsf{ActorRef}(A) \vdash E[\mathbf{return}\, a] : B$.

Recomposing:

$$\frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash E[\mathbf{return}\, a] : B \quad (\Gamma \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, E[\mathbf{return}\, a], \overrightarrow{V}\rangle}$$

as required.

Case E-Receive

$$\langle a, E[\textbf{receive}], V' \cdot \overrightarrow{V} \rangle \longrightarrow \langle a, E[\textbf{return}\, V'], \overrightarrow{V} \rangle$$

Assumption:

$$\frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash E[\textbf{receive}] : B \quad \Gamma \vdash V' : A \quad (\Gamma \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, E[\textbf{self}], V' \cdot \overrightarrow{V} \rangle}$$

By Lemma 20:

$$\frac{}{\Gamma \mid A \vdash \textbf{receive} : A}$$

By Lemma 21:

$$\frac{}{\Gamma \mid A \vdash \textbf{return}\, V' : A}$$

Recomposing:

$$\frac{\Gamma, a : \mathsf{ActorRef}(A) \mid A \vdash E[\textbf{return}\, V'] : B \quad (\Gamma \vdash V_i : A)_i}{\Gamma, a : \mathsf{ActorRef}(A); a : A \vdash \langle a, E[\textbf{return}\, V'], \overrightarrow{V} \rangle}$$

Case E-LiftM

Immediate by Lemmas 20, 22, and 21.

Case E-Lift

Immediate by Lemmas 24, the induction hypothesis, and 25.

□

B.3 Translation (λ_{act} into λ_{ch})

B.3.1 Operational Correspondence

Theorem 14: Operational Correspondence ($[\![-]\!]$)

Simulation *If* $\Gamma; \Delta \vdash C_1$ *and* $C_1 \longrightarrow C_2$*, then* $[\![C_1]\!] \Longrightarrow^* [\![C_2]\!]$

Reflection *If* $\Gamma; \Delta \vdash C_1$ *and* $[\![C_1]\!] \Longrightarrow \mathcal{D}$*, then there exists some* C_2 *such that* $C_1 \Longrightarrow C_2$ *and* $\mathcal{D} \Longrightarrow^* [\![C_2]\!]$

Proof.

Simulation: By induction on the derivation of $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$.

Case E-Spawn

$$
\begin{array}{l}
[\![\langle a, E[\mathbf{spawn}\, M], \overrightarrow{V} \rangle]\!] \\
\quad \text{Def. } [\![-]\!] \\
a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{let}\, c \Leftarrow \mathbf{newCh\ in\ fork}\, ([\![M]\!]_c); \mathbf{return}\, c]\, a) \\
\quad \longrightarrow \text{(E-NewCh)} \\
a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathbf{let}\, c \Leftarrow \mathbf{return}\, b\ \mathbf{in\ fork}\, ([\![M]\!]_c); \mathbf{return}\, c]\, a) \parallel b(\varepsilon)) \\
\quad \longrightarrow_{\mathrm{M}} \\
a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathbf{fork}\, ([\![M]\!]_b); \mathbf{return}\, b]\, a) \parallel b(\varepsilon)) \\
\quad \longrightarrow \text{(E-Fork)} \\
a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathbf{return}\, (); \mathbf{return}\, b]\, a) \parallel ([\![M]\!]_b) \parallel b(\varepsilon)) \\
\quad \longrightarrow_{\mathrm{M}} \\
a([\![\overrightarrow{V}]\!]) \parallel (\nu b)(([\![E]\!][\mathbf{return}\, b]\, a) \parallel ([\![M]\!]_b) \parallel b(\varepsilon)) \\
\quad \equiv \\
(\nu b)(a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{return}\, b]\, a) \parallel b(\varepsilon) \parallel ([\![M]\!]_b)) \\
\quad = \\
[\![(\nu b)(\langle a, E[\mathbf{return}\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \varepsilon \rangle)]\!]
\end{array}
$$

Case E-Self

$$
\begin{array}{l}
[\![\langle a, E[\mathbf{self}], \overrightarrow{V} \rangle]\!] \\
\quad \text{Def. } [\![-]\!] \\
a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{return}\, a]\, a) \\
\quad = \\
[\![\langle a, E[\mathbf{return}\, a], \overrightarrow{V} \rangle]\!]
\end{array}
$$

Case E-Send

$$
\begin{array}{l}
[\![\langle a, E[\mathbf{send}\, V'\, b], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \rangle]\!] \\
\quad \text{Def. } [\![-]\!] \\
a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{give}\, [\![V']\!]\, b]\, a) \parallel b([\![\overrightarrow{W}]\!]) \parallel ([\![M]\!]_b) \\
\quad \longrightarrow \text{(E-Give)} \\
a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{return}\, ()]\, a) \parallel b([\![\overrightarrow{W}]\!] \cdot [\![V']\!]) \parallel ([\![M]\!]_b) \\
\quad = \\
[\![\langle a, E[\mathbf{return}\, ()], \overrightarrow{V} \rangle \parallel \langle b, M, \overrightarrow{W} \cdot V' \rangle]\!]
\end{array}
$$

Case E-Receive

$$
\begin{array}{l}
[\![\, \langle a, E[\mathbf{receive}], W \cdot \overrightarrow{V} \rangle \,]\!] \\
\quad \text{Def. } [\![-]\!] \\
a([\![W]\!] \cdot [\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{take}\, a]\, a) \\
\quad \longrightarrow \text{(E-Take)} \\
a([\![\overrightarrow{V}]\!]) \parallel ([\![E]\!][\mathbf{return}\, [\![W]\!]]\, a) \\
\quad = \\
[\![\, \langle a, E[\mathbf{return}\, W], \overrightarrow{V} \rangle \,]\!]
\end{array}
$$

Lift is immediate by the induction hypothesis, and LiftV is immediate from Lemma 29.

Reflection:

Assume Δ contains entries $a_1 : A_1, \ldots, a_k : A_k$.

By Lemma 27:

$$\mathcal{C}_1 \equiv (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, M_1, \overrightarrow{V_1} \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle) = \mathcal{C}_{\text{canon}}$$

By the definition of $[\![-]\!]$:

$$[\![\, C_{\text{canon}} \,]\!] = (\nu a_{k+1}) \cdots (\nu a_n)(a_1(\overrightarrow{V_1}) \parallel [\![M_1]\!]_{a_1} \parallel \cdots \parallel a_n(\overrightarrow{V_n}) \parallel [\![M_n]\!]_{a_n})$$

We proceed by case analysis on the structure of translated terms, inspecting the reduction rules for λ_{ch}. Without loss of generality, we assume that term reduction in λ_{act} occurs in the thread translated with respect to name a_1.

Case $[\![\, \mathbf{send}\, V\, W \,]\!]_{a_1}$

$$[\![E[\mathbf{send}\, V\, W]]\!]_{a_1} = [\![E]\!][\mathbf{give}\, [\![V]\!] [\![W]\!]]\, a_1$$

The applicable reduction rule is E-Give. Thus, there exists some $\mathcal{G}$ such that:

$$\mathcal{C}_{\text{canon}} \equiv \mathcal{G}[[\![E]\!][\mathbf{give}\, [\![W]\!]\, b]\, a_1 \parallel b([\![\overrightarrow{V}]\!])]$$

We have two subcases, based on the value of b. We need not consider free names or variables without an associated buffer, as these would not reduce.

Subcase $b = a_1$

$$
\begin{array}{l}
[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_1]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) \\
\quad \longrightarrow \text{(E-Give)} \\
[\![E]\!][\mathbf{return}\, ()]\, a_1 \parallel b([\![\overrightarrow{V_1}]\!] \cdot [\![W]\!]) \\
\quad = \\
[\![\, \langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W \rangle \,]\!]
\end{array}
$$

By the definition of $[\![-]\!]$:

$$[\![E]\!][\mathbf{give}\, W\, b]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) = [\![\langle a_1, E[\mathbf{send}\, W\, a_1], \overrightarrow{V_1}\rangle]\!]$$

In λ_{act}, we can show:

$$\begin{array}{l}\langle a_1, E[\mathbf{send}\, W\, a_1], \overrightarrow{V_1}\rangle \\ \quad \longrightarrow \text{(E-SendSelf)} \\ \langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W\rangle\end{array}$$

Thus, we can see that:

$$[\![\, \mathcal{C}_{\text{canon}}\,]\!] \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)([\![E]\!][\mathbf{return}\, ()]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!] \cdot [\![W]\!]) \parallel \cdots \parallel [\![M_n]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!]))$$
$$=$$
$$[\![\, (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W\rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)\,]\!]$$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1} \cdot W\rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$$

as required.

Subcase $b = a_j$ for some $j \neq 1$

Without loss of generality, consider the case where $j = n$. By the definition of $[\![\, \mathcal{C}_{\text{canon}}\,]\!]$, we have that there must exist some $\mathcal{G}'$ such that

$$\begin{array}{l}\mathcal{G}[[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_n([\![\overrightarrow{V_n}]\!])] \\ \quad \equiv \\ \mathcal{G}'[[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_n([\![\overrightarrow{V_n}]\!]) \parallel [\![M]\!]_{a_n} \parallel a_1([\![\overrightarrow{V_1}]\!])] \\ \quad \equiv \\ \mathcal{G}'[[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!])] \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!])\end{array}$$

Next, we observe the E-Give reduction:

$$\begin{array}{l}[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!]) \\ \quad \Longrightarrow \text{(E-Give)} \\ [\![E]\!][\mathbf{return}\, ()]\, a_1 \parallel a_1([\![\overrightarrow{V_1}]\!]) \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!] \cdot [\![W]\!]) \\ \quad = \\ [\![\, \langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1}\rangle \parallel \langle a_n, M_n, V_n \cdot W\rangle\,]\!]\end{array}$$

By the definition of $[\![-]\!]$:

$$[\![E]\!][\mathbf{give}\, [\![W]\!]\, a_n]\, a_1 \parallel a_1(\overrightarrow{V_1}) \parallel [\![M]\!]_{a_n} \parallel a_n([\![\overrightarrow{V_n}]\!]) = [\![\langle a_1, E[\mathbf{send}\, W\, a_n], \overrightarrow{V_1}\rangle \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle]\!]$$

In λ_{act}, we can show:

$$\begin{array}{l}\langle a_1, E[\mathbf{send}\, W\, a_n], \overrightarrow{V_1}\rangle \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle \\ \quad \longrightarrow \text{(E-Send)} \\ \langle a_1, E[\mathbf{return}\, ()], \overrightarrow{V_1}\rangle \parallel \langle a_n, M_n, \overrightarrow{V_n} \cdot W\rangle\end{array}$$

Thus, we can see that:

$$[\![\, \mathcal{C}_{\text{canon}} \,]\!] \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)([\![E]\!][\mathbf{return}\ ()]\, a_1 \parallel a_1([\![\, \overrightarrow{V_1} \,]\!]) \parallel \cdots \parallel [\![M]\!]_{a_n} \parallel a_n([\![\, \overrightarrow{V_n} \,]\!] \cdot [\![W]\!]))$$
$$=$$
$$[\![\, (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\ ()], \overrightarrow{V_1} \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \cdot W \rangle) \,]\!]$$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\ ()], \overrightarrow{V_1} \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \cdot W \rangle)$$

as required.

Case $[\![\, \mathbf{self} \,]\!]_{a_1}$

$$[\![\, \mathbf{self} \,]\!]_{a_1} = \mathbf{return}\, a_1$$

Holds vacuously, as no reduction rule applies.

Case $[\![\, \mathbf{receive} \,]\!]_{a_1}$

$$[\![\, \mathbf{receive} \,]\!]_{a_1} = \mathbf{take}\, a_1$$

Thus, the applicable reduction rule is E-Take, and we have that there exists some configuration context $\mathcal{G}$ such that:

$$\begin{aligned} &[\![\, \mathcal{C}_{\text{canon}} \,]\!] \equiv \mathcal{G}[[\![E]\!][\mathbf{take}\, a_1]\, a_1 \parallel a_1([\![W]\!] \cdot [\![\, \overrightarrow{V_n} \,]\!])] \\ &\quad \longrightarrow \text{(E-Take)} \\ &\mathcal{G}[[\![E]\!][\mathbf{return}\, [\![W]\!]]\, a_1 \parallel a_1([\![\, \overrightarrow{V_n} \,]\!])] \end{aligned}$$

By the definition of $[\![-]\!]$:

$$[\![E]\!][\mathbf{take}\, a_1]\, a_1 \parallel a_1([\![W]\!] \cdot [\![\, \overrightarrow{V_n} \,]\!]) = [\![\, \langle a_1, E[\mathbf{receive}], W \cdot \overrightarrow{V_n} \rangle \,]\!]$$

In λ_{act}, we may write

$$\begin{aligned} &[\![\, \langle a_1, E[\mathbf{receive}], W \cdot \overrightarrow{V_n} \rangle \,]\!] \\ &\quad \Longrightarrow \text{(E-Recv)} \\ &[\![\, \langle a_1, E[\mathbf{return}\, W], \overrightarrow{V_n} \rangle \,]\!] \end{aligned}$$

Thus, we can see that:

$$[\![\, \mathcal{C}_{\text{canon}} \,]\!] \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)([\![E]\!][\mathbf{return}\, W]\, a_1 \parallel a([\![\, \overrightarrow{V_1} \,]\!]) \parallel \cdots \parallel [\![M_n]\!]_{a_n} \parallel a_n([\![\, \overrightarrow{V_n} \,]\!]))$$
$$=$$
$$[\![\, (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\mathbf{return}\, W], \overrightarrow{V_1} \rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n} \rangle) \,]\!]$$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(\langle a_1, E[\textbf{return}\, W], \overrightarrow{V_1}\rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle)$$

as required.

Case $[\![\, \textbf{spawn}\, M \,]\!]_{a_1}$

$$[\![\, \textbf{spawn}\, M \,]\!]_{a_1} = \textbf{let}\ chMb \Leftarrow \textbf{newCh}\ \textbf{in}$$
$$\textbf{fork}\, [\![\, M \,]\!]_{chMb}$$
$$\textbf{return}\ chMb$$

$[\![\, \mathcal{C}_{\text{canon}} \,]\!]$

$\equiv$

$\textbf{let}\ chMb \Leftarrow \textbf{newCh}\ \textbf{in}$

$\mathcal{G}[[\![E]\!][\textbf{fork}\, [\![\, M \,]\!]_{chMb}; \quad]\, a_1 \parallel a_1([\![\, \overrightarrow{V_1} \,]\!])]$

$\textbf{return}\ chMb$

By constructing the same derivation sequence as for E-Spawn in the simulation case :

$(\nu b)([\![E]\!][\textbf{return}\, b]\, a_1 \parallel a_1([\![\, \overrightarrow{V_1} \,]\!]) \parallel [\![\, M \,]\!]_b \parallel b(\varepsilon))$

$=$

$[\![\, (\nu b)(\langle a_1, E[\textbf{return}\, b], \overrightarrow{V_1}\rangle \parallel \langle b, M, \varepsilon\rangle) \,]\!]$

In λ_{act}, we may write

$$\langle a_1, E[\textbf{spawn}\, M], \overrightarrow{V}\rangle$$
$$\longrightarrow \text{(E-Spawn)}$$
$$(\nu b)(\langle a_1, E[\textbf{return}\, ()], \overrightarrow{V_1}\rangle \parallel \langle b, M, \varepsilon\rangle)$$

Thus, we can see that:

$[\![\, \mathcal{C}_{\text{canon}} \,]\!]$

$\Longrightarrow$

$(\nu a_{k+1}) \cdots (\nu a_n)(\nu b)([\![\, E[\textbf{return}\, b] \,]\!]_{a_1} \parallel a_1([\![\, \overrightarrow{V_1} \,]\!]) \parallel \cdots \parallel [\![\, M_n \,]\!]_{a_n} \parallel a_n([\![\, \overrightarrow{V_n} \,]\!]) \parallel [\![\, M \,]\!]_b \parallel b(\varepsilon))$

$=$

$[\![\, (\nu a_{k+1}) \cdots (\nu a_n)(\nu b)(\langle a_1, E[\textbf{return}\, b], \overrightarrow{V_1}\rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle \parallel \langle b, M, \varepsilon\rangle) \,]\!]$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(\nu b)(\langle a_1, E[\textbf{return}\, b], \overrightarrow{V_1}\rangle \parallel \cdots \parallel \langle a_n, M_n, \overrightarrow{V_n}\rangle \parallel \langle b, M, \varepsilon\rangle)$$

as required.

Case $[\![M]\!]_{ch}$, where M is not a communication or concurrency construct

Follows from Lemma 30.

□

B.4 Translation ($\lambda_{\mathbf{ch}}$ into $\lambda_{\mathbf{act}}$)

B.4.1 Operational Correspondence

Theorem 16: Operational Correspondence ($(\!| - |\!)$)

Simulation *If* $\{A\}\ \Gamma;\Delta \vdash C_1$, *and* $C_1 \longrightarrow C_2$, *then* $(\!| C_1 |\!) \Longrightarrow^* (\!| C_2 |\!)$.

Reflection *If* $\{A\}\ \Gamma;\Delta \vdash C_1$, *and* $(\!| C_1 |\!) \Longrightarrow \mathcal{D}$, *then there exist configurations* C_2 *and* $\mathcal{E}$ *such that* $C_1 \Longrightarrow^? C_2$ *and* $\mathcal{D} \Longrightarrow^* \mathcal{E}$, *where* $\mathcal{E} =_\beta (\!| C_2 |\!)$.

Proof. **Simulation:** By induction on the derivation of $C_1 \longrightarrow C_2$.

