Session-Typed Concurrent Contracts

Hannah Gommerstadt

CMU-CS-19-119
September 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning (co-chair)
Limin Jia (co-chair)
Jan Hoffmann
Bernardo Toninho (Universidade Nova de Lisboa)
Adrian Francalanza (University of Malta)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

This research was sponsored by the National Science Foundation under grant numbers CNS1423168, CCF1718267, and CNS1704542; by the Office of Naval Research under grant number N000141712892; by a scholarship from the Microsoft Research Graduate Womens Scholarship Program; by a scholarship from the ARCS Foundation Achievement Rewards for College Scientists; and, by a fellowship from the Carnegie Mellon University Presidential Scholars Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: contracts, session types, monitors, blame assignment

This thesis is dedicated to my father, Boris Yakov Gommerstadt.

Abstract

Multi-process systems control the behavior of everything from datacenters storing our information to banking systems managing money. Each one of these processes has a prescribed role, their contract, that governs their behavior during the joint computation. When a single process violates their communication contract, the impact of this misbehavior can rapidly propagate through the system. This thesis develops techniques for dynamically monitoring expressive classes of concurrent contracts. We provide multiple mechanisms to monitor contracts of increasing complexity. In order to model message-passing concurrent computation, we use a session type system. First, we present a method for dynamic monitoring and blame assignment where communication contracts are expressed using session types. Second, we describe contract-checking processes that handle stateful contracts that cannot be expressed with a session type. These contract-checking processes are also able to encode type refinements. Third, we encode dependent types in our system which allow us to monitor complex invariants. Finally, we survey a number of other monitoring extensions including a mechanism to monitor deadlock.

Acknowledgments

This thesis could not have existed without my advisors Frank Pfenning and Limin Jia. Frank has supported me in all aspects of this work, most importantly by promptly answering my panicked late night emails. On numerous occasions, Frank has saved my proofs, and debugged my code. Limin Jia taught me how to approach the gnarliest theoretical problem and muddled through many questionable proofs with me. I am thankful to my thesis committee, Jan Hoffmann, Bernardo Toninho and Adrian Francalanza for their feedback. Deb Cavlovich provided logistical and emotional support throughout the entire Ph.D. process. On the teaching front, I am indebted to Iliano Cervesato for training me to run large courses smoothly, and to Tom Cortina for teaching me how to be tough and fair, but still approachable. I would not have started a Ph.D. if not for the mentorship of Greg Morrisett, Steve Chong and Aslan Askarov while I was an undergraduate.

The programming languages group has listened to countless talks, helped me pass category theory and built my knowledge of this field. Michael Coblenz's feedback has convinced me to think more rigorously about usability. Stephanie Balzer has constantly supported me, usually while running together in Shenley Park. Deby Katz set the standard for Ph.D. friendships by making me a pecan pie the night before my defense. Ryan Kavanagh sensed when we were both stuck on our proofs and accompanied me on many cider finding missions. The inaugural Awefficemates John Dickerson and Vagelis Papalexakis handled all of my early grad student panic with grace and welcomed me to campus instantly. The later Awefficemates, Nathan Fulton, Ellis Hershkowitz, Roie Levin (and office dog Maddie) took me on my first backpacking trip and helped me when I jammed every printer in Gates.

The Pittsburgh Jewish community (including the Hoexeters, Silvermans, Bruks, and Elvgrens) have fed me countless Friday nights and allowed me to embrace the calm of Shabbat. Miranda Chang drastically increased the amount of adventure in my life by cycling from Pittsburgh to Washington, DC with me and recruiting me for her rowing team. Laure Thompson was the best roommate for programming languages conferences (that is, before she defected to studying natural languages). Andrew Ruef wins the award for most visits to Pittsburgh and always manages to entertain me. Renee Stern has greeted me with a risotto every time I stumbled into her Manhattan apartment after a long day of travel or interviewing. Eva Belmont and I have been emailing each other novellas since our undergraduate days and I am glad the tradition has continued. Sam Milner has made me frequently affirm my decision to get a Ph.D. in computer science and not history by (repeatedly) mocking the length of this thesis.

My sister has been a role model for her endless reserves of patience – she reminds me that even when I have to explain the same algorithm a dozen times, at least I do not have to teach kindergarten. My mom continuously supports any adventure I undertake (even when she finds it questionable) and puts up with me throughout the process.