Case E-Give

$E[\mathbf{give}\, W\, a] \parallel a(\overrightarrow{V})$

Def. $(\!|-|\!)$

$(\nu b)(\langle b, (\!|E|\!)[\mathbf{send}\,(\mathbf{inl}\,(\!|W|\!))\,a], \varepsilon\rangle) \parallel \langle a, \mathsf{body}\,((\!|\overrightarrow{V}|\!)), [\,]), \varepsilon\rangle$

$\equiv$

$(\nu b)(\langle b, (\!|E|\!)[\mathbf{send}\,(\mathbf{inl}\,(\!|W|\!))\,a], \varepsilon\rangle \parallel \langle a, \mathsf{body}\,((\!|\overrightarrow{V}|\!)), [\,]), \varepsilon\rangle)$

$\longrightarrow$ (E-Send)

$(\nu b)(\langle b, (\!|E|\!)[\mathbf{return}\,()], \varepsilon\rangle \parallel \langle a, \mathsf{body}\,((\!|\overrightarrow{V}|\!)), [\,]), \mathbf{inl}\,(\!|W|\!)\rangle)$

Let $\mathcal{G}[-] = (\nu b)([-] \parallel \langle b, (\!|E|\!)[\mathbf{return}\,()], \varepsilon\rangle)$

$\mathcal{G}[\langle a, \mathsf{body}\,([(\!|\overrightarrow{V}|\!)], [\,]), (\mathbf{inl}\,(\!|W|\!))\rangle]$

$=$ (expanding body)

$$\begin{aligned}
\mathcal{G}[\langle a,\ & (\mathbf{rec}\ g(state)\ . && ,\mathbf{inl}\,(\!|W|\!)\rangle] \\
& \quad \mathbf{let}\ recvVal \Leftarrow \mathbf{receive}\ \mathbf{in} \\
& \quad \mathbf{let}\ (vals, readers) = state\ \mathbf{in} \\
& \quad\ \ \mathbf{case}\ recvVal\,\{ \\
& \quad\quad \mathbf{inl}\, v \mapsto \mathbf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \mathbf{in} \\
& \quad\quad\quad\quad \mathbf{let}\ state' \Leftarrow \mathsf{drain}\,(newVals, readers)\ \mathbf{in} \\
& \quad\quad\quad\quad g\,(state') \\
& \quad\quad \mathbf{inr}\, pid \mapsto \mathbf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \mathbf{in} \\
& \quad\quad\quad\quad\quad \mathbf{let}\ state' \Leftarrow \mathsf{drain}\,(vals, newReaders)\ \mathbf{in} \\
& \quad\quad\quad\quad\quad g\,(state')\})\,((\!|\overrightarrow{V}|\!)), [\,])
\end{aligned}$$

$\longrightarrow_{\mathsf{M}}$

$$\begin{aligned}
\mathcal{G}[\langle a,\ & \mathbf{let}\ recvVal \Leftarrow \mathbf{receive}\ \mathbf{in} && ,\mathbf{inl}\,(\!|W|\!)\rangle] \\
& \quad \mathbf{let}\ (vals, readers) = ((\!|\overrightarrow{V}|\!)), [\,])\ \mathbf{in} \\
& \quad\ \ \mathbf{case}\ recvVal\,\{ \\
& \quad\quad \mathbf{inl}\, v \mapsto \mathbf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \mathbf{in} \\
& \quad\quad\quad\quad \mathbf{let}\ state' \Leftarrow \mathsf{drain}\,(newVals, readers)\ \mathbf{in} \\
& \quad\quad\quad\quad \mathsf{body}\,(state') \\
& \quad\quad \mathbf{inr}\, pid \mapsto \mathbf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \mathbf{in} \\
& \quad\quad\quad\quad\quad \mathbf{let}\ state' \Leftarrow \mathsf{drain}\,(vals, newReaders)\ \mathbf{in} \\
& \quad\quad\quad\quad\quad \mathsf{body}\,(state')\})
\end{aligned}$$

$$
\begin{array}{l}
\longrightarrow \text{(E-Receive)} \\
\mathcal{G}[\langle a,\ \textbf{let}\ recvVal \Leftarrow \textbf{return}\ \textbf{inl}\ (\!|\, W \,|\!)\ \textbf{in} \qquad\qquad ,\varepsilon\rangle] \\
\quad\ \ \textbf{let}\ (vals, readers) = (\!|\, \overrightarrow{V} \,|\!), [\,]\ \textbf{in} \\
\quad\quad \textbf{case}\ recvVal\,\{ \\
\quad\quad\ \ \textbf{inl}\ v \mapsto \textbf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(newVals, readers)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \mathsf{body}\,(state') \\
\quad\quad\quad\ \ \textbf{inr}\ pid \mapsto \textbf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(vals, newReaders)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{body}\,(state')\}) \\
\longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a,\ \textbf{let}\ (vals, readers) = (\!|\, \overrightarrow{V} \,|\!), [\,]\ \textbf{in} \qquad\qquad ,\varepsilon\rangle] \\
\quad\quad \textbf{case}\ \textbf{inl}\ (\!|\, W \,|\!)\,\{ \\
\quad\quad\ \ \textbf{inl}\ v \mapsto \textbf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(newVals, readers)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \mathsf{body}\,(state') \\
\quad\quad\quad\ \ \textbf{inr}\ pid \mapsto \textbf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(vals, newReaders)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{body}\,(state')\}) \\
\longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a,\ \textbf{case}\ \textbf{inl}\ (\!|\, W \,|\!)\,\{ \qquad\qquad ,\varepsilon\rangle] \\
\quad\quad\ \ \textbf{inl}\ v \mapsto \textbf{let}\ newVals \Leftarrow (\!|\, \overrightarrow{V} \,|\!) +\!\!+ [v]\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(newVals, [\,])\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad \mathsf{body}\,(state') \\
\quad\quad\quad\ \ \textbf{inr}\ pid \mapsto \textbf{let}\ newReaders \Leftarrow [\,] +\!\!+ [pid]\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(\!|\, \overrightarrow{V} \,|\!), newReaders)\ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathsf{body}\,(state')\}) \\
\longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a, \textbf{let}\ newVals \Leftarrow (\!|\, \overrightarrow{V} \,|\!) +\!\!+ [(\!|\, W \,|\!)]\ \textbf{in}\ ,\varepsilon\rangle] \\
\quad\quad \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(newVals, [\,])\ \textbf{in} \\
\quad\quad \mathsf{body}\,(state')
\end{array}
$$

$$
\begin{array}{l}
\longrightarrow_{\mathsf{M}}^{*} \\
\mathcal{G}[\langle a, \textbf{let}\ newVals \Leftarrow (\!|\ \overrightarrow{V} \cdot W\ |\!)\ \textbf{in} \qquad\qquad\qquad , \varepsilon\rangle] \\
\quad\ \ \textbf{let}\ state' \Leftarrow \mathsf{drain}\,(newVals, [\,])\ \textbf{in} \\
\quad\ \ \mathsf{body}\,(state') \\
\longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a, \textbf{let}\ state' \Leftarrow \mathsf{drain}\,((\!|\ \overrightarrow{V} \cdot W\ |\!), [\,])\ \textbf{in}, \varepsilon\rangle] \\
\quad\ \ \mathsf{body}\,(state') \\
= (\text{expanding drain}) \\
\mathcal{G}[\langle a, \textbf{let}\ state' \Leftarrow \lambda x. \qquad\qquad\qquad\qquad\qquad\qquad\qquad , \varepsilon\rangle] \\
\qquad\qquad\qquad\qquad \textbf{let}\,(vals, pids) = x\ \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{case}\ vals\,\{ \\
\qquad\qquad\qquad\qquad\quad [\,] \mapsto \textbf{return}\,(vals, pids) \\
\qquad\qquad\qquad\qquad\quad v :: vs \mapsto \\
\qquad\qquad\qquad\qquad\qquad \textbf{case}\ pids\,\{ \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \mapsto \textbf{return}\,(vals, pids) \\
\qquad\qquad\qquad\qquad\qquad\qquad pid :: pids \mapsto \textbf{send}\ v\ pid; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}\,(vs, pids) \\
\qquad\qquad\qquad\qquad\qquad \}\quad \}\,(\!|\ \overrightarrow{V} \cdot W\ |\!), [\,])\ \textbf{in} \\
\quad\ \ \mathsf{body}\,(state') \\
\longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a, \textbf{let}\ state' \Leftarrow \textbf{let}\,(vals, pids) = (\!|\ \overrightarrow{V} \cdot W\ |\!), [\,])\ \textbf{in} \qquad , \varepsilon\rangle] \\
\qquad\qquad\qquad\qquad \textbf{case}\ vals\,\{ \\
\qquad\qquad\qquad\qquad\quad [\,] \mapsto \textbf{return}\,(vals, pids) \\
\qquad\qquad\qquad\qquad\quad v :: vs \mapsto \\
\qquad\qquad\qquad\qquad\qquad \textbf{case}\ pids\,\{ \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \mapsto \textbf{return}\,(vals, pids) \\
\qquad\qquad\qquad\qquad\qquad\qquad pid :: pids \mapsto \textbf{send}\ v\ pid; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}\,(vs, pids) \\
\qquad\qquad\qquad\qquad\qquad \}\quad \}\ \textbf{in} \\
\quad\ \ \mathsf{body}\,(state')
\end{array}
$$

$$
\begin{array}{l}
\longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a, \mathbf{let}\ state' \Leftarrow \mathbf{case}\ (\!|\ \overrightarrow{V} \cdot W\ |\!)\ \{ \qquad\qquad\qquad\qquad\qquad\qquad ,\varepsilon\rangle] \\
\qquad\qquad\qquad\qquad\qquad\qquad [\,] \mapsto \mathbf{return}\ ((\!|\ \overrightarrow{V} \cdot W\ |\!), [\,]) \\
\qquad\qquad\qquad\qquad\qquad\qquad v :: vs \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{case}\ [\,]\ \{ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad [\,] \mapsto \mathbf{return}\ ((\!|\ \overrightarrow{V} \cdot W\ |\!), [\,]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad pid :: pids \mapsto \mathbf{send}\ v\ pid; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{return}\ (vs, pids) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \}\quad \}\ \mathbf{in} \\
\qquad \mathsf{body}\ (state') \\
\longrightarrow_{\mathsf{M}} \longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a, \mathbf{let}\ state' \Leftarrow \mathbf{return}\ ((\!|\ \overrightarrow{V} \cdot W\ |\!), [\,])\ \mathbf{in}, \varepsilon\rangle] \\
\qquad \mathsf{body}\ (state') \\
\longrightarrow_{\mathsf{M}} \\
\mathcal{G}[\langle a, \mathsf{body}\ ((\!|\ \overrightarrow{V} \cdot W\ |\!), [\,]), \varepsilon\rangle] \\
= \\
(\nu b)(\langle b, (\!|E|\!)[\mathbf{return}\ ()], \varepsilon\rangle \parallel \langle a, \mathsf{body}\ ((\!|\ \overrightarrow{V} \cdot W\ |\!), [\,]), \varepsilon\rangle) \\
\equiv \\
(\nu b)(\langle b, (\!|E|\!)[\mathbf{return}\ ()], \varepsilon\rangle) \parallel \langle a, \mathsf{body}\ ((\!|\ \overrightarrow{V} \cdot W\ |\!), [\,]), \varepsilon\rangle \\
= \\
(\!|E[\mathbf{return}\ ()] \parallel a((\!|\ \overrightarrow{V} \cdot W\ |\!))\ |\!)
\end{array}
$$

as required.

Case E-Take

$$
\begin{aligned}
&\text{Assumption } E[\mathbf{take}\, a] \parallel a(W \cdot \overrightarrow{V}) \\
&\quad \text{Def.}(\!|-|\!)\ (\nu b)(\langle b, (\!|E|\!)[\mathbf{let}\ \mathit{selfPid} \Leftarrow \mathbf{self}\ \mathbf{in}, \varepsilon\rangle) \parallel \langle a, \mathsf{body}\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,]), \varepsilon\rangle \\
&\qquad\qquad \mathbf{send}\ (\mathbf{inr}\ \mathit{selfPid})\ a; \\
&\qquad\qquad \mathbf{receive}] \\
&\qquad \equiv (\nu b)(\langle b, (\!|E|\!)[\mathbf{let}\ \mathit{selfPid} \Leftarrow \mathbf{self}\ \mathbf{in}, \varepsilon\rangle \parallel \langle a, \mathsf{body}\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,]), \varepsilon\rangle) \\
&\qquad\qquad \mathbf{send}\ (\mathbf{inr}\ \mathit{selfPid})\ a; \\
&\qquad\qquad \mathbf{receive}] \\
&\longrightarrow (\text{E-Self})\ (\nu b)(\langle b, (\!|E|\!)[\mathbf{let}\ \mathit{selfPid} \Leftarrow \mathbf{return}\ b\ \mathbf{in}, \varepsilon\rangle \parallel \langle a, \mathsf{body}\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,]), \varepsilon\rangle) \\
&\qquad\qquad \mathbf{send}\ (\mathbf{inr}\ \mathit{selfPid})\ a; \\
&\qquad\qquad \mathbf{receive}] \\
&\longrightarrow_{\mathrm{M}} (\text{Let})\ (\nu b)(\langle a, (\!|E|\!)[\mathbf{send}\ (\mathbf{inr}\ b)\ a;, \varepsilon\rangle \parallel \langle a, \mathsf{body}\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,]), \varepsilon\rangle) \\
&\qquad\qquad \mathbf{receive}] \\
&\longrightarrow (\text{E-Send})\ (\nu b)(\langle b, (\!|E|\!)[\mathbf{return}\ (); \mathbf{receive}], \varepsilon\rangle \parallel \langle a, \mathsf{body}\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,]), (\mathbf{inr}\ b)\rangle) \\
&\longrightarrow_{\mathrm{M}} (\text{E-Let})\ (\nu b)(\langle b, (\!|E|\!)[\mathbf{receive}], \varepsilon\rangle \parallel \langle a, \mathsf{body}\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,]), (\mathbf{inr}\ b)\rangle) \\
&\qquad \equiv (\nu b)(\langle a, \mathsf{body}\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,]), (\mathbf{inr}\ b)\rangle \parallel \langle b, (\!|E|\!)[\mathbf{receive}], \varepsilon\rangle)
\end{aligned}
$$

Now, let $\mathcal{G}[-] = (\nu b)(\langle b, (\!|E|\!)[\mathbf{receive}], \varepsilon\rangle \parallel [-])$.

Expanding, we begin with:

$$
\begin{aligned}
&\mathcal{G}[\langle a,\ (\mathbf{rec}\ g(\mathit{state})\ . && ,(\mathbf{inr}\ b)\rangle] \\
&\quad \mathbf{let}\ \mathit{recvVal} \Leftarrow \mathbf{receive}\ \mathbf{in} \\
&\quad \mathbf{let}\ (\mathit{vals}, \mathit{readers}) = \mathit{state}\ \mathbf{in} \\
&\quad\ \ \mathbf{case}\ \mathit{recvVal}\,\{ \\
&\quad\ \ \mathbf{inl}\ v \mapsto \mathbf{let}\ \mathit{newVals} \Leftarrow \mathit{vals} +\!\!+ [v]\ \mathbf{in} \\
&\qquad\qquad\ \ \mathbf{let}\ \mathit{state}' \Leftarrow \mathsf{drain}\,(\mathit{newVals}, \mathit{readers})\ \mathbf{in} \\
&\qquad\qquad\ \ g\,(\mathit{state}') \\
&\quad\ \ \mathbf{inr}\ \mathit{pid} \mapsto \mathbf{let}\ \mathit{newReaders} \Leftarrow \mathit{readers} +\!\!+ [\mathit{pid}]\ \mathbf{in} \\
&\qquad\qquad\ \ \mathbf{let}\ \mathit{state}' \Leftarrow \mathsf{drain}\,(\mathit{vals}, \mathit{newReaders})\ \mathbf{in} \\
&\qquad\qquad\ \ g\,(\mathit{state'})\})\ ((\!|W|\!) :: (\!|\overrightarrow{V}|\!)), [\,])
\end{aligned}
$$

Reducing the recursive function, receiving from the mailbox, splitting the pair, and then taking the second branch on the case statement, we have:

$$
\begin{aligned}
&\mathcal{G}[\langle a,\ \mathbf{let}\ \mathit{newReaders} \Leftarrow [\,] +\!\!+ [b]\ \mathbf{in} && ,\varepsilon\rangle] \\
&\quad \mathbf{let}\ \mathit{state}' \Leftarrow \mathsf{drain}\,(\mathit{vals}, \mathit{newReaders})\ \mathbf{in} \\
&\quad \mathsf{body}\ \mathit{state}'
\end{aligned}
$$

Reducing the list append operation, expanding drain, and re-expanding $\mathcal{G}$, we have:

$$
\begin{array}{l}
(\nu b)(\langle b, E[\textbf{receive}], \varepsilon\rangle \parallel \langle a, \textbf{let}\ state' \Leftarrow (\lambda x. \\
\quad \textbf{let}\ (vals, readers) = x\ \textbf{in} \\
\quad \textbf{case}\ vals\ \{ \\
\qquad [\,] \mapsto \textbf{return}\ (vals, readers) \\
\qquad v :: vs \mapsto \\
\qquad\quad \textbf{case}\ readers\ \{ \\
\qquad\qquad [\,] \mapsto \textbf{return}\ (vals, readers) \\
\qquad\qquad pid :: pids \mapsto \textbf{send}\ v\ pid; \\
\qquad\qquad\quad \textbf{return}\ (vs, pids)\}\})\ ((\!| W |\!) :: (\!| \overrightarrow{V} |\!), [b])\ \textbf{in} \\
\quad \text{body}\ state' \qquad , \varepsilon\rangle
\end{array}
$$

Next, we reduce the function application, pair split, and the case statements:

$$
\begin{array}{l}
(\nu b)(\langle b, E[\textbf{receive}], \varepsilon\rangle \parallel \langle a, \textbf{let}\ state' \Leftarrow \textbf{send}\ (\!| W |\!)\ b; \\
\qquad \textbf{return}\ ((\!| \overrightarrow{V} |\!), [\,])\ \textbf{in} \\
\quad \text{body}\ state' \qquad , \varepsilon\rangle
\end{array}
$$

We next perform the send operation, and thus we have:

$$
(\nu b)(\langle b, E[\textbf{receive}], (\!| W |\!)\rangle \parallel \langle a, \text{body}\ (((\!| \overrightarrow{V} |\!), [\,]), \varepsilon\rangle
$$

Finally, we perform the **receive** and apply an equivalence to arrive at

$$
(\nu b)(\langle b, E[(\!| W |\!)], \varepsilon\rangle) \parallel \langle a, \text{body}\ (((\!| \overrightarrow{V} |\!), [\,]), \varepsilon\rangle
$$

which is equal to

$$
(\!| E[\textbf{return}\ W] \parallel a(\overrightarrow{V}) |\!)
$$

as required.

Case E-NewCh

$$
\begin{array}{rl}
\text{Assumption} & E[\textbf{newCh}] \\
\text{Definition of } (\!| - |\!) & (\nu a)(\langle a, (\!| E |\!)[\textbf{spawn}\ (\text{body}\ ([\,],[\,]))], \varepsilon\rangle) \\
\longrightarrow \text{(E-Spawn)} & (\nu a)(\nu b)(\langle a, (\!| E |\!)[\textbf{return}\ b], \varepsilon\rangle \parallel \langle b, \text{body}\ ([\,],[\,]), \varepsilon\rangle) \\
\equiv & (\nu b)(\nu a)(\langle a, (\!| E |\!)[\textbf{return}\ b], \varepsilon\rangle) \parallel \langle b, \text{body}\ ([\,],[\,]), \varepsilon\rangle) \\
= & (\!| (\nu b)(E[\textbf{return}\ b] \parallel b(\varepsilon)) |\!)
\end{array}
$$

as required.

Case E-Fork

$$
\begin{aligned}
&\text{Assumption } E[\mathbf{fork}\, M] \\
&\quad \text{Def. } (\!|-|\!)\ (\nu a)(\langle a, (\!|E|\!)[\mathbf{let}\, x \Leftarrow \mathbf{spawn}\, (\!|M|\!)\, \mathbf{in}\, \mathbf{return}\, ()], \varepsilon\rangle) \\
&\longrightarrow (\text{E-Spawn})\ (\nu a)(\nu b)(\langle a, (\!|E|\!)[\mathbf{let}\, x \Leftarrow \mathbf{return}\, b\, \mathbf{in}\, \mathbf{return}\, ()], \varepsilon\rangle \parallel \langle a, (\!|M|\!), \varepsilon\rangle) \\
&\quad \longrightarrow_{\mathrm{M}} (\nu a)(\nu b)(\langle a, (\!|E|\!)[\mathbf{return}\, ()], \varepsilon\rangle \parallel \langle b, (\!|M|\!), \varepsilon\rangle) \\
&\quad\quad \equiv (\nu a)(\langle a, (\!|E|\!)[\mathbf{return}\, ()], \varepsilon\rangle) \parallel (\nu b)(\langle b, (\!|M|\!), \varepsilon\rangle) \\
&\quad\quad = (\!|E[\mathbf{return}\, ()] \parallel M|\!)
\end{aligned}
$$

as required.

Case **Lift** Immediate by the induction hypothesis.

Case **LiftV** Immediate by Lemma 35.

Reflection: Suppose reduction happens in an actor a_i emulating a buffer $a_i(\overrightarrow{V_i})$. Here, we have that a β-reduction takes place, and reduction becomes blocked on **receive**. This is β-equivalent to the original translation, as required.

Thus, we need only consider the case where reduction occurs in an actor emulating a thread. Without loss of generality, assume reduction occurs in actor b_1. We may then proceed by case analysis on the structure of $(\!|M_1|\!)$. We show the case for $(\!|\mathbf{give}\, V\, W|\!)$ here. The remaining cases follow the same pattern.

Case $(\!|\mathbf{give}\, V\, W|\!)$

For reduction to occur, we have that W must be some ν-bound name a_i. Let us assume that this is a_1. Thus, we have that there exists some $\mathcal{D}$ such that $\mathcal{C}_{\text{canon}} \equiv \mathcal{D}$, where

$$\mathcal{D} = \mathcal{G}[\langle b_1, (\!|E|\!)[\mathbf{send}\, (\!|V|\!)\, a_1], \varepsilon\rangle \parallel \langle a_1, \mathsf{body}\, (\!|\overrightarrow{V_1}|\!), [\,]), \varepsilon\rangle]$$

By constructing the same derivation as for the simulation case, we have that $\mathcal{D} \Longrightarrow^{+} \mathcal{D}'$, where

$$\mathcal{D}' = \mathcal{G}[\langle b_1, (\!|E|\!)[\mathbf{return}\, ()], \varepsilon\rangle \parallel \langle a_1, \mathsf{body}\, (\!|\overrightarrow{V_1} \cdot V|\!), [\,]), \varepsilon\rangle]$$

In λ_{ch}, we can show

$$
\begin{aligned}
&E[\mathbf{give}\, V\, a_1] \parallel a_1(\overrightarrow{V_1}) \\
&\quad \longrightarrow (\text{E-Give}) \\
&E[\mathbf{return}\, ()] \parallel a_1(\overrightarrow{V_1} \cdot V)
\end{aligned}
$$

Thus, we have that

$$
\begin{aligned}
(\!|\mathcal{C}_{\text{canon}}|\!) \Longrightarrow^{+}\ & (\nu a_{k+1}) \cdots (\nu a_n)((\nu b_1)(\langle b_1, (\!|E|\!)[\mathbf{return}\, ()], \varepsilon\rangle) \parallel \cdots \parallel (\nu b_n)(\langle b_n, (\!|M_n|\!), \varepsilon\rangle) \parallel \\
& \langle a_1, \mathsf{body}\, (\!|\overrightarrow{V_1} \cdot V|\!), [\,]), \varepsilon\rangle \parallel \cdots \parallel \langle a_n, \mathsf{body}\, (\!|\overrightarrow{V_n}|\!), [\,]), \varepsilon\rangle) \\
= (\!|&(\nu a_{k+1}) \cdots (\nu a_n)(E[\mathbf{return}\, ()] \parallel \cdots \parallel M_m \parallel a_1(\overrightarrow{V_1} \cdot V) \parallel \cdots \parallel a_n(\overrightarrow{V_n}))|\!)
\end{aligned}
$$

and

$$\mathcal{C}_{\text{canon}} \Longrightarrow (\nu a_{k+1}) \cdots (\nu a_n)(E[\textbf{return}\ ()] \parallel \cdots \parallel M_m \parallel a_1(\overrightarrow{V_1} \cdot V) \parallel \cdots \parallel a_n(\overrightarrow{V_n}))$$

as required.

□

B.5 Extensions

B.5.1 Translation (λ_{ch} with synchronisation into λ_{act})

Theorem 20: Operational Correspondence ($(\!|-|\!)$ with synchronisation)

Simulation *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$*, then* $(\!|\mathcal{C}_1|\!) \Longrightarrow^* (\!|\mathcal{C}_2|\!)$.

Reflection *If* $\Gamma;\Delta \vdash \mathcal{C}_1$ *and* $(\!|\mathcal{C}_1|\!) \Longrightarrow \mathcal{D}$*, then there exists some* $\mathcal{C}_2$ *such that* $\mathcal{C}_1 \Longrightarrow^* \mathcal{C}_2$ *and* $\mathcal{D} \Longrightarrow^* \mathcal{E}$*, where* $\mathcal{E} =_\beta (\!|\mathcal{C}_2|\!)$.

Proof. We show the simulation case for E-Take; the remaining cases are unchanged. The reflection argument is identical to that in the proof of Theorem 16, and the reflection case for E-Wait follows by constructing the same derivation as for the simulation case.

Case E-Take

$$E[\mathbf{take}\; a] \parallel a(W \cdot \overrightarrow{V})$$

Def. $(\!|-|\!)$

$$\begin{array}{l}(\nu b)(\langle b, (\!|E|\!)[\mathbf{let}\; \mathit{requestorPid} \Leftarrow \\ \quad\quad \mathbf{spawn}\; (\\ \quad\quad\quad \mathbf{let}\; \mathit{selfPid} \Leftarrow \mathbf{self}\; \mathbf{in} \\ \quad\quad\quad \mathbf{send}\; (\mathbf{inr}\; \mathit{selfPid})\; a; \\ \quad\quad\quad \mathbf{receive})\; \mathbf{in} \\ \quad\quad \mathbf{wait}\; \mathit{requestorPid} \quad], \epsilon\rangle) \parallel \langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \epsilon\rangle\end{array}$$

$\Longrightarrow$ (E-Spawn)

$$\begin{array}{l}(\nu b)(\nu c)(\langle b, (\!|E|\!)[\mathbf{let}\; \mathit{requestorPid} \Leftarrow \mathbf{return}\; c\; \mathbf{in}\; \mathbf{wait}\; \mathit{requestorPid}], \epsilon\rangle \parallel \langle c, \mathbf{let}\; \mathit{selfPid} \Leftarrow \mathbf{self}\; \mathbf{in} \\ \quad \mathbf{send}\; (\mathbf{inr}\; \mathit{selfPid})\; a; \\ \quad \mathbf{receive}\; \mathbf{in}, \epsilon\rangle) \parallel \langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \epsilon\rangle\end{array}$$

$\longrightarrow_{\mathrm{M}}$

$$\begin{array}{l}(\nu b)(\nu c)(\langle b, (\!|E|\!)[\mathbf{wait}\; c], \epsilon\rangle \parallel \langle c, \mathbf{let}\; \mathit{selfPid} \Leftarrow \mathbf{self}\; \mathbf{in} \\ \quad \mathbf{send}\; (\mathbf{inr}\; \mathit{selfPid})\; a; \\ \quad \mathbf{receive}, \epsilon\rangle) \parallel \langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \epsilon\rangle\end{array}$$

$\Longrightarrow$ (E-Self)

$$\begin{array}{l}(\nu b)(\nu c)(\langle b, (\!|E|\!)[\mathbf{wait}\; c], \epsilon\rangle \parallel \langle c, \mathbf{let}\; \mathit{selfPid} \Leftarrow \mathbf{return}\; c\; \mathbf{in} \\ \quad \mathbf{send}\; (\mathbf{inr}\; \mathit{selfPid})\; a; \\ \quad \mathbf{receive}, \epsilon\rangle) \parallel \langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \epsilon\rangle\end{array}$$

$\longrightarrow_{\mathrm{M}}$

$$\begin{array}{l}(\nu b)(\nu c)(\langle b, (\!|E|\!)[\mathbf{wait}\; c], \epsilon\rangle \parallel \langle c, \mathbf{send}\; (\mathbf{inr}\; c)\; a;, \\ \quad \mathbf{receive} \quad \epsilon\rangle \parallel \langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \epsilon\rangle)\end{array}$$

$\Longrightarrow$ (E-Send)

$$\begin{array}{l}(\nu b)(\nu c)(\langle b, (\!|E|\!)[\mathbf{wait}\; c], \epsilon\rangle \parallel \langle c, \mathbf{return}\; ();, \\ \quad \mathbf{receive} \quad \epsilon\rangle \parallel \langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \mathbf{inr}\; c\rangle)\end{array}$$

$\longrightarrow_{\mathrm{M}}$

$$(\nu b)(\nu c)(\langle b, (\!|E|\!)[\mathbf{wait}\; c], \epsilon\rangle \parallel \langle c, \mathbf{receive}, \epsilon\rangle \parallel \langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \mathbf{inr}\; c\rangle)$$

$\equiv$

$$(\nu b)(\nu c)(\langle a, \mathrm{body}\; ((\!|W|\!) :: (\!|\overrightarrow{V}|\!), [\,]), \mathbf{inr}\; c\rangle \parallel \langle b, (\!|E|\!)[\mathbf{wait}\; c], \epsilon\rangle \parallel \langle c, \mathbf{receive}, \epsilon\rangle)$$

Now, let $\mathcal{G}[-] = (\nu b)(\nu c)([-] \parallel \langle b, (\!|E|\!)[\mathbf{wait}\; c], \epsilon\rangle \parallel \langle c, \mathbf{receive}, \epsilon\rangle$.

Now, we observe the reduction of actor a.

$$
\begin{array}{l}
\mathcal{G}[\langle a, \quad (\mathbf{rec}\ g(state)\ . \\
\quad \mathbf{let}\ recvVal \Leftarrow \mathbf{receive}\ \mathbf{in} \\
\quad \mathbf{let}\ (vals, readers) = state\ \mathbf{in} \\
\quad\quad \mathbf{case}\ recvVal\ \{ \\
\quad\quad\quad \mathbf{inl}\ v \mapsto \mathbf{let}\ newVals \Leftarrow vals +\!\!+ [v]\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ state' \Leftarrow \mathrm{drain}\ (newVals, readers)\ \mathbf{in} \\
\quad\quad\quad\quad g\ (state') \\
\quad\quad\quad \mathbf{inr}\ pid \mapsto \mathbf{let}\ newReaders \Leftarrow readers +\!\!+ [pid]\ \mathbf{in} \\
\quad\quad\quad\quad \mathbf{let}\ state' \Leftarrow \mathrm{drain}\ (vals, newReaders)\ \mathbf{in} \\
\quad\quad\quad\quad g\ (state')\}) \\
\quad ((\!| W |\!) :: (\!| \overrightarrow{V} |\!), [\,]) \quad , (\mathbf{inr}\ c)\rangle]
\end{array}
$$

Reducing the recursive function, receiving from the mailbox, splitting the pair, and then taking the second branch on the case statement, we have:

$$
\begin{array}{l}
\mathcal{G}[\langle a, \ \mathbf{let}\ newReaders \Leftarrow [\,] +\!\!+ [c]\ \mathbf{in} \quad ,\epsilon\rangle] \\
\quad \mathbf{let}\ state' \Leftarrow \mathrm{drain}\ (vals, newReaders)\ \mathbf{in} \\
\quad \mathrm{body}\ state'
\end{array}
$$

Reducing the list append operation, expanding drain, and re-expanding $\mathcal{G}$, we have:

$$
\begin{array}{l}
(\nu b)(\nu c)(\langle a, \mathbf{let}\ state' \Leftarrow (\lambda x. \\
\quad \mathbf{let}\ (vals, readers) = x\ \mathbf{in} \\
\quad \mathbf{case}\ vals\ \{ \\
\quad\quad [\,] \mapsto \mathbf{return}\ (vals, readers) \\
\quad\quad v :: vs \mapsto \\
\quad\quad\quad \mathbf{case}\ readers\ \{ \\
\quad\quad\quad\quad [\,] \mapsto \mathbf{return}\ (vals, readers) \\
\quad\quad\quad\quad pid :: pids \mapsto \mathbf{send}\ v\ pid; \\
\quad\quad\quad\quad\quad \mathbf{return}\ (vs, pids)\}\} \\
\quad)\ ((\!| W |\!) :: (\!| \overrightarrow{V} |\!), [c])\ \mathbf{in} \\
\quad \mathrm{body}\ state' \quad ,\epsilon\rangle \parallel \langle b, (\!| E |\!)[\mathbf{wait}\ c], \epsilon\rangle \parallel \langle c, \mathbf{receive}, \epsilon\rangle
\end{array}
$$

Finally:

$$
\begin{array}{l}
(\nu b)(\nu c)(\langle a, \textbf{let}\ state' \Leftarrow \textbf{send}\ (\!|W|\!)\ c; \qquad ,\varepsilon\rangle \parallel \langle b, (\!|E|\!)[\textbf{wait}\ c],\varepsilon\rangle \parallel \langle c, \textbf{receive},\varepsilon\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}\ ((\!|\overrightarrow{V}|\!),[\,])) \\
\qquad\qquad\quad \text{body}\ state' \\
\quad \Longrightarrow \text{(E-Send)} \\
(\nu b)(\nu c)(\langle a, \text{body}\ (((\!|\overrightarrow{V}|\!),[\,])),\varepsilon\rangle \parallel \langle b, (\!|E|\!)[\textbf{wait}\ c],\varepsilon\rangle \parallel \langle c, \textbf{receive}, (\!|W|\!)\rangle) \\
\quad \Longrightarrow \text{(E-Receive)} \\
(\nu b)(\nu c)(\langle a, \text{body}\ (((\!|\overrightarrow{V}|\!),[\,])),\varepsilon\rangle \parallel \langle b, (\!|E|\!)[\textbf{wait}\ c],\varepsilon\rangle \parallel \langle c, \textbf{return}\ W,\varepsilon\rangle) \\
\quad \Longrightarrow \text{(E-Wait)} \\
(\nu b)(\nu c)(\langle a, \text{body}\ (((\!|\overrightarrow{V}|\!),[\,])),\varepsilon\rangle \parallel \langle b, (\!|E|\!)[\textbf{return}\ (\!|W|\!)],\varepsilon\rangle \parallel \langle c, \textbf{return}\ (\!|W|\!), (\!|W|\!)\rangle) \\
\quad \equiv \\
(\nu b)(\langle b, (\!|E|\!)[\textbf{return}\ (\!|W|\!)],\varepsilon\rangle) \parallel \langle a, \text{body}\ (((\!|\overrightarrow{V}|\!),[\,])),\varepsilon\rangle \\
\quad = \\
(\!|E[\textbf{return}\ W] \parallel a(\overrightarrow{V})|\!)
\end{array}
$$

as required.

□

B.5.2 Translation (λ_{act} with selective receive into λ_{act})

Lemma 42 *Suppose* $\Gamma \mid \langle \ell_i : A_i\rangle_i \vdash \textbf{receive}\ \{\overrightarrow{c}\}$ *and* $\Gamma \vdash V : \langle \ell_i : A_i\rangle_i$, *where* $V = \langle \ell = V'\rangle$.
If $\neg(\mathit{matchesAny}(\overrightarrow{c}, V))$, *then*

$$\textbf{case}\ \lfloor V \rfloor\ \{\mathit{branches}(\overrightarrow{c}, mb, \mathit{default})\} \longrightarrow_M^+ \mathit{default}\ \lfloor V \rfloor$$

Proof. For it to be the case that $\neg(\text{matchesAny}(\overrightarrow{c}, V))$, we must have that for all $c_i \in \overrightarrow{c}$ where $c_i = \langle \ell_i = x_i\rangle$ **when** $M_i \mapsto N_i$, that either $\ell_i \neq \ell$, or $\ell_i = \ell$, but $M_i\{V'/x_i\} \longrightarrow_{\mathsf{M}}^* \text{false}$.

Recall that:

$$\text{branches}(\overrightarrow{c}, mb, \mathit{default}) \triangleq \text{patBranches}(\overrightarrow{c}, mb, \mathit{default}) \cdot \text{defaultBranches}(\overrightarrow{c}, mb, \mathit{default})$$

If $\ell \neq \ell_i$ for all patterns in $\overrightarrow{c}$, then there will be no corresponding branch in patBranches, but there will be a branch $\langle \ell = x\rangle \mapsto \mathit{default}\,\langle \ell = x\rangle$ in defaultBranches, reducing to the required result $\mathit{default}\ \lfloor V \rfloor$.

On the other hand, suppose we have a set of clauses $\overrightarrow{c'} \subseteq \overrightarrow{c}$ such that for each $(\langle \ell' = x_j\rangle$ **when** $M_j \mapsto N_j)_j \in \overrightarrow{c'}$, it is the case that $\ell' = \ell$. Then, for all M_j, $M_j\{V'/x_j\} \longrightarrow_{\mathsf{M}}^* \text{false}$. By the definition of patBranches, the **case** statement will reduce to $\text{ifPats}(mb, \ell, x, \overrightarrow{c'}, \mathit{default})$ for some fresh x.