Contents

List of Figures

Chapter 1

Introduction

Multi-process systems control the behavior of everything from datacenters storing our information to banking systems managing money. Each one of these processes has a prescribed role, their *contract*, that governs their behavior during the joint computation. For example, in a cryptographic protocol, one process may be responsible for factoring numbers, a second for encrypting a message and a third for decrypting communication. The desired contract for the factoring process confirms that an integer is correctly factored into two integer factors. Some contracts are simple to monitor, for example, the fact that a process takes an integer as input and produces an integer as output. In other cases, contacts can be significantly more difficult to validate. For instance, checking that a sorted list has the same elements as the original list can be just as complex as sorting the original list. Ensuring that a process correctly factors a given integer lies somewhere in the middle of the contract complexity spectrum. The goal of this thesis is to extend the monitoring frontier to be able to handle a larger segment of the contract spectrum.

When a single process violates their communication contract, the impact of this misbehavior can rapidly propagate through the system. Moreover, when the problem is eventually discovered, determining which process is to blame for the contract breach can be a challenge. A rogue factoring process could go unnoticed until a decryption went awry, leading to confusion about whether the factoring, the decryption, or both were at fault. In order to maintain the integrity of the system, it is necessary to not only abort the computation once incorrect behavior has occured, but also to hold the correct party accountable. This thesis develops techniques to detect and contain process misbehavior by dynamically monitoring violations of a process' contract.

In functional languages, a contract for a function can be modeled as an expressive type that places constraints on its arguments and return value [14]. Typically, as the function executes, these constraints are checked dynamically. If a constraint is violated, the system will attempt to assign blame to the party responsible for the contract violation. For example, if the argument of the function does not meet the constraints encoded in the function type, the fault likely lies with the caller of the function. Otherwise, if the return value is inconsistent with the function type, the function itself is to blame. While significant research has been done on dynamic contract checking in a sequential setting, concurrent computation presents an additional challenge.

In order to model concurrent computation, we use a session type system that was designed by Toninho et al. [37]. Session types are based on a computational interpretation of linear logic, which is a substructural logic that allows reasoning about resources within the logic itself. In

this logic, all resources are ephemeral and are consumed with each step of the computation. That is, we annotate each communication channel with a session type and as processes communicate over the channel by message passing, the type of the channel changes with every step of the communication. This temporal structure allows us to model concurrent communication protocols by imposing an ordering on the messages that flow through a given channel. Our type system supports higher-order communication, where channels can delegate communication to other channels. This property significantly complicates our monitoring infrastructure.

We can think of each channel's session type as prescribing the communication contract on that channel – for example, a session type could require that first an integer, z, be sent over a channel, and then a response consisting of an integer x and an integer y be received. As each integer is sent and received, the type of the channel is updated to reflect the current communication contract. However, a malicious process could replace the process that is supposed to send the integer z with a process that sends some nefarious string over this channel instead. In this situation, a monitor must detect that the communication contract has been breached and trigger an alarm.

Thesis Statement: Session typed monitors give rise to novel techniques for dynamically monitoring expressive classes of concurrent contracts and provide strong theoretical guarantees (safety and transparency).

Our first contribution is to dynamically monitor that each channel's communication contract is being upheld. If a string is observed flowing through a channel that expects an integer, our monitors will detect that the message is inconsistent with the channel's type and raise an alarm. We have designed a mechanism where monitors are placed on the communication channels. A monitor observes messages that flow through its channel and ascertains whether the messages are consistent with the channel's contract. If the messages follow the protocol, the monitor lets them through with no observable change to the computation. Otherwise, the monitor raises an alarm and indicates which process or processes are to blame for the faulty messages. Our monitoring mechanism is also able to handle higher-order communication, that is, when a channel decides to delegate communication to another channel.

We now consider a more complicated contract that guarantees that the integer response z is the product of the integers x and y. Monitoring this contract requires designing a system where a monitor has access to the values of x and y in order to compute their product and compare it with the integer z. In this thesis, we consider two orthogonal approaches for handling this contract. One method involves designing an operational monitor that executes concurrently with the factoring code and has the capacity to store all the integers it observes. The other tactic relies on augmenting the session type system with dependency in order to express the relevant contract within the type system.