We must now show:

$$\text{ifPats}(mb, \ell, x, \overrightarrow{c'}, \mathit{default}) \longrightarrow_{\mathsf{M}}^* \mathit{default}\ \lfloor V \rfloor$$

This result can now be established by induction on the structure of $\overrightarrow{c'}$, by the definition of ifPats. If $\overrightarrow{c'} = \varepsilon$, then by definition we have $\mathit{default}\lfloor V \rfloor$. Now suppose $\overrightarrow{c'} = (\langle \ell = x \rangle \textbf{ when } M \mapsto N) \cdot \overrightarrow{c''}$, where $M\{V'/x\} \longrightarrow_M^* \text{false}$. By Lemma 41, $\lfloor M\{V'/x\} \rfloor\, mb \longrightarrow_M^* \textbf{return}\,(\text{false}, mb)$, and thus the **else** branch will be taken and we can conclude by the induction hypothesis. □

Lemma 43 *Suppose* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash \textbf{receive}\,\{\overrightarrow{c}\}$ *and* $\Gamma \vdash V : \langle \ell_i : A_i \rangle_i$, *where* $V = \langle \ell = V' \rangle$. *If* $\neg(\mathit{matchesAny}(\overrightarrow{c}, V))$, *then*

$$\mathit{find}(\overrightarrow{c})\,(mb_1, \lfloor V \rfloor :: \lfloor \overrightarrow{W} \rfloor) \longrightarrow_M^+ \mathit{find}(\overrightarrow{c})\,(mb_1 +\!\!+ [\lfloor V \rfloor], \lfloor \overrightarrow{W} \rfloor)$$

Proof. We begin with

$$\mathsf{find}(\overrightarrow{c})\,(mb_1, \lfloor V \rfloor :: \lfloor \overrightarrow{W} \rfloor)$$

By β-reducing the recursive function application, pair deconstruction, and case expression:

$$\begin{array}{l}\textbf{let } mb' \Leftarrow mb_1 +\!\!+ \lfloor \overrightarrow{W} \rfloor \textbf{ in} \\ \textbf{case } \lfloor V \rfloor \,\{\mathsf{branches}(\overrightarrow{c}, mb', \\ \quad \lambda y.\textbf{let } mb_1' \Leftarrow mb_1 +\!\!+ [y] \textbf{ in} \\ \qquad \mathsf{find}(\overrightarrow{c})\,(mb_1', \lfloor \overrightarrow{W} \rfloor))\}\end{array}$$

By reducing the **let** binding and applying Lemma 42, we arrive at

$$(\lambda y.\textbf{let } mb_1' = mb_1 +\!\!+ [y] \textbf{ in } \mathsf{find}(\overrightarrow{c})\,(mb_1', \lfloor \overrightarrow{W} \rfloor))\,\lfloor V \rfloor$$

Finally, reducing the function application and **let** binding, we arrive at

$$\mathsf{find}(\overrightarrow{c})\,(mb_1 +\!\!+ [\lfloor V \rfloor], \lfloor \overrightarrow{W} \rfloor)$$

as required. □

Lemma 44 *Suppose* $\Gamma \mid \langle \ell_i : A_i \rangle_i \vdash \textbf{receive}\,\{\overrightarrow{c}\}$.

Suppose:

- $\exists k, l. \forall i. i < k. \neg(\mathit{matchesAny}(\overrightarrow{c}, V_i)) \wedge \mathit{matches}(c_l, V_k) \wedge \forall j. j < l \Rightarrow \neg(\mathit{matches}(c_j, V_k))$.
- $\Gamma \vdash V_k : \langle \ell_i : A_i \rangle_i$.
- $V_k = \langle \ell_k = V_k' \rangle$

Then:

$$\textbf{case } \lfloor V_k \rfloor\,\{\mathit{branches}(\overrightarrow{c}, mb, \mathit{default})\} \longrightarrow_M^+ (\lfloor N_l \rfloor\, mb)\{\lfloor V_k' \rfloor / x_l\}$$

Proof. If $\mathsf{matches}(c_l, V_k)$, then c_l must be of the form $\{\langle \ell_k = x_l \rangle \textbf{ when } M_l \mapsto N_l\}$, with $M_l\{V_k'/x_l\} \longrightarrow_{\mathsf{M}}^* \mathsf{true}$. By the definition of patBranches, we know that there exists an ordered list $\overrightarrow{c_\ell}$ where $\overrightarrow{c_\ell}$ is a sublist of $\overrightarrow{c}$, and $\mathsf{label}(c_l) = \ell_k$. We must now show

$$\mathsf{ifPats}(mb, \ell, x, \overrightarrow{c_\ell}, \mathit{default}) \longrightarrow_{\mathsf{M}}^* (\lfloor N_l \rfloor\, mb)\{\lfloor V_k \rfloor / x_l\}$$

We proceed by induction on the structure of the sublist $[c_i \mid c_i \in \overrightarrow{c_\ell}, i \leq l]$.

The base case is that of a singleton list consisting of c_l. For $\mathsf{ifPats}(mb, \ell, y, c_l, \mathit{default})$, by Lemma 41, $\lfloor M_l \rfloor\, mb \longrightarrow_{\mathsf{M}} \textbf{return}\,(\mathsf{true}, mb)$. Reducing, we take the true branch of the if-statement, and arrive at $(\lfloor N_l \rfloor\, mb)\{\lfloor V_k \rfloor / x_l\}$ as required.

The inductive case considers some pattern $c_i \cdot \overrightarrow{c_\ell}'$ where $i < l$. Since $i < l \implies \neg(\mathsf{matches}(c_i, V_k))$ we have that $M_i \longrightarrow_{\mathsf{M}}^* \textbf{return}\,\mathsf{false}$ and by Lemma 41, we have that $\lfloor M \rfloor\, mb\{\lfloor V_k \rfloor / x_i\} \longrightarrow_{\mathsf{M}}^* \textbf{return}\,(\mathsf{false}, mb)$. We can thus take the **false** branch and conclude with the induction hypothesis. □

Lemma 45 *Suppose* $\Gamma; \Delta \vdash \langle a, \textbf{receive}\ \{\overrightarrow{c}\}, \overrightarrow{W} \cdot U \cdot \overrightarrow{W'} \rangle$ *and* $\neg \mathit{matchesAny}(\overrightarrow{c}, U)$. *Then:*

$$\langle a, E[\mathit{loop}(\overrightarrow{c})\ \lfloor \overrightarrow{W} \rfloor], \lfloor U \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle \longrightarrow^+ \langle a, E[\mathit{loop}(\overrightarrow{c})\ \lfloor \overrightarrow{W} \rfloor +\!\!+ [\lfloor U \rfloor]], \lfloor \overrightarrow{W'} \rfloor \rangle$$

Proof. We begin with

$$\langle a, E[\mathsf{loop}(\overrightarrow{c})\ \lfloor \overrightarrow{W} \rfloor], \lfloor U \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle$$

Expanding $\mathsf{loop}(\overrightarrow{c})$ and reducing the function application:

$$\begin{aligned} \langle a, E[&\textbf{let}\ x \Leftarrow \textbf{receive}\ \textbf{in} &&], \lfloor U \rfloor \cdot \lfloor \overrightarrow{W'} \rfloor \rangle \\ &\textbf{case}\ x\ \{\mathsf{branches}(\overrightarrow{c}, \lfloor \overrightarrow{W} \rfloor, \lambda y.\textbf{let}\ mb' \Leftarrow \lfloor \overrightarrow{W} \rfloor +\!\!+ [y]\ \textbf{in} \\ &&& \mathsf{loop}(\overrightarrow{c})\ mb'\}) \end{aligned}$$

Applying E-Receive and β-reducing:

$$\begin{aligned} \langle a, E[&\textbf{case}\ \lfloor U \rfloor\ \{\mathsf{branches}(\overrightarrow{c}, \lfloor \overrightarrow{W} \rfloor, \lambda y.\textbf{let}\ mb' \Leftarrow \lfloor \overrightarrow{W} \rfloor +\!\!+ [y]\ \textbf{in} &&], \lfloor \overrightarrow{W'} \rfloor \rangle \\ &&& \mathsf{loop}(\overrightarrow{c})\ mb'\}) \end{aligned}$$

By Lemma 42:

$$\begin{gathered} \langle a, E[(\lambda y.\textbf{let}\ mb' \Leftarrow \lfloor \overrightarrow{W} \rfloor +\!\!+ [y]\ \textbf{in}], \lfloor \overrightarrow{W'} \rfloor \rangle \\ \mathsf{loop}(\overrightarrow{c})\ mb')\ \lfloor U \rfloor \end{gathered}$$

β-reducing the function application and **let**-binding, we arrive at

$$\langle a, E[\mathsf{loop}(\overrightarrow{c})\ \lfloor \overrightarrow{W} \rfloor +\!\!+ [\lfloor U \rfloor]], \lfloor \overrightarrow{W'} \rfloor \rangle$$

as required. □

Appendix C

Proofs for Chapter 8 (Asynchronous GV)

C.1 Preservation

Theorem 25: Preservation (Configuration reduction) *If* $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$*, then there exist some* Γ',Δ' *such that* $\Gamma;\Delta \longrightarrow^{?} \Gamma';\Delta'$ *and* $\Gamma';\Delta' \vdash^{\phi} \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$. Where there is a choice of value for ϕ, we consider the case where $\phi = \bullet$; the cases where $\phi = \circ$ are similar.

Case E-Fork

Assumption:

$$\frac{\Gamma_1,\Gamma_2 \vdash E[\mathbf{fork}\,\lambda x.M] : A}{\Gamma_1,\Gamma_2;\cdot \vdash^{\bullet} \bullet E[\mathbf{fork}\,\lambda x.M]}$$

By Lemma 47:

$$\frac{\dfrac{\Gamma_2, x : S \vdash M : \mathsf{End}_!}{\Gamma_2 \vdash \lambda x.M : S \multimap \mathsf{End}_!}}{\Gamma_2 \vdash \mathbf{fork}\,\lambda x.M : \overline{S}}$$

By Lemma 46, $\Gamma_2, b : S \vdash M\{b/x\} : \mathsf{End}_!$, and by Lemma 48, $\Gamma_1, a : \overline{S} \vdash E[a] : A$.

Reconstructing:

$$\dfrac{\dfrac{\Gamma_1, a : \overline{S} \vdash E[a] : A}{\Gamma_1, a : \overline{S}; \cdot \vdash^\bullet \bullet E[a]} \quad \dfrac{\Gamma_2, b : S \vdash^\circ \circ M\{b/x\} \qquad \dfrac{S/\varepsilon = \overline{\overline{S}/\varepsilon} \quad \overline{\cdot \vdash \varepsilon : \varepsilon} \quad \overline{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : S, b : \overline{S} \vdash^\circ a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma_2; a : S, b : S^\sharp \vdash^\circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\dfrac{\Gamma_1, \Gamma_2; a : \overline{S}^\sharp, b : S^\sharp \vdash^\bullet \bullet E[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}{\dfrac{\Gamma_1, \Gamma_2; a : \overline{S}^\sharp \vdash^\bullet (\nu b)(\bullet E[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}{\Gamma_1, \Gamma_2; \cdot \vdash^\bullet (\nu a)(\nu b)(\bullet E[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}}}$$

Case E-Send

Assumption:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2 \vdash E[\mathbf{send}\, U\, a] : C}{\Gamma_1, \Gamma_2, a : S; \cdot \vdash^\bullet \bullet E[\mathbf{send}\, U\, a]} \quad \dfrac{\overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3, \Gamma_4; a : \overline{S}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S^\sharp, b : T \vdash^\bullet \bullet E[\mathbf{send}\, U\, a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma 47:

$$\dfrac{\Gamma_2 \vdash U : A \quad a : !A.S' \vdash a : !A.S'}{\Gamma_2, a : !A.S' \vdash \mathbf{send}\, U\, a : S'}$$

Thus, $S = !A.S'$, and $\overline{S} = ?A.\overline{S'}$. We may therefore refine our original derivation:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : !A.S' \vdash E[\mathbf{send}\, U\, a] : C}{\Gamma_1, \Gamma_2, a : !A.S'; \cdot \vdash^\bullet \bullet E[\mathbf{send}\, U\, a]} \quad \dfrac{?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3, \Gamma_4; a : ?A.\overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : !A.S'^\sharp, b : T \vdash^\bullet \bullet E[\mathbf{send}\, U\, a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Since $?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}}$ is defined, we have that $\overrightarrow{A} = \varepsilon$. By the definition of slicing, we have that $\overline{T} = !B_1.\cdots.!B_n.!A.\overline{S'}$ for each B_i. It follows that $\overline{S'}/\overrightarrow{A} = \overline{T}/\overrightarrow{B} \cdot A$.

By Lemma 48, we have $\Gamma_1, \Gamma_2, a : S' \vdash E[a] : C$.

Reconstructing:

$$\dfrac{\dfrac{\Gamma_1, a : S' \vdash E[a] : C}{\Gamma_1, a : S'; \cdot \vdash^\bullet \bullet E[a]} \quad \dfrac{\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B} \cdot A} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_2, \Gamma_4 \vdash \overrightarrow{W} \cdot U : \overrightarrow{B} \cdot A}{\Gamma_2, \Gamma_3, \Gamma_4; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U)}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T \vdash^\bullet \bullet E[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Finally, we must show environment reduction:

$$\dfrac{!A.S' \longrightarrow S'}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : !A.S'^\sharp, b : T \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T}$$

as required.

Case E-Receive

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S \vdash E[\mathbf{receive}\, a] : C}{\Gamma_1, a : S; \cdot \vdash^\bullet \bullet E[\mathbf{receive}\, a]} \quad \dfrac{\overline{S}/A \cdot \overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_2, \Gamma_3 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3, \Gamma_4; a : \overline{S}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S^\sharp, b : T \vdash^\bullet \bullet E[\mathbf{receive}\, a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma 47:

$$\dfrac{a : ?A.S' \vdash a : ?A.S'}{a : ?A.S' \vdash \mathbf{receive}\, a : (A \times S')}$$

Thus, we have that $S = ?A.S'$ and $\overline{S} = !A.\overline{S'}$, and we may therefore refine the original typing derivation:

$$\dfrac{\dfrac{\Gamma_1, a : ?A.S' \vdash E[\mathbf{receive}\, a] : C}{\Gamma_1, a : ?A.S'; \cdot \vdash^\bullet \bullet E[\mathbf{receive}\, a]} \quad \dfrac{!A.\overline{S'}/A \cdot \overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \dfrac{\Gamma_1 \vdash U : A \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A}}{\Gamma_2, \Gamma_3 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A}} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3, \Gamma_4; a : !A.\overline{S'}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : (?A.S')^\sharp, b : T \vdash^\bullet \bullet E[\mathbf{receive}\, a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma 48, we have $\Gamma_1, \Gamma_2, a : S' \vdash E[(U, a)] : C$ (that Γ_1, Γ_2 is defined follows from the fact that Γ_1 and Γ_2 are sub-environments of the original typing environment and are therefore necessarily disjoint).

By the definition of slicing, $!A.\overline{S'}/A \cdot \overrightarrow{A} = \overline{S'}/\overrightarrow{A}$.

Thus, recomposing:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : S' \vdash E[(U, a)] : C}{\Gamma_1, \Gamma_2, a : S'; \cdot \vdash^\bullet \bullet E[(U, a)]} \quad \dfrac{\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3, \Gamma_4; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T \vdash^\bullet \bullet E[(U, a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Finally, we must show environment reduction:

$$\dfrac{?A.S' \longrightarrow S'}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : ?A.S'^\sharp; b : T \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T}$$

as required.

Case E-Wait

Assumption:

$$\dfrac{\dfrac{\Gamma, a : S \vdash E[\mathbf{wait}\, a] : C}{\Gamma, a : S; \cdot \vdash^{\bullet} \bullet E[\mathbf{wait}\, a]} \quad \dfrac{\dfrac{b : \mathsf{End}_! \vdash b : \mathsf{End}_!}{b : \mathsf{End}_!; \cdot \vdash^{\circ} \circ b} \quad \dfrac{\overline{S}/\varepsilon = \overline{\mathsf{End}_?/\varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : \overline{S}, b : \mathsf{End}_? \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot; a : \overline{S}, b : \mathsf{End}_!^{\sharp} \vdash^{\circ} \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma; a : S^{\sharp}, b : \mathsf{End}_!^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{wait}\, a] \parallel \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}$$
$$\dfrac{}{\Gamma; a : S^{\sharp} \vdash^{\bullet} (\nu b)(\bullet E[\mathbf{wait}\, a] \parallel \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}$$
$$\dfrac{}{\Gamma; \cdot \vdash^{\bullet} (\nu a)(\nu b)(\bullet E[\mathbf{wait}\, a] \parallel \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}$$

By Lemma 47:

$$\dfrac{a : \mathsf{End}_? \vdash a : \mathsf{End}_?}{a : \mathsf{End}_? \vdash \mathbf{wait}\, a : \mathbf{1}}$$

Thus, we can refine our original typing derivation:

$$\dfrac{\dfrac{\Gamma, a : \mathsf{End}_? \vdash E[\mathbf{wait}\, a] : C}{\Gamma, a : \mathsf{End}_?; \cdot \vdash^{\bullet} \bullet E[\mathbf{wait}\, a]} \quad \dfrac{\dfrac{b : \mathsf{End}_! \vdash b : \mathsf{End}_!}{b : \mathsf{End}_!; \cdot \vdash^{\circ} \circ b} \quad \dfrac{\mathsf{End}_!/\varepsilon = \overline{\mathsf{End}_?/\varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : \mathsf{End}_!, b : \mathsf{End}_? \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot; a : \mathsf{End}_!, b : \mathsf{End}_!^{\sharp} \vdash^{\circ} \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma; a : \mathsf{End}_?^{\sharp}, b : \mathsf{End}_!^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{wait}\, a] \parallel \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}$$
$$\dfrac{}{\Gamma; a : \mathsf{End}_?^{\sharp} \vdash^{\bullet} (\nu b)(\bullet E[\mathbf{wait}\, a] \parallel \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}$$
$$\dfrac{}{\Gamma; \cdot \vdash^{\bullet} (\nu a)(\nu b)(\bullet E[\mathbf{wait}\, a] \parallel \circ b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}$$

By Lemma 48, we can show that $\Gamma \vdash E[()] : C$, and thus that

$$\dfrac{\Gamma \vdash E[()] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet E[()]}$$

as required.

Case E-LiftC

Assumptions:

- $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$
- $\mathcal{C} \longrightarrow \mathcal{D}$

Let $\mathbf{D}$ be a derivation of $\Gamma; \Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$. By Lemma 49, we have that there exists some $\mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding $\Gamma'; \Delta' \vdash^{\phi'} \mathcal{C}$, where the position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to that of the hole in $\mathcal{G}$.

By the IH, we have that there exists some $\Gamma''; \Delta''$ such that $\Gamma; \Delta \longrightarrow^{?} \Gamma''; \Delta''$ and $\Gamma''; \Delta'' \vdash^{\phi} \mathcal{D}$.

By Lemma 50, we have that there exist some $\Gamma''';\Delta'''$ such that $\Gamma;\Delta \longrightarrow^{?} \Gamma''';\Delta'''$ and $\Gamma''';\Delta''' \vdash^{\phi} \mathcal{G}[\mathcal{D}]$, as required.

Case E-LiftM

Assumptions:

$$\frac{\Gamma \vdash M : A}{\Gamma;\cdot \vdash^{\bullet} \bullet M}$$

and $M \longrightarrow_{\mathsf{M}} N$. By Lemma 51, we have that $\Gamma \vdash N : A$. Recomposing:

$$\frac{\Gamma \vdash N : A}{\Gamma;\cdot \vdash^{\bullet} \bullet N}$$

as required.

□

C.2 Progress

Lemma 56 *Suppose $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$, where $\mathcal{C}$ is in canonical form and $\mathcal{C} \not\Longrightarrow$. Then $\mathcal{C}$ satisfies open progress.*

Proof. By induction on the derivation of $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$. By the definition of canonical forms in AGV, we have three cases:

Case $\mathcal{C} = (\nu a)(\mathcal{A} \parallel \mathcal{D})$, with $a \in \mathsf{fn}(\mathcal{A})$, and where $\mathcal{D}$ is in canonical form

By assumption, we know that $\Psi;\Delta \vdash^{\bullet} (\nu a)(\mathcal{A} \parallel \mathcal{D})$.

This configuration is typeable by T-Nu, followed by either T-Connect$_1$ or T-Connect$_2$.

Subcase T-Connect$_1$

$$\frac{\dfrac{\Psi_1, a : S;\Delta_1 \vdash^{\circ} \mathcal{A} \quad \Psi_2;\Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{D}}{\Psi_1,\Psi_2;\Delta_1,\Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{D}}}{\Psi_1,\Psi_2;\Delta_1,\Delta_2 \vdash^{\bullet} (\nu a)(\mathcal{A} \parallel \mathcal{D})}$$

By the definition of auxiliary threads and inversion on the typing relation, we know that $\mathcal{A}$ is of one of the following forms:

1. $\circ M$, where $a \in \mathsf{fn}(M)$, and $\Psi_1, a : S \vdash M : \mathsf{End}_!$
2. $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \in \mathsf{fn}(\Delta_1)$ and $a \in \mathsf{fn}(\overrightarrow{V})$

3. $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \in \mathsf{fn}(\Delta_1)$ and $a \in \mathsf{fn}(\overrightarrow{W})$

(since $a \notin \mathsf{fn}(\Delta_1)$, it cannot be the case that a appears as a buffer endpoint). Items (2) and (3) satisfy (1)(a)(ii) in the definition of open progress.

Lemma 55 tells us that either there exists some M' such that $M \longrightarrow_{\mathsf{M}} M'$; that M is a value; or that M is a communication and concurrency construct. Since $\mathcal{C} \not\Longrightarrow$, we have that M is unable to reduce (as otherwise $\mathcal{C}$ could reduce by E-LiftM).

Since $a \in \mathsf{fn}(M)$, it could be the case that $\Psi_1 = \cdot$ and thus $a : \mathsf{End}_! \vdash \circ a$, satisfying (1)(a)(i).

Otherwise, we have that M has the form $E[N]$, where N is a communication / concurrency construct (i.e., $\mathbf{fork}\, V$, $\mathbf{send}\, V\, W$, $\mathbf{receive}\, V$, or $\mathbf{wait}\, V$). Of these, $\mathbf{fork}\, V$ can always reduce. By T-Send, W must have session type $!A.S$; since Ψ contains only runtime names, it must therefore be the case that W is a runtime name $b \in \mathsf{fn}(\Psi_1, a : S)$ and thus $\mathsf{ready}(b, \mathbf{send}\, V\, b)$, satisfying (1)(a)(i). A similar argument applies for T-Recv and T-Wait.

Subcase T-Connect$_2$

$$\frac{\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A} \quad \Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{D}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{D}}$$
$$\Psi_1, \Psi_2; \Delta_1, \Delta_2 \vdash^{\bullet} (\nu a)(\mathcal{A} \parallel \mathcal{D})$$

By the definition of auxiliary threads and inversion on the typing relation, we know that $\mathcal{A}$ is of the following forms:

- $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$, where $b \in \mathsf{fn}(\Delta_1)$
- $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$, where $b \in \mathsf{fn}(\Delta_1)$

(as $a \in \mathsf{fn}(\mathcal{A})$ and $a \in \mathsf{fn}(\Delta_1)$, it cannot be the case that $\mathcal{A}$ is a child thread, as these require empty runtime typing environments).

By the induction hypothesis, we know that $\mathcal{D}$ satisfies open progress; hence $(\nu a)(\mathcal{A} \parallel \mathcal{D})$ satisfies open progress.

Case $\mathcal{C} = \mathcal{A} \parallel \mathcal{M}$

There are again two subcases, based on whether the parallel composition arises as a result of T-Connect$_1$ or T-Connect$_2$.

Subcase T-Connect$_1$

$$\frac{\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A} \quad \Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

By inversion on the typing rules, we have that $\mathcal{A}$ may be:

- A child thread $\circ M$, where $a \in \mathsf{fn}(M)$
- A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \neq a$ and either $a \in \mathsf{fn}(\overrightarrow{V})$ or $a \in \mathsf{fn}(\overrightarrow{W})$

In the case of (1), by Lemma 55, we have that either M is a value; there exists N such that $M \longrightarrow_{\mathsf{M}} N$; or $M = E[N]$ for some E, N, where N is a communication / concurrency construct.

By T-Child, $\Psi_1, a : S \vdash M : \mathsf{End}_!$. Since $a \in \mathsf{fn}(M)$ and the only value with type $\mathsf{End}_!$ is a variable or runtime name, it could be the case that $\Psi_1 = \cdot$ and thus $M = a$, satisfying (1)(a)(i).

Otherwise, since $\mathcal{C} \not\Longrightarrow$, it cannot be the case that $M \longrightarrow_{\mathsf{M}} N$, since otherwise $\mathcal{C}$ could reduce. Thus, it must be the case that $M = E[N]$ where N is a communication and concurrency construct; by similar reasoning as above cases, it therefore must be the case that $\mathsf{ready}(b, M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$.

(2) and (3) satisfy the required conditions by definition.

Subcase T-Connect$_2$

$$\frac{\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}; \Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

Since the runtime typing environment $\Delta_1, a : \overline{S}$ is non-empty, it cannot be the case that $\mathcal{A}$ is a child thread. Thus, $\mathcal{A}$ must either be of the form:

1. $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$, where $a, b \in \mathsf{fn}(\Delta_1)$; or
2. $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$, where $a, b \in \mathsf{fn}(\Delta_1)$

which satisfy the required conditions by definition.

By the induction hypothesis, we know that $\mathcal{M}$ satisfies open progress; hence $\mathcal{A} \parallel \mathcal{M}$ satisfies open progress.

Case $\mathcal{C} = \bullet M$

By Lemma 55, we have that either M is a value V, which satisfies open progress, or that M is a concurrency construct (i.e., $\mathbf{fork}\, V$, $\mathbf{send}\, V\, W$, $\mathbf{receive}\, V$, or $\mathbf{wait}\, V$). Of these, $\mathbf{fork}\, V$ can always reduce. By T-Send, W must have session type $!A.S$; since Ψ contains only runtime names, it must therefore be the case that W is a runtime name $a \in \mathsf{fn}(\Psi)$ and thus $\mathsf{ready}(a, \mathbf{send}\, V\, a)$. A similar argument applies for T-Recv and T-Wait.

□

Appendix D

Proofs for Chapter 9 (Exceptional GV)

D.1 Preservation

In this section, we present proofs that typeability is preserved by configuration reduction.

D.1.1 Equivalence

We begin by describing the properties of configuration equivalence. As described in §9.3, typeability of configurations is *not* preserved by equivalence. Nonetheless, Lemma 67 shows that only the associativity of parallel composition may cause a configuration to be ill-typed.

Lemma 67. *If* $\Gamma;\Delta \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \equiv \mathcal{D}$*, where the derivation of* $\mathcal{C} \equiv \mathcal{D}$ *does not contain a use of the axiom for associativity, then* $\Gamma;\Delta \vdash^{\phi} \mathcal{D}$*.*

Proof. By induction on the derivation of $\mathcal{C} \equiv \mathcal{D}$, examining the equivalence in both directions to account for symmetry. We show that a typing derivation of the left-hand side of an equivalence rule implies the existence of the right-hand side, and vice versa.

That reflexivity, transitivity, and symmetry of the equivalence relation respect typing follows immediately because equality of typing derivations is an equivalence relation.

We make implicit use of the induction hypothesis.

Congruence rules

Case Name restriction

$$\frac{\mathcal{C} \equiv \mathcal{D}}{(\nu a)\mathcal{C} \equiv (\nu a)\mathcal{D}}$$

$$\frac{\Gamma;\Delta,a:S^\sharp \vdash^\phi \mathcal{C}}{\Gamma;\Delta \vdash^\phi (\nu a)\mathcal{C}} \quad \Longleftrightarrow \quad \frac{\Gamma;\Delta,a:S^\sharp \vdash^\phi \mathcal{D}}{\Gamma;\Delta \vdash^\phi (\nu a)\mathcal{D}}$$

Case Parallel Composition

$$\frac{\mathcal{C} \equiv \mathcal{D}}{\mathcal{C} \parallel \mathcal{E} \equiv \mathcal{D} \parallel \mathcal{E}}$$

There are three subcases, based on whether the parallel composition arises from T-Connect$_1$, T-Connect$_2$, or T-Mix.

Subcase T-Mix

$$\frac{\Gamma_1;\Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2;\Delta_2 \vdash^{\phi_2} \mathcal{E}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{E}} \quad \Longleftrightarrow \quad \frac{\Gamma_1;\Delta_1 \vdash^{\phi_1} \mathcal{D} \quad \Gamma_2;\Delta_2 \vdash^{\phi_2} \mathcal{E}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash^{\phi_1+\phi_2} \mathcal{D} \parallel \mathcal{E}}$$

Subcase T-Connect$_1$

$$\frac{\Gamma_1,a:S;\Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2;\Delta_2,a:\overline{S} \vdash^{\phi_2} \mathcal{E}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{E}} \quad \Longleftrightarrow \quad \frac{\Gamma_1,a:S;\Delta_1 \vdash^{\phi_1} \mathcal{D} \quad \Gamma_2;\Delta_2,a:\overline{S} \vdash^{\phi_2} \mathcal{E}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{D} \parallel \mathcal{E}}$$

Subcase T-Connect$_2$

$$\frac{\Gamma_1;\Delta_1,a:\overline{S} \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2,a:S;\Delta_2 \vdash^{\phi_2} \mathcal{E}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{E}} \quad \Longleftrightarrow \quad \frac{\Gamma_1;\Delta_1,a:\overline{S} \vdash^{\phi_1} \mathcal{D} \quad \Gamma_2,a:S;\Delta_2 \vdash^{\phi_2} \mathcal{E}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{D} \parallel \mathcal{E}}$$

Equivalence Axioms

Case $\mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C}$

There are three subcases, based on which rule is used for parallel composition.

Subcase T-Mix

$$\frac{\Gamma_1;\Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2;\Delta_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{D}} \quad \Longleftrightarrow \quad \frac{\Gamma_2;\Delta_2 \vdash^{\phi_2} \mathcal{D} \quad \Gamma_1;\Delta_1 \vdash^{\phi_1} \mathcal{C}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2 \vdash^{\phi_2+\phi_1} \mathcal{D} \parallel \mathcal{C}}$$

Subcase T-Connect$_1$

$$\frac{\Gamma_1,a:S;\Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2;\Delta_2,a:\overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{D}} \quad \Longleftrightarrow \quad \frac{\Gamma_2;\Delta_2,a:\overline{S} \vdash^{\phi_2} \mathcal{D} \quad \Gamma_1,a:S;\Delta_1 \vdash^{\phi_1} \mathcal{C}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp \vdash^{\phi_2+\phi_1} \mathcal{D} \parallel \mathcal{C}}$$

Subcase T-Connect$_2$

$$\frac{\Gamma_1;\Delta_1,a:\overline{S}\vdash^{\phi_1}\mathcal{C}\quad\Gamma_2,a:S;\Delta_2\vdash^{\phi_2}\mathcal{D}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp\vdash^{\phi_1+\phi_2}\mathcal{C}\parallel\mathcal{D}}\iff\frac{\Gamma_2,a:S;\Delta_2\vdash^{\phi_2}\mathcal{D}\quad\Gamma_1;\Delta_1,a:\overline{S}\vdash^{\phi_1}\mathcal{C}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp\vdash^{\phi_2+\phi_1}\mathcal{D}\parallel\mathcal{C}}$$

Case $\mathcal{C}\parallel(\nu a)\mathcal{D}\equiv(\nu a)(\mathcal{C}\parallel\mathcal{D})$ if $a\notin\mathsf{fn}(\mathcal{C})$

There are again three subcases based on which parallel composition rule is used. The exact rule does not affect the discussion, so without loss of generality we assume this is T-Mix.

$$\frac{\Gamma_1;\Delta_1\vdash^{\phi_1}\mathcal{C}\quad\dfrac{\Gamma_2;\Delta_2,a:S^\sharp\vdash^{\phi_2}\mathcal{D}}{\Gamma_2;\Delta_2\vdash^{\phi_2}(\nu a)\mathcal{D}}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2\vdash^{\phi_1+\phi_2}\mathcal{C}\parallel(\nu a)\mathcal{D}}\iff\frac{\dfrac{\Gamma_1;\Delta_1\vdash^{\phi_1}\mathcal{C}\quad\Gamma_2;\Delta_2,a:S^\sharp\vdash^{\phi_2}\mathcal{D}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp\vdash^{\phi_1+\phi_2}\mathcal{C}\parallel\mathcal{D}}}{\Gamma_1,\Gamma_2;\Delta_1,\Delta_2\vdash^{\phi_1+\phi_2}(\nu a)(\mathcal{C}\parallel\mathcal{D})}$$

In the left-to-right direction, that $\Gamma_1,\Gamma_2;\Delta_1,\Delta_2,a:S^\sharp$ is well-defined follows because $a\notin\mathsf{fn}(\mathcal{C})$.

Case $(\nu a)(\nu b)\mathcal{C}\equiv(\nu b)(\nu a)\mathcal{C}$

$$\frac{\dfrac{\Gamma;\Delta,a:S^\sharp,b:T^\sharp\vdash^{\phi}\mathcal{C}}{\Gamma;\Delta,a:S^\sharp\vdash^{\phi}(\nu b)\mathcal{C}}}{\Gamma;\Delta\vdash^{\phi}(\nu a)(\nu b)\mathcal{C}}\iff\frac{\dfrac{\Gamma;\Delta,b:T^\sharp,a:S^\sharp\vdash^{\phi}\mathcal{C}}{\Gamma;\Delta,b:T^\sharp\vdash^{\phi}(\nu a)\mathcal{C}}}{\Gamma;\Delta\vdash^{\phi}(\nu b)(\nu a)\mathcal{C}}$$

Case $a(\overrightarrow{V})\leftrightsquigarrow b(\overrightarrow{W})\equiv b(\overrightarrow{W})\leftrightsquigarrow a(\overrightarrow{V})$

$$\frac{S/\overrightarrow{A}=\overline{T/\overrightarrow{B}}\quad\Gamma_1\vdash\overrightarrow{V}:\overrightarrow{A}\quad\Gamma_2\vdash\overrightarrow{W}:\overrightarrow{B}}{\Gamma_1,\Gamma_2;a:S,b:T\vdash^{\circ}a(\overrightarrow{V})\leftrightsquigarrow b(\overrightarrow{W})}\iff\frac{T/\overrightarrow{B}=\overline{S/\overrightarrow{A}}\quad\Gamma_2\vdash\overrightarrow{W}:\overrightarrow{B}\quad\Gamma_1\vdash\overrightarrow{V}:\overrightarrow{A}}{\Gamma_1,\Gamma_2;a:S,b:T\vdash^{\circ}b(\overrightarrow{W})\leftrightsquigarrow a(\overrightarrow{V})}$$

The above holds because $S/\overrightarrow{A}=\overline{T/\overrightarrow{B}}\iff T/\overrightarrow{B}=\overline{S/\overrightarrow{A}}$:

$$\begin{aligned}&S/\overrightarrow{A}=\overline{T/\overrightarrow{B}}\\&\quad\iff\text{(duality)}\\&\overline{S/\overrightarrow{A}}=\overline{\overline{T/\overrightarrow{B}}}\\&\quad\iff\text{(duality is involutive)}\\&\overline{S/\overrightarrow{A}}=T/\overrightarrow{B}\\&\quad\iff\text{(equality is symmetric)}\\&T/\overrightarrow{B}=\overline{S/\overrightarrow{A}}\end{aligned}$$

Case $\circ()\parallel\mathcal{C}\equiv\mathcal{C}$

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash () : \mathbf{1}}}{\cdot;\cdot \vdash^{\circ} \circ()} \quad \Gamma;\Delta \vdash^{\phi} \mathcal{C}}{\Gamma;\Delta \vdash^{\phi} \circ() \parallel \mathcal{C}} \qquad \iff \Gamma;\Delta \vdash^{\phi} \mathcal{C}$$

Case $(\nu a)(\nu b)(\lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)) \parallel \mathcal{C} \equiv \mathcal{C}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a : S;\cdot \vdash^{\circ} \lightning a} \quad \dfrac{\dfrac{}{b : T;\cdot \vdash^{\circ} \lightning b} \quad \dfrac{\overline{S}/\varepsilon = \overline{\overline{T}/\varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot;a : \overline{S}, b : \overline{T} \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot;a : \overline{S}, b : T^{\sharp} \vdash^{\circ} \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot;a : S^{\sharp}, b : T^{\sharp} \vdash^{\circ} \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot;a : S^{\sharp} \vdash^{\circ} (\nu b)(\lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}}{\cdot;\cdot \vdash^{\circ} (\nu a)(\nu b)(\lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))} \quad \Gamma;\Delta \vdash^{\phi} \mathcal{C}}{\Gamma;\Delta \vdash^{\phi} (\nu a)(\nu b)(\lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)) \parallel \mathcal{C}} \qquad \iff \Gamma;\Delta \vdash^{\phi} \mathcal{C}$$

□

While it is true that re-associating parallel composition may cause a configuration to be ill-typed, Lemma 68 shows that it is always possible to re-associate parallel composition either directly, or by first commuting two subconfigurations.

Lemma 68 (Associativity)**.**

- *If* $\Gamma;\Delta \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$*, then either* $\Gamma;\Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$ *or* $\Gamma;\Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D}$*.*
- *If* $\Gamma;\Delta \vdash^{\phi} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$*, then either* $\Gamma;\Delta \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$ *or* $\Gamma;\Delta \vdash^{\phi} \mathcal{D} \parallel (\mathcal{C} \parallel \mathcal{E})$*.*

Proof. The cases where either parallel composition arises by T-Mix are unproblematic and can be re-associated without jeopardising typeability. Therefore, we concentrate on the cases where both compositions arise via T-Connect$_i$.

Case $\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$

By the assumption that $\Gamma;\Delta \vdash^{\phi} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})$ we have that $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$, and $\Delta = \Delta_1, \Delta_2, \Delta_3, a : S^{\sharp}, b : T^{\sharp}$, and $\phi = \phi_1 + \phi_2 + \phi_3$. There are 8 cases, based on whether $a, b \in \mathsf{fn}(\mathcal{D})$ or $a, b \in \mathsf{fn}(\mathcal{E})$ (it cannot be the case that $a, b \in \mathsf{fn}(\mathcal{C})$, as $\mathcal{C}$ only occurs under a single parallel composition), and the exact dualisation (i.e., whether composition happens via T-Connect$_1$ or T-Connect$_2$).