The second contribution of this thesis is to develop partial identity monitors to handle a variety of contracts. These monitors are processes that are able to maintain state and perform calculations. This allows them to check complicated properties such as whether a list of parenthesis is matched or a list is in sorted order. The monitoring processes are observably equivalent to the identity process, up to termination, thus ensuring transparency of our monitors. Our partial identity monitors are also able to encode refinement contracts by checking whether a refinement

from one type to another type causes a runtime error.

Our third contribution is to encode dependent types in our system which allows us to monitor complex invariants. Monitoring dependent contracts is challenging because usual encodings of dependent types involve sending proof objects through the system. These proof objects must be generated and transmitted through the system, which requires significant infrastructure [25]. We provide a more lightweight approach by exploiting the fact that many proof objects are not relevant to the rest of the computation [28]. Therefore, we can avoid sending the actual proof object, but instead can check the condition encoded by the proof object dynamically.

The rest of this thesis is organized as follows. Chapter 2 provides background on session types and contracts. Chapter 3 presents results on dynamic monitoring in an untrusted setting. The work presented in this chapter is a reformulation and extension of a prior publication. [21]. Chapter 4 introduces partial identity monitors and examines key examples. Chapter 5 and Chapter 6 explore the encoding of refinement and dependent types, respectively. The work described in Chapter 4 and Chapter 5 are an expansion of a prior publication. [16]. Chapter 7 reviews various monitoring extensions including a mechanism to monitor deadlock.

Chapter 2

Background

In this chapter we first provide background for using session types to reason about concurrent computation. We then provide examples of contract checking in a functional language and survey recent work on contracts.

2.1 Session Types

Session types prescribe the communication behavior of message-passing concurrent processes. We approach them here via their foundation in intuitionistic linear logic [6, 7, 34]. Building on the Curry-Howard correspondence, the key idea is that an intuitionistic linear sequent

$$A_1, \ldots, A_n \vdash C$$

is interpreted as the interface to a *process expression* P. We label each of the antecedents with a channel name a_i and the succedent with a channel name c. The a_i are the channels *used* and c is the channel *provided* by P. Linearity requires that the process P provide a service on exactly one channel c.

$$a_1 : A_1, \ldots, a_n : A_n \vdash P :: (c : C)$$

We abbreviate the antecedents by the context Δ. All the channels a_i and c must be distinct, and bound variables may be silently renamed to preserve this invariant in the rules. Furthermore, the antecedents are considered modulo exchange. Cut corresponds to parallel composition of two processes that communicate along a private channel x, where P is the *provider* along x and Q the *client*.

$$\frac{\Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Delta, \Delta' \vdash x{:}A \leftarrow P \,;\, Q :: (c : C)}\ \mathsf{cut}$$

Operationally, the process $x : A \leftarrow P \,;\, Q$ spawns P as a new process and continues as Q, where P and Q communicate along a fresh channel a, which is substituted for x. We sometimes omit the type A of x in the syntax when it is not relevant or can be inferred.

In order to define the operational semantics rigorously, we use *multiset rewriting* [8]. The configuration of executing processes is described as a collection $\mathcal{C}$ of propositions $\mathsf{proc}(c, P)$

(process P is executing, providing along c) and $\mathsf{msg}(c, M)$ (message M is sent along c). All the channels c provided by processes and messages in a configuration must be distinct.

To begin with, a cut just spawns a new process, and is in fact the only way new processes are spawned. We describe a transition $\mathcal{C} \longrightarrow \mathcal{C}'$ by defining how a subset of $\mathcal{C}$ can be rewritten to a subset of $\mathcal{C}'$, possibly with a freshness condition that applies to all of $\mathcal{C}$ in order to guarantee the uniqueness of each channel provided.

$$\mathsf{proc}(c, x{:}A \leftarrow P\ ;\ Q) \longrightarrow \mathsf{proc}(a, [a/x]P), \mathsf{proc}(c, [a/x]Q) \quad \textit{(a fresh)}$$

Each of the connectives of linear logic then describes a particular kind of communication behavior which we capture in similar rules. Before we move on to that, we consider the identity rule, in logical form and operationally.

$$\frac{}{A \vdash A}\ \mathsf{id} \qquad \frac{}{b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id} \qquad \mathsf{proc}(a, a \leftarrow b), \mathcal{C} \longrightarrow [b/a]\mathcal{C}$$

Operationally, it corresponds to identifying the channels a and b, which we implement by substituting b for a in the remainder $\mathcal{C}$ of the configuration (which we make explicit in this rule because it is the only rule that substitutes into the entire configuration). The process offering a terminates. We refer to $a \leftarrow b$ as *forwarding* since any messages along a are instead "forwarded" to b.

We consider each class of session type constructors, describing their process expression, typing, and asynchronous operational semantics. The linear logical semantics can be recovered by ignoring the process expressions and channels.

Internal and external choice Even though we distinguish a *provider* and its *client*, this distinction is orthogonal to the direction of communication: both may either send or receive along a common private channel. Session typing guarantees that both sides will always agree on the direction and kind of message that is sent or received, so our situation corresponds to so-called *binary session types* [19].

First, the *internal choice* $c : A \oplus B$ requires the provider to send a token inl or inr along c and continue as prescribed by type A or B, respectively. For convenience, we support n-ary labelled choice $\oplus\{\ell : A_\ell\}_{\ell \in L}$ where L is a set of labels. A process providing $c : \oplus\{\ell : A_\ell\}_{\ell \in L}$ sends a label $k \in L$ along c and continues with type A_k. The client will operate dually, branching on a label received along c.

$$\frac{k \in L \quad \Delta \vdash P :: (c : A_k)}{\Delta \vdash c.k\ ;\ P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

The operational semantics is somewhat tricky, because we communicate asynchronously. We need to spawn a message carrying the label k, but we also need to make sure that the *next* message send along the same channel does not overtake the first (which would violate session fidelity). Sending a message therefore creates a fresh continuation channel c' for further communication,

which we substitute in the continuation of the process. Moreover, the recipient also switches to this continuation channel after the message is received.

$$\mathsf{proc}(c, c.k\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, c.k\ ;\ c \leftarrow c') \quad \textit{(}c'\textit{ fresh)}$$
$$\mathsf{msg}(c, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \longrightarrow \mathsf{proc}(d, [c'/c]Q_k)$$

It is interesting that the message along c, followed by its continuation c' can be expressed as a well-typed process expression using forwarding $c.k\ ;\ c \leftarrow c'$. This pattern will work for all other pairs of send/receive operations.

External choice reverses the roles of client and provider, both in the typing and the operational rules. The typing and semantic rules are shown below.

$$\frac{\Delta \vdash P_\ell :: (c : A_\ell) \quad \text{for every } \ell \in L}{\Delta \vdash \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (c : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L \quad \Delta, c : A_k \vdash Q :: (d : D)}{\Delta, c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\ ;\ Q :: (d : D)}\ \&L$$

$$\mathsf{proc}(d, c.k\ ;\ Q) \longrightarrow \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh})$$
$$\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P_k)$$

Sending and receiving channels Session types are *higher-order* in the sense that we can send and receive channels along channels. Sending a channel is perhaps less intuitive from the logical point of view, so we show that and just summarize the rules for receiving.

If we provide $c : A \otimes B$, we send a channel $a : A$ along c and continue as B. From the typing perspective, it is a restricted form of the usual two-premise $\otimes R$ rule by requiring the first premise to be an identity. This restriction separates spawning of new processes from the sending of channels.

$$\frac{\Delta \vdash P :: (c : B)}{\Delta, a : A \vdash \mathsf{send}\ c\ a\ ;\ P :: (c : A \otimes B)}\ \otimes R^*$$

$$\frac{\Delta, x : A, c : B \vdash Q :: (d : D)}{\Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \otimes L$$

The operational rules follow the same patterns as the previous case.

$$\mathsf{proc}(c, \mathsf{send}\ c\ a\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c') \quad \textit{(}c'\textit{ fresh)}$$
$$\mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c][a/x]Q)$$

Receiving a channel (written as a linear implication $A \multimap B$) works symmetrically. The typing and semantic rules are given below.

$$\frac{\Delta, x : A \vdash P :: (c : B)}{\Delta \vdash x \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : A \multimap B)}\ \multimap R$$

$$\frac{\Delta, c : B \vdash Q :: (d : D)}{\Delta, a : A, c : A \multimap B \vdash \mathsf{send}\ c\ a\ ;\ Q :: (d : D)}\ \multimap L$$

$$\mathsf{proc}(d, \mathsf{send}\ c\ a\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh})$$
$$\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][a/x]P)$$

Termination We have already seen that a process can terminate by forwarding. Communication along a channel ends explicitly when it has type $\mathbf{1}$ (the unit of $\otimes$) and is closed. By linearity there must be no antecedents in the right rule.