Of these, we are only interested in the cases where the sharing of the names differs, as opposed to the dualisation. Thus, we consider the following two cases, where both compositions occur using T-Connect$_1$:

1. $\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C}$, and $\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$
2. $\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C}$, and $\Gamma_2, b : T; \Delta_2 \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$

Subcase $a \in \mathsf{fn}(\mathcal{C}), a, b \in \mathcal{D}, b \in \mathcal{E}$

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})} \dfrac{\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \quad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}$$

As $\mathcal{D}$ contains both a and b, associativity does not alter the sharing of names and may be applied safely.

$$\dfrac{\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, b : T; \Delta_1, \Delta_2, a : S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{D}} \quad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}$$

Subcase $a \in \mathsf{fn}(\mathcal{C}); b \in \mathcal{D}; a, b \in \mathcal{E}$

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})} \dfrac{\Gamma_2, b : T; \Delta_2 \vdash^{\phi_2} \mathcal{D} \quad \Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}$$

Here, we may not apply associativity directly. But, we may first commute $\mathcal{D}$ and $\mathcal{E}$:

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{E} \parallel \mathcal{D}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{E} \parallel \mathcal{D})} \dfrac{\Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E} \quad \Gamma_2, b : T; \Delta_2 \vdash^{\phi_2} \mathcal{D}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{E} \parallel \mathcal{D}}$$

and from here we may safely re-associate to the left:

$$\dfrac{\dfrac{\Gamma_2, a : S; \Delta_2 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_3; \Delta_3, a : \overline{S}, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : S^\sharp, b : \overline{T} \vdash^{\phi_1+\phi_2} \mathcal{D} \parallel \mathcal{E}} \quad \Gamma_3, b : T; \Delta_3 \vdash^{\phi_3} \mathcal{D}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{E}) \parallel \mathcal{D}}$$

Case $(\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}$

1. $\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C}$, and $\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}$
2. $\Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} \mathcal{C}$, and $\Gamma_2; \Delta_2, b : \overline{T} \vdash^{\phi_2} \mathcal{D}$, and $\Gamma_3; \Delta_3, a : \overline{S} \vdash^{\phi_3} \mathcal{E}$

Subcase $a \in \mathcal{C}; a, b \in \mathcal{D}; b \in \mathcal{E}$

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, b : T; \Delta_1, \Delta_2, a : S^\sharp \vdash^{\phi_1+\phi_2} \mathcal{C} \parallel \mathcal{D}} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}$$

Applying associativity here does not make the configuration ill-typed, as $\mathcal{D}$ contains both names:

$$\dfrac{\Gamma_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{C} \qquad \dfrac{\Gamma_2, b : T; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \quad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_2, \Gamma_3; \Delta_2, \Delta_3, a : \overline{S}, b : T^\sharp \vdash^{\phi_2+\phi_3} \mathcal{D} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E})}$$

Subcase $a, b \in \mathcal{C}; a \in \mathcal{D}; b \in \mathcal{E}$

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_2, \Gamma_3, b : T; \Delta_2, \Delta_3, a : S^\sharp \vdash^{\phi_2+\phi_3} \mathcal{C} \parallel \mathcal{D}} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E}}$$

By commutativity:

$$\dfrac{\dfrac{\Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \quad \Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} \mathcal{C}}{\Gamma_2, \Gamma_3, b : T; \Delta_2, \Delta_3, a : S^\sharp \vdash^{\phi_2+\phi_1} \mathcal{D} \parallel \mathcal{C}} \qquad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} (\mathcal{D} \parallel \mathcal{C}) \parallel \mathcal{E}}$$

By associativity:

$$\dfrac{\Gamma_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D} \qquad \dfrac{\Gamma_1, a : S, b : T; \Delta_1 \vdash^{\phi_1} \mathcal{C} \quad \Gamma_3; \Delta_3, b : \overline{T} \vdash^{\phi_3} \mathcal{E}}{\Gamma_1, \Gamma_3, a : S; \Delta_1, \Delta_3, b : T^\sharp \vdash^{\phi_1+\phi_3} \mathcal{C} \parallel \mathcal{E}}}{\Gamma_1, \Gamma_2, \Gamma_3; \Delta_1, \Delta_2, \Delta_3, a : S^\sharp, b : T^\sharp \vdash^{\phi_1+\phi_2+\phi_3} \mathcal{D} \parallel (\mathcal{C} \parallel \mathcal{E})}$$

as required.

□

D.1.2 Configuration Reduction

We may now show that configuration reduction preserves typeability of configurations. We begin by stating some auxiliary results about substitution and evaluation contexts.

Typing of terms is preserved by substitution.

Lemma 69 (Substitution)**.** *If:*

1. $\Gamma_1 \vdash M : B$
2. $\Gamma_2, x : B \vdash N : A$
3. Γ_1, Γ_2 *is well-defined*

then $\Gamma_1, \Gamma_2 \vdash N\{M/x\} : A$.

Proof. By induction on the derivation of $\Gamma_2, x : B \vdash N : A$. □

Theorem 31 (Preservation (Configurations))
Assume Γ *only contains entries of the form* $a_i : S_i$.

If $\Gamma; \Delta \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{D}$, *then there exist* Γ', Δ' *such that* $\Gamma; \Delta \longrightarrow^{?} \Gamma'; \Delta'$ *and* $\Gamma'; \Delta' \vdash^{\phi} \mathcal{D}$.

Proof. By induction on the derivation of $\mathcal{C} \longrightarrow \mathcal{D}$. Where there is a choice of value for ϕ, we consider the case where $\phi = \bullet$; the cases where $\phi = \circ$ are similar.

Case E-Fork

Assumption:

$$\frac{\Gamma_1, \Gamma_2 \vdash \bullet E[\mathbf{fork}\, \lambda x.M] : A}{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} \bullet E[\mathbf{fork}\, \lambda x.M]}$$

By Lemma 57:

$$\frac{\dfrac{\Gamma_2, x : S \vdash M : \mathbf{1}}{\Gamma_2 \vdash \lambda x.M : S \multimap \mathbf{1}}}{\Gamma_2 \vdash \mathbf{fork}\, \lambda x.M : \overline{S}}$$

By Lemma 69, $\Gamma_2, b : S \vdash M\{b/x\} : \mathbf{1}$, and by Lemma 58, $\Gamma_1, a : \overline{S} \vdash E[a] : A$. As duality is involutive, $\overline{\overline{S}} = S$.

Reconstructing:

$$\dfrac{\dfrac{\Gamma_1, a : \overline{S} \vdash E[a] : A}{\Gamma_1, a : \overline{S}; \cdot \vdash^{\bullet} \bullet E[a]} \quad \dfrac{\Gamma_2, b : S \vdash^{\circ} \circ M\{b/x\} \qquad \dfrac{S/\varepsilon = \overline{\overline{S}/\varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : S, b : \overline{S} \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma_2; a : S, b : S^{\sharp} \vdash^{\circ} M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\dfrac{\Gamma_1, \Gamma_2; a : \overline{S}^{\sharp}, b : S^{\sharp} \vdash^{\bullet} \bullet E[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}{\dfrac{\Gamma_1, \Gamma_2; a : \overline{S}^{\sharp} \vdash^{\bullet} (\nu b)(\bullet E[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}{\Gamma_1, \Gamma_2; \cdot \vdash^{\bullet} (\nu a)(\nu b)(\bullet E[a] \parallel \circ M\{b/x\} \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}}$$

Case E-Send

Assumption:

$$\frac{\Gamma_1,\Gamma_2 \vdash E[\mathbf{send}\,U\,a] : C \qquad \overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1,\Gamma_2,a : S;\cdot \vdash^\bullet \bullet E[\mathbf{send}\,U\,a] \qquad \Gamma_3,\Gamma_4;a : \overline{S},b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$
$$\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : S^\sharp,b : T \vdash^\bullet \bullet E[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

By Lemma 57:

$$\frac{\Gamma_2 \vdash U : A \quad a : !A.S' \vdash a : !A.S'}{\Gamma_2,a : !A.S' \vdash \mathbf{send}\,U\,a : S'}$$

Thus, $S = !A.S'$, and $\overline{S} = ?A.\overline{S'}$. We may therefore refine our original derivation:

$$\frac{\Gamma_1,\Gamma_2,a : !A.S' \vdash E[\mathbf{send}\,U\,a] : C \qquad ?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1,\Gamma_2,a : !A.S';\cdot \vdash^\bullet \bullet E[\mathbf{send}\,U\,a] \qquad \Gamma_3,\Gamma_4;a : ?A.\overline{S'},b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$
$$\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : !A.S'^\sharp,b : T \vdash^\bullet \bullet E[\mathbf{send}\,U\,a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

Since $?A.\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}}$ is well-defined, we have that $\overrightarrow{A} = \varepsilon$. By the definition of slicing, we have that $\overline{T} = \overline{!B_1.\cdots.!B_n.!A.S'}$, where $\overrightarrow{B} = B_1,\ldots,B_n$. It follows that $\overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}\cdot A}$.

By Lemma 58, we have $\Gamma_1,\Gamma_2,a : S' \vdash E[a] : C$.

Reconstructing:

$$\frac{\Gamma_1,a : S' \vdash E[a] : C \qquad \overline{S'}/\overrightarrow{A} = \overline{T/\overrightarrow{B}\cdot A} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_2,\Gamma_4 \vdash \overrightarrow{W}\cdot U : \overrightarrow{B}\cdot A}{\Gamma_1,a : S';\cdot \vdash^\bullet \bullet E[a] \qquad \Gamma_2,\Gamma_3,\Gamma_4;a : \overline{S'},b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}\cdot U)}$$
$$\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : S'^\sharp,b : T \vdash^\bullet \bullet E[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}\cdot U)$$

Finally, we must show environment reduction:

$$\frac{!A.S' \longrightarrow S'}{\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : (!A.S')^\sharp,b : T \longrightarrow \Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : S'^\sharp,b : T}$$

as required.

Case E-Receive

Assumption:

$$\frac{\Gamma_1,a : S \vdash E[\mathbf{receive}\,a] : C \qquad \overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_2,\Gamma_3 \vdash U\cdot\overrightarrow{V} : \overrightarrow{A} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1,a : S;\cdot \vdash^\bullet E[\mathbf{receive}\,a] \qquad \Gamma_2,\Gamma_3,\Gamma_4;a : \overline{S},b : T \vdash^\circ a(U\cdot\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$
$$\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4;a : S^\sharp,b : T \vdash^\bullet \bullet E[\mathbf{receive}\,a] \parallel a(U\cdot\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

By Lemma 57:

$$\frac{a : ?A.S' \vdash a : ?A.S'}{a : ?A.S' \vdash \mathbf{receive}\,a : (A \times S')}$$

Thus, we have that $S = ?A.S'$ and $\overline{S} = !A.\overline{S'}$, and we may therefore refine the original typing derivation:

$$\dfrac{\dfrac{\Gamma_1, a : ?A.S' \vdash E[\mathbf{receive}\, a] : C}{\Gamma_1, a : ?A.S'; \cdot \vdash^\bullet E[\mathbf{receive}\, a]} \quad \dfrac{!A.\overline{S'}/A \cdot \overrightarrow{A'} = \overline{T/B} \quad \dfrac{\Gamma_1 \vdash U : A \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A'}}{\Gamma_2, \Gamma_3 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A'}} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2, \Gamma_3, \Gamma_4; a : !A.\overline{S'}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : (?A.S')^\sharp, b : T \vdash^\bullet \bullet E[\mathbf{receive}\, a] \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma 58, we have $\Gamma_1, \Gamma_2, a : S' \vdash E[(U, a)] : C$ (that Γ_1, Γ_2 is defined follows from the fact that Γ_1 and Γ_2 are sub-environments of the original typing environment and are therefore necessarily disjoint).

By the definition of slicing, $!A.\overline{S'}/A \cdot \overrightarrow{A'} \iff \overline{S'}/\overrightarrow{A'}$.

Thus, recomposing:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_2, a : S' \vdash E[(U, a)] : C}{\Gamma_1, \Gamma_2, a : S'; \cdot \vdash^\bullet E[(U, a)]} \quad \dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/B} \quad \Gamma_3 \vdash \overrightarrow{V} : \overrightarrow{A'} \quad \Gamma_4 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_3, \Gamma_4; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T \vdash^\bullet \bullet E[(U, a)] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Finally, we must show environment reduction:

$$\dfrac{?A.S' \longrightarrow S'}{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : (?A.S'^\sharp); b : T \longrightarrow \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4; a : S'^\sharp, b : T}$$

as required.

Case E-Close

Assumption:

$$\dfrac{\dfrac{\dfrac{\Gamma_1, a : S \vdash E[\mathbf{close}\, a] : C}{\Gamma_1, a : S; \cdot \vdash^\bullet \bullet E[\mathbf{close}\, a]} \quad \dfrac{\dfrac{\Gamma_2, b : T \vdash E'[\mathbf{close}\, b] : \mathbf{1}}{\Gamma_2, b : T; \cdot \vdash^\circ \circ E'[\mathbf{close}\, b]} \quad \dfrac{\overline{S}/\varepsilon = \overline{\overline{T}/\varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : \overline{S}, b : \overline{T} \vdash^\circ a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma_2; a : \overline{S}, b : T^\sharp \vdash^\circ E'[\mathbf{close}\, b] \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma_1, \Gamma_2; a : S^\sharp, b : T^\sharp \vdash^\bullet \bullet E[\mathbf{close}\, a] \parallel \circ E'[\mathbf{close}\, b] \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma_1, \Gamma_2; a : S^\sharp \vdash^\bullet (\nu b)(\bullet E[\mathbf{close}\, a] \parallel \circ E'[\mathbf{close}\, b] \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}}{\Gamma_1, \Gamma_2; \cdot \vdash^\bullet (\nu a)(\nu b)(\bullet E[\mathbf{close}\, a] \parallel \circ E'[\mathbf{close}\, b] \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon))}$$

By Lemma 57:

$$\dfrac{a : \mathsf{End} \vdash a : \mathsf{End}}{a : \mathsf{End} \vdash \mathbf{close}\, a : \mathbf{1}} \qquad \dfrac{b : \mathsf{End} \vdash b : \mathsf{End}}{b : \mathsf{End} \vdash \mathbf{close}\, b : \mathbf{1}}$$

By Lemma 58, we have that $\Gamma_1 \vdash E[()] : C$ and that $\Gamma_2 \vdash E'[()] : \mathbf{1}$. Thus by T-Mix, we may show:

$$\dfrac{\dfrac{\Gamma_1 \vdash E[()] : C}{\Gamma_1;\cdot \vdash^\bullet \bullet E[()]} \quad \dfrac{\Gamma_2 \vdash E[()] : \mathbf{1}}{\Gamma_2;\cdot \vdash^\circ \circ E[()]}}{\Gamma_1,\Gamma_2;\cdot \vdash^\bullet \bullet E[()] \parallel \circ E[()]}$$

as required.

Case E-Cancel

$$\mathcal{F}[\mathbf{cancel}\, a] \longrightarrow \mathcal{F}[()] \parallel \lightning a$$

Assumption:

$$\dfrac{\Gamma \vdash E[\mathbf{cancel}\, a] : C}{\Gamma;\cdot \vdash^\bullet \bullet E[\mathbf{cancel}\, a]}$$

By Lemma 57, $\Gamma = \Gamma_1,\Gamma_2$, where

$$\dfrac{\Gamma_2 \vdash a : S}{\Gamma_2 \vdash \mathbf{cancel}\, a : \mathbf{1}}$$

Thus $\Gamma_2 = a : S$. By Lemma 58, $\Gamma_1 \vdash E[()] : C$. By T-Zap, we have that $a : S \vdash^\circ \lightning a$. Thus, recomposing:

$$\dfrac{\dfrac{\Gamma \vdash E[()] : C}{\Gamma_1;\cdot \vdash^\bullet \bullet E[()]} \quad \dfrac{}{a : S;\cdot \vdash^\circ \lightning a}}{\Gamma_1, a : S;\cdot \vdash^\bullet \bullet E[()] \parallel \lightning a}$$

as required.

Case E-Zap

$$\lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \lightning a \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

where $\mathsf{fn}(U) = \{c_i\}_i$.

Assumption:

$$\dfrac{\dfrac{}{a : S;\cdot \vdash^\circ \lightning a} \quad \dfrac{\overline{S}/\overrightarrow{A} = \overline{T/\overrightarrow{B}} \quad \Gamma_1,\Gamma_2 \vdash U \cdot \overrightarrow{V} : \overrightarrow{A} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1,\Gamma_2,\Gamma_3; a : \overline{S}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1,\Gamma_2,\Gamma_3; a : S^\sharp, b : T \vdash^\circ \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By the definition of slicing, we have that there exist some A and S' such that $\overline{S} = !A.\overline{S'}$. Thus, we may refine our judgement:

$$\dfrac{\dfrac{}{a : ?A.S';\cdot \vdash^\circ \lightning a} \quad \dfrac{!A.\overline{S'}/A \cdot \overrightarrow{A'} = \overline{T/B} \quad \Gamma_1,\Gamma_2 \vdash U \cdot \overrightarrow{V} : A \cdot \overrightarrow{A'} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_1,\Gamma_2,\Gamma_3; a : !A.\overline{S'}, b : T \vdash^\circ a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1,\Gamma_2,\Gamma_3; a : (?A.S')^\sharp, b : T \vdash^\circ \lightning a \parallel a(U \cdot \overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By the definition of buffer typing, we have that $\Gamma_1 \vdash U : A$. By the definition of the reduction rule, $\mathsf{fn}(U) = \{c_i\}_i$, and by assumption, Γ contains only runtime names. Thus, we may conclude that U is closed and therefore that $\Gamma_1 = c_1 : S_1, \ldots c_n : S_n$ for some session types $S_1, \ldots S_n$.

By the definition of slicing, we have that $!A.\overline{S'}/A \cdot \overrightarrow{A'} \iff \overline{S'}/\overrightarrow{A'}$. Correspondingly, by T-Buffer, we may show

$$\dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/B} \quad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A'} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2,\Gamma_3; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

By repeated applications of T-Zap and T-Mix, we have that

$$\Gamma_2,\Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : \overline{S'}, b : T \vdash^\circ \lightning c_1 \parallel \cdots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$$

Recomposing:

$$\dfrac{\dfrac{}{a : S';\cdot \vdash^\circ \lightning a} \quad \dfrac{\dfrac{}{c_1 : S_1;\cdot \vdash^\circ \lightning c_1} \quad \dfrac{\dfrac{}{c_n : S_n;\cdot \vdash^\circ \lightning c_n} \quad \dfrac{\overline{S'}/\overrightarrow{A'} = \overline{T/B} \quad \Gamma_2 \vdash \overrightarrow{V} : \overrightarrow{A'} \quad \Gamma_3 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2,\Gamma_3; a : \overline{S'}, b : T \vdash^\circ a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\vdots}}{\Gamma_2,\Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : \overline{S'}, b : T \vdash^\circ \lightning c_1 \parallel \ldots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2,\Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : S'^\sharp, b : T \vdash^\circ \lightning a \parallel \lightning c_1 \parallel \ldots \parallel \lightning c_n \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})}$$

Finally, we must show environment reduction:

$$\dfrac{?A.S' \longrightarrow S'}{\Gamma_2,\Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : (?A.S'^\sharp), b : T \longrightarrow \Gamma_2,\Gamma_3, c_1 : S_1, \ldots, c_n : S_n; a : S'^\sharp, b : T}$$

as required.