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Delta \vdash Q :: (d : D)}{\Delta, c : \mathbf{1} \vdash \mathsf{wait}\ c\ ;\ Q :: (d : D)}\ \mathbf{1}L$$

Since there cannot be any continuation, the message takes a simple form.

$$\mathsf{proc}(c, \mathsf{close}\ c) \longrightarrow \mathsf{msg}(c, \mathsf{close}\ c)$$
$$\mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, Q)$$

Quantification First-order quantification over elements of domains such as integers, strings, or booleans allows ordinary basic data values to be sent and received. At the moment, since we have no type families indexed by values, the quantified variables cannot actually appear in their scope. This will change in Section 5 so we anticipate this in these rules.

In order to track variables ranging over values, a new context Ψ is added to all judgments and the preceding rules are modified accordingly. All value variables n declared in Ψ must be distinct. Such variables are not linear, but can be arbitrarily reused, and are therefore propagated to all premises in all rules. We write $\Psi \vdash v : \tau$ to check that value v has type τ in context Ψ.

$$\frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R \qquad \frac{\Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{:}\tau.\ A \vdash n \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L$$

$$\mathsf{proc}(c, \mathsf{send}\ c\ v\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ v\ ;\ c \leftarrow c') \quad (c' \text{ fresh})$$
$$\mathsf{msg}(c, \mathsf{send}\ c\ v\ ;\ c \leftarrow c'), \mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c][v/n]Q)$$

The situation for universal quantification is symmetric. The typing and semantic rules are provided below.

$$\frac{\Psi, n{:}\tau\ ;\ \Delta \vdash P :: (c : A)}{\Psi\ ;\ \Delta \vdash n \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \forall n{:}\tau.\ A)}\ \forall R \qquad \frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \forall n{:}\tau.\ A \vdash \mathsf{send}\ c\ v\ ;\ Q :: (d : D)}\ \forall L$$

$$\mathsf{proc}(d, \mathsf{send}\ c\ v\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ v\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh})$$
$$\mathsf{proc}(c, n \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ v\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][v/n]P)$$

Processes may also make internal transitions while computing ordinary values, which we don't fully specify here. Such a transition would have the form

$$\mathsf{proc}(c, P[e]) \longrightarrow \mathsf{proc}(c, P[e']) \quad \text{if} \quad e \mapsto e'$$

where $P[e]$ would denote a process with an ordinary value expression in evaluation position and $e \mapsto e'$ would represent a step of computation.

Shifts Finally, we come to shifts. The purpose of shifts is to track the direction of communication, which simplifies monitoring. To make this explicit, we *polarize* the syntax and use so-called *shifts* to change the direction of communication. For more detail, see Pfenning and Griffith [29].

$$\begin{array}{lllll} \text{Negative types} & A^-, B^- & ::= & \&\{\ell : A^-_\ell\}_{\ell \in L} \mid A^+ \multimap B^- \mid \forall n{:}\tau.\, A^- \mid \uparrow A^+ \\ \text{Positive types} & A^+, B^+ & ::= & \oplus\{\ell : A^+_\ell\}_{\ell \in L} \mid A^+ \otimes B^+ \mid 1 \mid \exists n{:}\tau.\, A^+ \mid \downarrow A^- \\ \text{Types} & A, B, C, D & ::= & A^- \mid A^+ \end{array}$$

From the perspective of the provider, all negative types receive and all positive types send. It is then clear that $\uparrow A$ must receive a shift message and then start sending, while $\downarrow A$ must send a shift message and then start receiving. For this restricted form of shift, the logical rules are otherwise uninformative. The semantics and the typing is given below.

$$\begin{array}{l} \mathsf{proc}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ P) \longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c') \quad (c' \text{ fresh}) \\ \mathsf{msg}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c]Q) \end{array}$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^-)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ P :: (c : \downarrow A^-)}\ \downarrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^- \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \downarrow A^- \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \downarrow L$$

$$\begin{array}{l} \mathsf{proc}(d, \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q) \quad (c' \text{ fresh}) \\ \mathsf{proc}(c, \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P) \end{array}$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^+)}{\Psi\ ;\ \Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \uparrow A^+)}\ \uparrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^+ \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \uparrow A^+ \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q :: (d : D)}\ \uparrow L$$

Recursive types Practical programming with session types requires them to be recursive, and processes using them also must allow recursion. For example, lists with elements of type int can be defined as the purely positive type list^+.

$$\mathsf{list}^+ = \oplus\{\mathsf{cons} : \exists n{:}\mathsf{int}.\mathsf{list}^+\ ;\ \mathsf{nil} : \mathbf{1}\}$$

A provider of type $c : \mathsf{list}$ is required to send a sequence such as $\mathsf{cons} \cdot v_1 \cdot \mathsf{cons} \cdot v_2 \cdots$ where each v_i is an integer. If it is finite, it must be terminated with $\mathsf{nil} \cdot \mathsf{end}$ where the end message is shorthand for $\mathsf{msg}(c, \mathsf{close}\ c)$. In the form of a grammer, we could write

$$\mathit{From} ::= \mathsf{cons} \cdot v \cdot \mathit{From} \mid \mathsf{nil} \cdot \mathsf{end}$$

A second example is a multiset (bag) of integers, where the interface allows inserting and removing elements, and testing if it is empty. If the bag is empty when tested, the provider terminates after responding with the empty label.

$$\begin{aligned}\mathsf{bag} = \&\{&\mathsf{insert} : \forall n{:}\mathsf{int}.\mathsf{bag}^- \ ; \\ &\mathsf{remove} : \forall n{:}\mathsf{int}.\mathsf{bag}^- \ ; \\ &\mathsf{is\_empty} : \uparrow \oplus\{\mathsf{empty} : \mathbf{1} \ ; \ \mathsf{nonempty} : \downarrow \mathsf{bag}^-\}\}\end{aligned}$$

The protocol now describes the following grammar of exchanged messages, where *To* goes to the provider, *From* comes from the provider, and v stands for integers.

$$\begin{array}{lcl} \mathit{To} & ::= & \mathsf{insert} \cdot v \cdot \mathit{To} \mid \mathsf{remove} \cdot v \cdot \mathit{To} \mid \mathsf{is\_empty} \cdot \mathsf{shift} \cdot \mathit{From} \\ \mathit{From} & ::= & \mathsf{empty} \cdot \mathsf{end} \mid \mathsf{nonempty} \cdot \mathsf{shift} \cdot \mathit{To} \end{array}$$

For these protocols to be realized in this form and support rich subtyping and refinement types without change of protocol, it is convenient for recursive types to be *equirecursive*. This means a defined type such as list^+ is viewed as *equal* to its definition $\oplus\{\ldots\}$ rather than *isomorphic*. For this view to be consistent, we require type definitions to be *contractive* [15], that is, they need to provide at least one send or receive interaction before recursing.

The most popular formalization of equirecursive types is to introduce an explicit μ-constructor. For example,

$$\mathsf{list} = \mu\alpha.\, \oplus\{\ \mathsf{cons} : \exists n{:}\mathsf{int}.\, \alpha, \mathsf{nil} : \mathbf{1}\ \}$$

with rules unrolling the type $\mu\alpha.\, A$ to $[(\mu\alpha.\, A)/\alpha]A$. An alternative (see, for example, Balzer and Pfenning [3]) is to use an explicit definition just as we stated, for example, list and bag, and consider the left-hand side *equal* to the right-hand side in our discourse. In typing, this works without a hitch. When we consider subtyping explicitly, we need to make sure we view inference systems on types as being defined *co-inductively*. Since a co-inductively defined judgment essentially expresses the absence of a counterexample, this is exactly what we need for the operational properties like progress, preservation, or absence of blame. We therefore adopt this view.