Case E-CloseZap

$$\mathcal{F}[\mathbf{close}\, a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon) \longrightarrow \mathcal{F}[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)$$

Assumption:

$$\dfrac{\dfrac{\Gamma, a : S \vdash E[\mathbf{close}\, a] : C}{\Gamma, a : S; \cdot \vdash^{\bullet} \bullet E[\mathbf{close}\, a]} \quad \dfrac{\dfrac{}{b : T; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\overline{S} = \overline{\overline{T}} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : \overline{S}, b : \overline{T} \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot; a : \overline{S}, b : T^{\sharp} \vdash^{\circ} \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma; a : S^{\sharp}, b : T^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{close}\, a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}$$

By Lemma 57:

$$\dfrac{a : \mathsf{End} \vdash a : \mathsf{End}}{a : S \vdash \mathbf{close}\, a : \mathbf{1}}$$

We may therefore refine our original derivation:

$$\dfrac{\dfrac{\Gamma, a : \mathsf{End} \vdash E[\mathbf{close}\, a] : C}{\Gamma, a : \mathsf{End}; \cdot \vdash^{\bullet} \bullet E[\mathbf{close}\, a]} \quad \dfrac{\dfrac{}{b : \mathsf{End}; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\mathsf{End} = \mathsf{End} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : \mathsf{End}, b : \mathsf{End} \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot; a : \mathsf{End}, b : \mathsf{End}^{\sharp} \vdash^{\circ} \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma; a : \mathsf{End}^{\sharp}, b : \mathsf{End}^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{close}\, a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}$$

By Lemma 58, $\Gamma \vdash E[\mathbf{raise}] : C$.
Thus, recomposing:

$$\dfrac{\dfrac{\Gamma \vdash E[\mathbf{raise}] : C}{\Gamma \vdash^{\bullet} \bullet E[\mathbf{raise}]} \quad \dfrac{\dfrac{}{a : \mathsf{End}; \cdot \vdash^{\circ} \lightning a} \quad \dfrac{\dfrac{}{b : \mathsf{End}; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\mathsf{End} = \mathsf{End} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon}}{\cdot; a : \mathsf{End}, b : \mathsf{End} \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot; a : \mathsf{End}, b : \mathsf{End}^{\sharp} \vdash^{\circ} \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\cdot; a : \mathsf{End}^{\sharp}, b : \mathsf{End}^{\sharp} \vdash^{\circ} \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}}{\Gamma; a : \mathsf{End}^{\sharp}, b : \mathsf{End}^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{close}\, a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\varepsilon)}$$

as required.

Case E-ReceiveZap

$$\bullet E[\mathbf{receive}\, a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W}) \longrightarrow \bullet E[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})$$

Assumption:

$$\dfrac{\dfrac{\Gamma_1, a : S \vdash E[\mathbf{receive}\, a] : C}{\Gamma_1, a : S; \cdot \vdash^{\bullet} \bullet E[\mathbf{receive}\, a]} \quad \dfrac{\dfrac{}{b : T; \cdot \vdash^{\circ} \lightning b} \quad \dfrac{\overline{S}/\varepsilon = \overline{\overline{T}/\overrightarrow{B}} \quad \dfrac{}{\cdot \vdash \varepsilon : \varepsilon} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2; a : \overline{S}, b : \overline{T} \vdash^{\circ} a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2; a : \overline{S}, b : T^{\sharp} \vdash^{\circ} \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2; a : S^{\sharp}, b : T^{\sharp} \vdash^{\bullet} \bullet E[\mathbf{receive}\, a] \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})}$$

By Lemma 57:

$$\frac{a : ?A.S' \vdash a : ?A.S'}{a : ?A.S' \vdash \mathbf{receive}\, a : (A \times S')}$$

By Lemma 58, $\Gamma_1 \vdash E[\mathbf{raise}] : S'$. Thus, recomposing:

$$\dfrac{\dfrac{\Gamma_1 \vdash E[\mathbf{raise}] : C}{\Gamma_1; \cdot \vdash^\bullet \bullet E[\mathbf{raise}]} \quad \dfrac{\dfrac{a : S; \cdot \vdash \lightning a}{} \quad \dfrac{\dfrac{}{b : T; \cdot \vdash^\circ \lightning b} \quad \dfrac{\overline{S}/\varepsilon = \overline{T}/\overrightarrow{B} \quad \overline{\cdot \vdash \varepsilon : \varepsilon} \quad \Gamma_2 \vdash \overrightarrow{W} : \overrightarrow{B}}{\Gamma_2; a : \overline{S}, b : \overline{T} \vdash^\circ a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2; a : \overline{S}, b : T^\sharp \vdash^\circ \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_2; a : S^\sharp, b : T^\sharp \vdash^\circ \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})}}{\Gamma_1, \Gamma_2; a : S^\sharp, b : T^\sharp \vdash^\bullet \bullet E[\mathbf{raise}] \parallel \lightning a \parallel \lightning b \parallel a(\varepsilon) \leftrightsquigarrow b(\overrightarrow{W})}$$

as required.

Case E-Raise

$$\bullet E[\mathbf{try}\, P[\mathbf{raise}]\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N] \longrightarrow E[N] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n$$

and $\mathsf{fn}(P) = \{c_i\}_i$.

Assumption:

$$\frac{\Gamma \vdash E[\mathbf{try}\, P[\mathbf{raise}]\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N] : A'}{\Gamma; \cdot \vdash^\bullet \bullet E[\mathbf{try}\, P[\mathbf{raise}]\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N]}$$

By Lemma 57, there exist $\Gamma_1, \Gamma_2, A, B, C$ such that $\Gamma = \Gamma_1, \Gamma_2, \Gamma_3$ and

$$\frac{\Gamma_2 \vdash P[\mathbf{raise}] : A \quad \Gamma_3, x : B \vdash M : C \quad \Gamma_3 \vdash N : C}{\Gamma_2, \Gamma_3 \vdash \mathbf{try}\, P[\mathbf{raise}]\, \mathbf{as}\, x\, \mathbf{in}\, M\, \mathbf{otherwise}\, N : C}$$

Since Γ contains only runtime names and $\mathsf{fn}(P) = \{c_i\}_i$, we know that $\Gamma_2 = c_1 : S_1, \ldots, c_n : S_n$ for some S_i.

By Lemma 58, we have that:

$$\frac{}{\Gamma_1, \Gamma_3 \vdash E[N] : A'}$$

By repeated applications of T-Zap and T-Mix, we have that $\Gamma_2 \vdash \lightning c_1 \parallel \cdots \parallel \lightning c_n$. Therefore, recomposing:

$$\dfrac{\dfrac{\Gamma_1, \Gamma_3 \vdash E[N] : C}{\Gamma_1, \Gamma_3; \cdot \vdash^\bullet \bullet E[N]} \quad \dfrac{\dfrac{c_1 : S_1; \cdot \vdash^\circ \lightning c_1}{} \quad \dfrac{\dfrac{}{c_{n-1} : S_{n-1}; \cdot \vdash^\circ \lightning c_{n-1}} \quad \dfrac{}{c_n : S_n; \cdot \vdash^\circ \lightning c_n}}{\vdots}}{c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^\circ \lightning c_1 \parallel \cdots \parallel \lightning c_n}}{\Gamma_1, \Gamma_3, c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^\bullet \bullet E[N] \parallel \lightning c_1 \parallel \cdots \parallel \lightning c_n}$$

as required.

Case E-RaiseChild

$$\circ P[\mathbf{raise}] \longrightarrow \lightning\, c_1 \parallel \cdots \parallel \lightning\, c_n$$

Assumption:

$$\frac{\Gamma \vdash P[\mathbf{raise}] : \mathbf{1}}{\Gamma; \cdot \vdash^{\circ} \circ P[\mathbf{raise}]}$$

By Lemma 57, the knowledge that Γ contains only runtime names, the knowledge that $\mathsf{fn}(P) = c_1, \ldots, c_n$, and the typing rule T-Raise, we have that $\Gamma = c_1 : S_1, \ldots, c_n : S_n$ for some session types $\{S_i\}_i$.

Thus, by repeated applications of T-Zap and T-Mix, we may deduce that

$$\Gamma; \cdot \vdash^{\circ} \lightning\, c_1 \parallel \cdots \parallel \lightning\, c_n$$

as required.

Case E-RaiseMain

$$\bullet P[\mathbf{raise}] \longrightarrow \mathbf{halt} \parallel \lightning\, c_1 \parallel \cdots \parallel \lightning\, c_n$$

where $\mathsf{fn}(P) = \{c_i\}_i$.

Assumption:

$$\frac{\Gamma \vdash P[\mathbf{raise}] : C}{\Gamma; \cdot \vdash^{\bullet} \bullet P[\mathbf{raise}]}$$

By Lemma 57, the knowledge that Γ contains only runtime names, the knowledge that $\mathsf{fn}(P) = c_1, \ldots, c_n$, and the typing rule T-Raise, we have that $\Gamma = c_1 : S_1, \ldots, c_n : S_n$ for some session types $\{S_i\}_i$.

By repeated applications of T-Zap and T-Mix, we may deduce that

$$\Gamma; \cdot \vdash^{\circ} \lightning\, c_1 \parallel \cdots \parallel \lightning\, c_n$$

By T-Halt, we have that $\cdot; \cdot \vdash^{\bullet} \mathbf{halt}$. Thus, recomposing, we arrive at

$$\dfrac{\cdot; \cdot \vdash^{\bullet} \mathbf{halt} \qquad \dfrac{\dfrac{}{c_1 : S_1; \cdot \vdash^{\circ} \lightning\, c_1} \qquad \dfrac{\dfrac{}{c_{n-1} : S_{n-1}; \cdot \vdash^{\circ} \lightning\, c_{n-1}} \qquad \dfrac{}{c_n : S_n; \cdot \vdash^{\circ} \lightning\, c_n}}{\vdots}}{c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^{\circ} \lightning\, c_1 \parallel \cdots \parallel \lightning\, c_n}}{c_1 : S_1, \ldots, c_n : S_n; \cdot \vdash^{\bullet} \mathbf{halt} \parallel \lightning\, c_1 \parallel \cdots \parallel \lightning\, c_n}$$

as required.

Case E-LiftC

Assumptions:

- $\Gamma;\Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$
- $\mathcal{C} \longrightarrow \mathcal{D}$

Let $\mathbf{D}$ be a derivation of $\Gamma;\Delta \vdash^{\phi} \mathcal{G}[\mathcal{C}]$. By Lemma 59, we have that there exists some $\mathbf{D}'$ such that $\mathbf{D}'$ is a subderivation of $\mathbf{D}$ concluding $\Gamma';\Delta' \vdash^{\phi'} \mathcal{C}$, where the position of $\mathbf{D}'$ in $\mathbf{D}$ corresponds to that of the hole in $\mathcal{G}$.

By the IH, we have that there exists some $\Gamma'';\Delta''$ such that $\Gamma;\Delta \longrightarrow^{?} \Gamma'';\Delta''$ and $\Gamma'';\Delta'' \vdash^{\phi} \mathcal{D}$.

By Lemma 60, we have that there exist some $\Gamma''';\Delta'''$ such that $\Gamma;\Delta \longrightarrow^{?} \Gamma''';\Delta'''$ and $\Gamma''';\Delta''' \vdash^{\phi} \mathcal{G}[\mathcal{D}]$, as required.

Case E-LiftM

Assumptions:

$$\frac{\Gamma \vdash M : A}{\Gamma;\cdot \vdash^{\bullet} \bullet M}$$

and $M \longrightarrow_{\mathsf{M}} N$. By Lemma 61, we have that $\Gamma \vdash N : A$. Recomposing:

$$\frac{\Gamma \vdash N : A}{\Gamma;\cdot \vdash^{\bullet} \bullet N}$$

as required.

□

D.2 Canonical Forms

Theorem 33: Canonical Forms *Given $\mathcal{C}$ such that $\Gamma;\Delta \vdash^{\bullet} \mathcal{C}$, there exists some $\mathcal{C}' \equiv \mathcal{C}$ such that $\Gamma;\Delta \vdash^{\bullet} \mathcal{C}'$ and $\mathcal{C}'$ is in canonical form.*

Proof. The proof is by induction on the count of ν-bound variables, following Lindley and Morris [132]. Without loss of generality, assume that the ν-bound variables of $\mathcal{C}$ are distinct. Let $\{a_i \mid 1 \leq i \leq n\}$ be the set of ν-bound variables in $\mathcal{C}$ and let $\{\mathcal{D}_j \mid 1 \leq j \leq m\}$ be the set of threads in $\mathcal{C}$.

In the case that $n = 0$, by Lemma 67 we can safely commute the main thread such that it is the rightmost configuration, and associate parallel composition to the right using Lemma 68 to derive a well-typed canonical form.

In the case that $n \geq 1$, pick some a_i and $\mathcal{D}_j$ such that a_i is the only ν-bound variable in $\mathsf{fn}(\mathcal{D}_j)$; Lemma 63 and a standard counting argument ensure that such a name and configuration exist. By the equivalence rules, there exists $\mathcal{E}$ such that $\Gamma;\Delta \vdash^{\phi} \mathcal{C} \equiv (\nu a_i)(\mathcal{D}_j \parallel \mathcal{E})$ (that a_i is the only ν-bound variable in $\mathsf{fn}(\mathcal{D}_j)$ ensures well-typing). Moreover, we have that there exist $\Gamma' \subseteq \Gamma$, $\Delta' \subseteq \Delta$, and S, such that either $\Gamma', a_i : S; \Delta' \vdash^{\phi} \mathcal{E}$ or $\Gamma'; \Delta', a_i : S \vdash^{\phi} \mathcal{E}$. By the induction hypothesis, there exists $\mathcal{E}'$ in canonical form such that either $\Gamma', a_i : S; \Delta' \vdash^{\phi} \mathcal{E} \equiv \mathcal{E}'$ or $\Gamma'; \Delta', a_i : S \vdash^{\phi} \mathcal{E} \equiv \mathcal{E}'$. Let $\mathcal{C}' = (\nu a_i)(\mathcal{D}_j \parallel \mathcal{E}')$. By construction it holds that $\Gamma;\Delta \vdash^{\phi} \mathcal{C} \equiv \mathcal{C}'$ and that $\mathcal{C}'$ is in canonical form. □

D.3 Progress

To prove Theorem 10, we prove a similar property in which canonical configurations are decomposed step-by-step rather than in one go.

Definition 20 (Open Progress). *Suppose $\Psi;\Delta \vdash^{\bullet} \mathcal{C}$, where $\mathcal{C}$ is in canonical form and $\mathcal{C} \not\Longrightarrow$. We say that $\mathcal{C}$* satisfies open progress *if:*

1. *$\mathcal{C} = (\nu a)(\mathcal{A} \parallel \mathcal{D})$, where $\Psi = \Psi_1, \Psi_2$ and $\Delta = \Delta_1, \Delta_2$ such that either:*
   (a) *$\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{D}$ where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either:*
      i. *A thread $\circ M$ where $\mathit{ready}(b, M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$; or*
      ii. *A zapper thread $\lightning a$; or*
      iii. *A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \overrightarrow{V}$ or $a \in \overrightarrow{W}$*
   (b) *$\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{D}$, where $\mathcal{D}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$*
2. *$\mathcal{C} = \mathcal{A} \parallel \mathcal{M}$, where $\Psi = \Psi_1, \Psi_2$ and either:*
   (a) *$\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A}$ and $\Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either:*
      i. *A thread $\circ M$ where $\mathit{ready}(b, M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$; or*
      ii. *A zapper thread $\lightning a$; or*
      iii. *A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$ where $b, c \neq a$ and either $a \in \mathsf{fn}(\overrightarrow{V})$ or $a \in \mathsf{fn}(\overrightarrow{W})$*
   (b) *$\Delta = \Delta_1, \Delta_2, a : S^{\sharp}$, where $\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}$ and $\Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{M}$, where $\mathcal{M}$ satisfies open progress, and $\mathcal{A}$ is either $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ or $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$ for some $b \in \mathsf{fn}(\Delta_1)$*

(c) $\Delta = \Delta_1, \Delta_2$, *where* $\Psi_1; \Delta_1 \vdash^{\circ} \mathcal{A}$ *and* $\Psi_2; \Delta_2 \vdash^{\bullet} \mathcal{M}$, *where* $\mathcal{M}$ *satisfies open progress, and* $\mathcal{A}$ *is either:*

i. *A thread* $\circ M$ *where either* $M = ()$, *or* $\mathit{ready}(a, M)$ *for some* $a \in \mathsf{fn}(\Psi_1)$; *or*

ii. *A zapper thread* $\lightning a$ *for some* $a \in \mathsf{fn}(\Psi_1)$; *or*

iii. *A buffer* $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ *for some* $a, b \in \mathsf{fn}(\Delta_1)$

3. $\mathcal{C} = \mathcal{T}$, *where either:*

(a) $\mathcal{T} = \bullet N$, *where* N *is either a value or* $\mathit{ready}(b, N)$ *for some* $b \in \mathsf{fn}(\Psi)$

(b) $\mathcal{T} = \mathbf{halt}$

Lemma 64 *Suppose* $\Psi; \Delta \vdash^{\bullet} \mathcal{C}$, *where* $\mathcal{C}$ *is in canonical form and* $\mathcal{C} \not\Longrightarrow$. *Then* $\mathcal{C}$ *satisfies open progress.*

Proof. By induction on the derivation of $\Psi; \Delta \vdash^{\bullet} \mathcal{C}$. We have three cases, based on the structure of the given canonical form.

Case $\mathcal{C} = (\nu a)(\mathcal{A} \parallel \mathcal{D})$, with $a \in \mathsf{fn}(\mathcal{A})$, and where $\mathcal{D}$ is in canonical form

By assumption, we know that $\Psi; \Delta \vdash^{\phi} (\nu a)(\mathcal{A} \parallel \mathcal{D})$.

This configuration is typeable by T-Nu, followed by either T-Connect$_1$ or T-Connect$_2$. As the definition of canonical forms requires that $a \in \mathsf{fn}(\mathcal{A})$, it cannot be the case that the parallel composition arises as a result of T-Mix.

We consider these two subcases to show that $\mathcal{A}$ satisfies the properties required by open progress.

Subcase T-Connect$_1$

$$\dfrac{\dfrac{\Psi_1, a : S; \Delta_1 \vdash^{\phi_1} \mathcal{A} \quad \Psi_2; \Delta_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{A} \parallel \mathcal{D}}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2 \vdash^{\phi_1 + \phi_2} (\nu a)(\mathcal{A} \parallel \mathcal{D})}$$

By the definition of auxiliary threads and inversion on the typing relation, we know that $\mathcal{A}$ is of the following forms:

- $\circ M$, where $a \in \mathsf{fn}(M)$, and $\Psi_1, a : S \vdash M : \mathbf{1}$
- $\lightning a$
- $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \in \mathsf{fn}(\Delta_1)$ and $a \in \mathsf{fn}(V)$
- $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \in \mathsf{fn}(\Delta_1)$ and $a \in \mathsf{fn}(W)$

(since $a \notin \mathsf{fn}(\Delta_1)$, it cannot be the case that a appears as a buffer endpoint).