Recursive processes In addition to recursively defined types, we also need recursively defined processes. We follow the general approach of Toninho et al. [37] for the integration of a (functional) data layer into session-typed communication. A process can be named p, ascribed a type, and be defined as follows.

$$\begin{array}{l} p : \forall n_1{:}\tau_1.\ \ldots\ \forall n_k{:}\tau_k.\{A \leftarrow A_1, \ldots, A_m\} \\ x \leftarrow p\, n_1\ \ldots\ n_k \leftarrow y_1, \ldots, y_m = P \end{array}$$

where we check $(n_1{:}\tau_1, \ldots, n_k{:}\tau_k)\ ;\ (y_1{:}A_1, \ldots, y_m{:}A_m) \vdash P :: (x : A)$

We use such process definitions when spawning a new process with the syntax

$$c \leftarrow p\, e_1\ \ldots\ e_k \leftarrow d_1, \ldots, d_m\ ;\ Q$$

which we check with the rule

$$\frac{\begin{array}{cc} (\Phi \vdash e_i : \tau_i)_{i \in \{1,\ldots,k\}} & \Theta = [e_1/n_1, \ldots, e_k/n_k] \\ \Delta' = [\Theta](d_1{:}A_1, \ldots, d_m{:}A_m) & \Phi\ ;\ \Delta, c : [\Theta]A \vdash Q :: (d : D) \end{array}}{\Phi\ ;\ \Delta, \Delta' \vdash c \leftarrow p\, e_1 \ldots e_k \leftarrow d_1, \ldots, d_m\ ;\ Q :: (d : D)}\ \mathsf{pdef}$$

After evaluating the value arguments, the call consumes the channels d_j (which will not be available to the continuation Q, due to linearity). We note that Θ is a sequence of substitutions and the notation $[\Theta]A$ applies the sequence of subsitutions to the session type A. The continuation Q will then be the (sole) client of c and the new process providing c will execute $[c/x][d_1/y_1]\ldots[d_m/y_m]P$.

One more quick shorthand used in the examples: a tail-call $c \leftarrow p\ \overline{e} \leftarrow \overline{d}$ in the definition of a process that provides along c is expanded into $c' \leftarrow p\ \overline{e} \leftarrow \overline{d}\ ;\ c \leftarrow c'$ for a fresh c'. Depending on how forwarding is implemented, however, the expanded version may be less efficient than the tail call [18].

Stopping computation Finally, in order to successfully monitor computation, we need the capability to assert conditions and stop the computation at particular points. We add assert statements to check conditions on observable values and an abort action to stop computation. We tag the assert and abort blocks with a label l which allows us to determine which assertion failed when the computation aborts. The semantics are given below and the typing is in Figure 2.3.

$$\begin{aligned}
&\mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{True}; Q) \longrightarrow \mathsf{proc}(c, Q)\\
&\mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{False}; Q) \longrightarrow \mathsf{abort}(l)\\
&\mathsf{proc}(c, \mathsf{abort}\ l) \longrightarrow \mathsf{abort}(l)
\end{aligned}$$

We overload the $\mathsf{abort}(l)$ notation to refer to both the semantic construct (as shown above) and the process expression (used frequently in our examples in Section 5). Progress and preservation were proven for the above system, with the exception of the abort and assert rules, in prior work [29]. The additional proof cases do not change the proof significantly.

We summarize the process expressions in Figure 2.1, the semantic rules in Figure 2.2, and the typing rules in Figure 2.3. The typing and semantic rules presented so far in this chapter are restricted to contexts that consist of linear channels. In this setting, process communication forms a tree at runtime because a client of a process must be the only client of that process. In the next section, we will augment our system to support nonlinear channels.

Shared Channels So far in this chapter we have discussed a system where every process is linear and provides a service along exactly one channel, but has the ability to be a client of multiple channels. This guarantees that there is exactly one channel between each pair of processes. Consider the case of a web service with multiple clients. In this situation, it does not make sense to have the web service be a linear process offering database services along a linear channel. Rather, the web service should be encoded as a persistent resource that can be replicated to create linear copies for linear clients to communicate with. We accomplish this "copying" by use of shifts which shift between the unrestricted (persistent) and linear modalities.