Lemma 62 tells us that either there exists some M' such that $M \longrightarrow_{\mathsf{M}} M'$; that M is a value; or there exist E, N such that $M = E[N]$ where N is either **raise** or a communication and concurrency construct. Since $\mathcal{C} \not\Longrightarrow$, we have that M is unable to reduce (as otherwise $\mathcal{C}$ could reduce by E-LiftM). Since $a \in \mathsf{fn}(M)$ and a does not have type $\mathbf{1}$, it cannot be the case that M is a value.

Therefore, we have that M has the form $E[N]$, where N is either **raise** or a communication / concurrency construct. This cannot be **fork**, since **fork** may always reduce by E-Fork, nor can it be **raise**, which could reduce by E-Raise, or E-RaiseChild depending on the enclosing evaluation context. Thus, there must exist some $b \in \mathsf{fn}(\Psi, a : S)$ such that $\mathsf{ready}(b, M)$.

Subcase T-Connect$_2$

$$\frac{\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A} \quad \Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{D}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{D}}$$
$$\Psi_1, \Psi_2; \Delta_1, \Delta_2 \vdash^{\bullet} (\nu a)(\mathcal{A} \parallel \mathcal{D})$$

By the definition of auxiliary threads and inversion on the typing relation, we know that $\mathcal{A}$ is of the following forms:

- $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$, where $b \in \mathsf{fn}(\Delta_1)$
- $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$, where $b \in \mathsf{fn}(\Delta_1)$

(as $a \in \mathsf{fn}(\mathcal{A})$ and $a \in \mathsf{fn}(\Delta_1)$, it cannot be the case that $\mathcal{A}$ is a child thread or a zapper thread, as these require empty runtime typing environments).

By the induction hypothesis, we know that $\mathcal{D}$ satisfies open progress; hence $(\nu a)(\mathcal{A} \parallel \mathcal{D})$ satisfies open progress.

Case $\mathcal{C} = \mathcal{A} \parallel \mathcal{M}$

There are three subcases, based on whether the parallel composition arises as a result of T-Connect$_1$, T-Connect$_2$, or T-Mix.

Subcase T-Connect$_1$

$$\frac{\Psi_1, a : S; \Delta_1 \vdash^{\circ} \mathcal{A} \quad \Psi_2; \Delta_2, a : \overline{S} \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

By inversion on the typing rules, we have that $\mathcal{A}$ may be:

- A child thread $\circ M$, where $a \in \mathsf{fn}(M)$
- A zapper thread $\lightning a$

- A buffer $b(\overrightarrow{V}) \leftrightsquigarrow c(\overrightarrow{W})$, where $b, c \neq a$ and either $a \in \mathsf{fn}(\overrightarrow{V})$ or $a \in \mathsf{fn}(\overrightarrow{W})$

In the case of (1), by Lemma 62, we have that either M is a value; there exists N such that $M \longrightarrow_{\mathsf{M}} N$; or $M = E[N]$ for some E, N, where N is a communication / concurrency construct.

By T-Child, $\Psi_1, a : S \vdash M : \mathbf{1}$. Since $a \in \mathsf{fn}(M)$ and the only value with type $\mathbf{1}$ is the unit value $()$ it therefore cannot be the case that M is a value. Since $\mathcal{C} \not\Longrightarrow$, it cannot be the case that $M \longrightarrow_{\mathsf{M}} N$, since otherwise $\mathcal{C}$ could reduce. Thus, it must be the case that $M = E[N]$ where N either **raise** or a communication and concurrency construct; by similar reasoning as above cases, it therefore must be the case that $\mathsf{ready}(b, M)$ for some $b \in \mathsf{fn}(\Psi_1, a : S)$.

(2) and (3) satisfy the required conditions by definition.

Subcase T-Connect$_2$

$$\frac{\Psi_1; \Delta_1, a : \overline{S} \vdash^{\circ} \mathcal{A}; \Psi_2, a : S; \Delta_2 \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2, a : S^{\sharp} \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

Since the runtime typing environment $\Delta_1, a : \overline{S}$ is non-empty, it cannot be the case that $\mathcal{A}$ is a child thread or zapper thread. Thus, $\mathcal{A}$ must either be of the form:

1. $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$, where $a, b \in \mathsf{fn}(\Delta_1)$; or
2. $b(\overrightarrow{V}) \leftrightsquigarrow a(\overrightarrow{W})$, where $a, b \in \mathsf{fn}(\Delta_1)$

which satisfy the required conditions by definition.

Subcase T-Mix

$$\frac{\Psi_1; \Delta_1 \vdash^{\circ} \mathcal{A} \quad \Psi_2; \Delta_2 \vdash^{\bullet} \mathcal{M}}{\Psi_1, \Psi_2; \Delta_1, \Delta_2 \vdash^{\bullet} \mathcal{A} \parallel \mathcal{M}}$$

By inversion on the typing rules, we have that $\mathcal{A}$ may either be:

1. A child thread $\circ M$
2. A zapper thread $\lightning a$ for some $a \in \mathsf{fn}(\Psi_1)$
3. A buffer thread $a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W})$ for some $a, b \in \mathsf{fn}(\Delta_1)$

By Lemma 62, we have that M is either a value V; there exists some N such that $M \longrightarrow_{\mathsf{M}} N$; or $M = E[N]$ for some E, N such that N is either **raise** or a communication and concurrency primitive. It cannot be the case that $M \longrightarrow_{\mathsf{M}} N$ since otherwise the configuration could reduce.

By T-Child, it must be the case that $\Psi_1; \Delta_1 \vdash M : \mathbf{1}$; if M is a value then by inversion on the term typing rules, it must be the case that $M = ()$.

Following the same reasoning as previous cases, if $M = E[N]$ then it must be that $\mathsf{ready}(a, M)$ for some $a \in \mathsf{fn}(\Psi_1)$.

By the induction hypothesis, we know that $\mathcal{M}$ satisfies open progress; hence $\mathcal{A} \parallel \mathcal{M}$ satisfies open progress.

Case $\mathcal{C} = \mathcal{T}$

Assumption: $\Psi; \Delta \vdash^{\bullet} \mathcal{T}$. By the definition of $\mathcal{T}$, we have two subcases:

Subcase $\mathcal{T} = \bullet M$

$$\frac{\Psi \vdash M : A}{\Psi; \cdot \vdash^{\bullet} \bullet M}$$

By Lemma 62, we have that either M is a value; that there exists some N such that $M \longrightarrow_{\mathsf{M}} N$; or that there exist some E, N such that $M = E[N]$ where N is a communication / concurrency primitive.

Again, as $\mathcal{C} \not\Longrightarrow$, it cannot be the case that $M \longrightarrow_{\mathsf{M}} N$, since otherwise $\mathcal{C}$ could reduce. If M is a value, then $\mathcal{T}$ satisfies open progress.

Finally, if $M = E[N]$ where N is a either **raise** or communication / concurrency primitive, it cannot be the case that $N =$ **raise** since it could reduce either by E-Raise or E-RaiseMain, and it cannot be the case that $N =$ **fork** M' since it could reduce by T-Fork. Therefore it must be the case that $\mathsf{ready}(a, M)$ for some $a \in \mathsf{fn}(\Psi)$, satisfying open progress, as required.

Subcase $\mathcal{T} =$ **halt**

Immediate by the definition of open progress.

□

Theorem 10 provides a more global and concise view of the properties exhibited by a non-reducing process in canonical form, and arises as an immediate corollary.

Theorem 10 *Suppose* $\Psi; \Delta \vdash^{\bullet} \mathcal{C}$ *where* $\mathcal{C}$ *is in canonical form and* $\mathcal{C} \not\Longrightarrow$.

Let $\mathcal{C} = (\nu a_1)(\mathcal{A}_1 \parallel (\nu a_2)(\mathcal{A}_2 \parallel \cdots \parallel (\nu a_n)(\mathcal{A}_n \parallel \mathcal{M}))\ldots))$.

1. *For* $1 \leq i \leq n$, *each thread in* $\mathcal{A}_i$ *is either:*
   (a) *a child thread* $\circ M$ *for which there exists* $a \in \{a_j \mid 1 \leq j \leq i\} \cup \mathsf{fn}(\Psi)$ *such that* $\mathsf{ready}(a, M)$;
   (b) *a zapper thread* $\lightning a_i$; *or*
   (c) *a buffer.*
2. $\mathcal{M} = \mathcal{A}'_1 \parallel \cdots \parallel \mathcal{A}'_m \parallel \mathcal{T}$ *such that for* $1 \leq j \leq m$:
   (a) $\mathcal{A}'_j$ *is either:*

i. a child thread $\circ N$ *such that* $N = ()$ *or ready*(a, N) *for some* $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$;

ii. a zapper thread $\lightning a$ *for some* $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$; *or*

iii. a buffer.

(b) Either $\mathcal{T} = \bullet N$, *where* N *is either a value or ready*(a, N) *for some* $a \in \{a_i \mid 1 \leq i \leq n\} \cup \mathsf{fn}(\Psi) \cup \mathsf{fn}(\Delta)$; *or* $\mathcal{T} = \textbf{\textit{halt}}$.

D.4 Confluence

Theorem 35 (Diamond Property) *If* $\Psi; \Delta \vdash^{\phi} \mathcal{C}$, *and* $\mathcal{C} \Longrightarrow \mathcal{D}_1$, *and* $\mathcal{C} \Longrightarrow \mathcal{D}_2$, *then either* $\mathcal{D}_1 \equiv \mathcal{D}_2$, *or there exists some* $\mathcal{D}_3$ *such that* $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$ *and* $\mathcal{D}_2 \Longrightarrow \mathcal{D}_3$.

Proof. As noted in Section 9.3.4, $\longrightarrow_{\mathsf{M}}$ is deterministic and hence confluent due to the setup of term evaluation contexts, and linearity ensures that endpoints to a buffer may not be shared. Consequently, communication actions on different channels may be performed in any order.

Nevertheless, two critical pairs arise due to asynchrony. The first arises when it is possible to send to or receive from a buffer; there is a choice of whether the send or the receive happens first. Both cases reduce to the same configuration after a single further step.

$$\begin{array}{ccc} & \mathcal{F}[\mathbf{send}\, U\, a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) \parallel \mathcal{F}'[\mathbf{receive}\, b] & \\ \swarrow & & \searrow \\ \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) \parallel \mathcal{F}'[\mathbf{receive}\, b] & & \mathcal{F}[\mathbf{send}\, U\, a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \parallel \mathcal{F}'[(V, b)] \\ \searrow & & \swarrow \\ & \mathcal{F}[a] \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) \parallel \mathcal{F}'[(V, b)] & \end{array}$$

The second critical pair arises when sending to a buffer where the peer endpoint has a non-empty buffer and has been cancelled. There is a choice as to whether the value at the head of the queue is cancelled before or after the send takes place. Again, both cases reduce to the same configuration after a single further step.

$$\begin{array}{ccc} & \mathcal{F}[\mathbf{send}\, U\, a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W}) & \\ \swarrow & & \searrow \\ \mathcal{F}[a] \parallel \lightning b \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(V \cdot \overrightarrow{W} \cdot U) & & \mathcal{F}[\mathbf{send}\, U\, a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W}) \\ \searrow & & \swarrow \\ & \mathcal{F}[a] \parallel \lightning b \parallel \lightning V \parallel a(\overrightarrow{V}) \leftrightsquigarrow b(\overrightarrow{W} \cdot U) & \end{array}$$

□

Appendix E

Distributed Delegation

We saw an example of session delegation in §10.4, in the `ChatClient` type:

```
typename ChatClient =!Nickname.
  [&|Join:?(Topic, [Nickname], ClientReceive).ClientSend,
     Nope:End|&];
```

In this example, an endpoint of type `ClientReceive` is passed as a message.

Challenges of Distributed Delegation. Session delegation is a vital abstraction in session-based programming. However, its integration with both asynchrony *and* distribution brings several challenges. The seminal work on distributed delegation is Session Java [106].

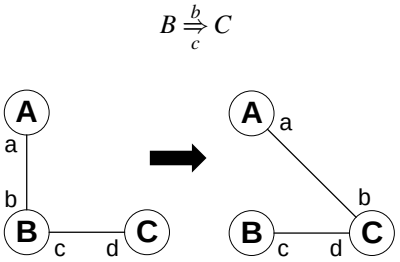
Fig. E.1 shows three scenarios of distributed delegation. We write $X \overset{x}{\underset{y}{\Rightarrow}} Y$ to indicate that X wishes to send x to Y over y on the basis that X's last known location of the corresponding endpoint for y is Y.

Consider Figure E.1a, where $B \overset{b}{\underset{c}{\Rightarrow}} C$. Following Hu et al. [106], we refer to B as the *session-sender*, C as the *session-receiver*, and A as the *passive party*.

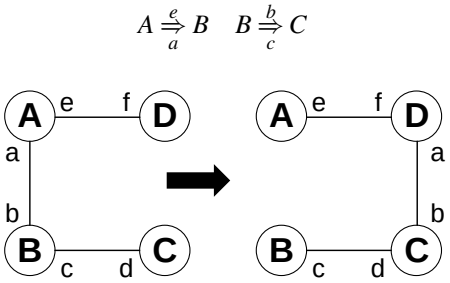
There is no happens-before relation between A sending a message to B along a, and B delegating b to C along c. Thus, a message could be sent to A *after* A has given up control of a. Following Hu et al. [106], we call such messages *lost messages*.

Approaches to Distributed Delegation. The simplest safe way to implement distributed delegation is to store all buffers on the server, but this requires a blocking remote call for every receive operation. A second naïve method is *indefinite redirection*, where the session-sender indefinitely forwards all messages to the session-receiver. This retains buffer locality, but requires the session-sender to remain online for the duration of the delegated session.

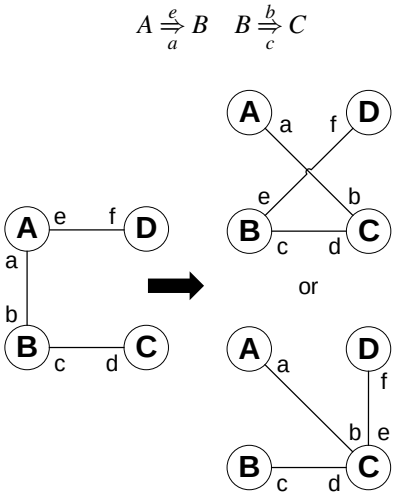
Hu et al. [106] describe two more realistic distributed delegation algorithms: a *resending* protocol, which re-sends lost messages *after* a connection for the delegated session is established, and a *forwarding* protocol, which forwards lost messages *before* the delegated session is established. The key idea behind both algorithms is to establish a connection between the



(a) Simple Delegation



(b) Simultaneous Delegation



(c) Entangled Delegation

Figure E.1: Cases of Distributed Delegation

passive party and the session-receiver, to ensure that the lost messages are received by the session-receiver, and to continue the session only once lost messages are received.

Alas, we cannot directly re-use the resending and forwarding protocols of Hu et al. [106] because of two fundamental differences in our setting: Links clients do not connect to each other directly, and in Links multiple sessions may be sent at once. Thus, we describe the high-level details of a modified algorithm which addresses these two constraints. We utilise two key ideas:

- Much like the resending protocol, lost messages are retrieved and relayed to the session-receiver once the new session has been established.
- We ensure the session-receiver endpoint is not delegated until the delegation has completed, by queueing messages that include the session-receiver endpoint, and sending them once delegation has completed.

We now consider the case where session-sender and session-receiver are different clients; the case where session-sender is a client and session-receiver the server is similar.

Let client A be the session-sender and client B be the session-receiver.

Example. Suppose client A sends a value v containing a session endpoint b along channel (s,t), recalling that s is the peer endpoint and t is the local endpoint. The initial endpoint

1. $A \to S$: Send$(t, v, [b \mapsto \overrightarrow{V}])$
2. A : *start recording lost messages* $\overrightarrow{W}$ *for* b
3. $S : \sigma = \sigma[b \mapsto B]; \delta = \delta \cup \{t\}$
4. $S \to B$: Deliver$(t, v, [b \mapsto \overrightarrow{V}])$
5. $S \to A$: GetLostMessages$([b])$
6. A : *stop recording lost messages for* b
7. $A \to S$: LostMessageResponse$([b \mapsto \overrightarrow{W}])$
8. $S \to B$: Commit$(t, [b \mapsto \overrightarrow{W}])$
9. $S : \delta = \delta \backslash \{t\}$
10. B : *buffers*$[b] = \overrightarrow{V} +\!\!+ \overrightarrow{W} +\!\!+ \overrightarrow{U}$
    where $\overrightarrow{U}$ = *messages received for* b *between (3) and (8)*

Figure E.2: Operation of Distributed Delegation Protocol

location table is:

$$\sigma \triangleq [s \mapsto A, t \mapsto B, b \mapsto A, c \mapsto A]$$

Fig. E.2 shows the operation of the delegation protocol on this example. In Step 1, A sends a message to the server S, containing the peer endpoint t, value to send v, and the buffer $\overrightarrow{V}$ for b, before beginning to record lost messages for b. Upon receiving this message, the server updates its internal mapping for the location of b to be B, adds t to the set of delegation carriers δ, and sends a Deliver message containing t, v, and $\overrightarrow{V}$, before sending a GetLostMessages request to A. Upon receiving this message, A will stop recording lost messages for b, and relay the lost messages $\overrightarrow{W}$ for b to S. The server then sends a Commit message containing t and the lost messages for all delegated endpoints, and removes t from the set of delegation carriers.

Lost cancellation notifications for the peer of b are retrieved and relayed in the same way as lost messages for b.

The final buffer for b is the concatenation of the initial buffer $\overrightarrow{V}$, the lost messages $\overrightarrow{W}$, and all messages $\overrightarrow{U}$ received for b before the Commit message.

Correctness. We argue correctness of the algorithm in a similar manner to Hu et al. [106]. Due to co-operative threading, we can treat each sequence of actions happening at a single participant (for example, steps 3–8) as atomic. Since (as per step 3) the endpoint location table is updated prior to the lost message request, we can safely split the buffer of the delegated session into three parts: the initial buffer being delegated ($\overrightarrow{V}$); the lost messages ($\overrightarrow{W}$); and the messages received after the change in the lookup table but before the Commit message is received ($\overrightarrow{U}$) and reassemble them, retaining ordering.

In our setting, since session channels are not associated with sockets, simultaneous delegation (Fig. E.1b) can be handled in the same way as simple delegation. In the case of entangled

delegation (Fig. E.1c), since delegation carriers may not be delegated themselves until the lost messages have been received, we can be sure that the lost message requests are sent to the correct participant. Hence, the case devolves to simple delegation.