Shared channels introduce some complications into the operational semantics since processes offering along such channels may have multiple clients and are "replicating". We restrict the syntax so that there are no unrestricted propositions except for $\uparrow^U_L A^+_L$ which shifts from the linear layer to the unrestricted layer. In this case, the only relevant operational rules are for identity, cut, and the rules for up and down shifts. We start with the shift rules. In the substructural operational semantics, any persistent proposition is preceded by an exponential modality (!). When occurring on the left-hand side of a rule, the corresponding persistent proposition is not

$P, Q, R ::=$	
$\mathsf{close}\ c$	send end and terminate
$\mathsf{wait}\ c\ ;\ Q$	recv end, continue with Q
$\mathsf{send}\ c\ a\ ;\ Q$	send channel a along c, and continue as Q
$x \leftarrow \mathsf{recv}\ c\ ;\ Q$	receive a along c, continue as Q
$c.\ell_j\ ;\ Q$	send ℓ_j along c, continue as Q
$\mathsf{case}\ c\ \mathsf{of}\ \{\ell_i \Rightarrow Q_i\}_i$	recv ℓ_j along c, cont. as Q_j
$\mathsf{send}\ c\ v\ ;\ Q$	send value v along c, continue as Q
$n \leftarrow \mathsf{recv}\ c\ ;\ Q$	recv value v along c, continue as Q
$\mathsf{send}\ c\ \mathsf{shift}\ ;\ Q$	send shift along c, continue as Q
$\mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q$	receive shift along c, continue as Q
$x \leftarrow P\ ;\ Q$	create new a, spawn P, continue as Q
$c \leftarrow d$	connect c with d and terminate
$\mathsf{assert}\ l\ p\ ;\ Q$	assert predicate p with label l and continue as Q
$\mathsf{abort}\ l$	abort with label l

Figure 2.1: Process Expressions

consumed, but instead remains. When occurring on the right-hand side of a rule, a corresponding persistent proposition is created in the state.

The two most interesting rules are $\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s}$ and $\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{r}$. When a down shift message is sent, the persistent process providing a service on channel c_U replaces the process which was providing a service on channel c_L. When an up shift message is received, a fresh ephemeral process $\mathsf{proc}(c_L, P)$ is spawned while the original persistent process continues to exist. The id_U rule looks different than its linear variant because forwarding between shared channels will always be connected to a message process. Therefore, the forwarding rule simply updates the message. Finally, the cut rule is similar to its linear analog with the caveat that the newly spawned process is unrestricted. The semantics are given in Figure 2.4 and the typing is shown in Figure 2.5. The most important feature of this typing judgement is that $\Phi \vdash P :: (x_r : A_r)$ presupposes $\Phi \geq r$. For example, in the cut_U typing rule, the context Φ must be unrestricted while the context Φ' could either be linear or unrestricted. We also assume that $r ::= U \mid L$ where $U > L$.

2.2 Contracts for Functions

We root our discussion of contracts in a process-based setting by first reviewing how contracts are defined and used in a functional setting. A contract for a function can be modeled by an expressive type that places constraints on the arguments and return value. As the function executes, this contract is checked at runtime. For example, consider a function f where both the argument and the return value must be positive. We first write the standard type:

$$f : \mathsf{int} \to \mathsf{int}$$

We can now define a more precise type $\mathsf{posInt} = \{x : \mathsf{int} \mid x > 0\}$. This type is a refinement of the integer type and can be used to express the desired contract.

$$f : \mathsf{posInt} \to \mathsf{posInt}$$

If an argument to f is not positive, then f's caller is blamed for the contract violation. Symmetrically, if f's result is not positive, the blame falls on f itself. Unfortunately, this simple approach to contract checking fails to generalize to a higher-order setting. Consider the following function:

$$g : (\mathsf{posInt} \to \mathsf{posInt}) \to \mathsf{posInt}$$

The contract's domain accepts functions that map positive integers to positive integers. The contract's range obliges g to produce positive numbers. The function g could be passed a function $f : \mathsf{posInt} \to \mathsf{posInt}$ which matches g's domain or a function that has a stricter domain such as $f : \mathsf{posInt} \to \{x : \mathsf{posInt} \mid x > 10\}$. The key insight here is that a contract checker cannot determine if g's argument meets its contract when g is called. It must wait until this argument, say the function f, is applied to another argument to validate its contract. This notion of deferred contract checking for higher-order functions, first introduced by Findler and Felleisen [14], allows the contract checker to assign blame to the party that actually violated the contract. When performing contract-checking for session-types in a higher-order setting (described in Section 3), we use a similar approach.

Contracts are frequently used to prescribe the interactions between code typed with different levels of precision. In an extreme case, contracts can be used to integrate code that is typed with code that is untyped to ensure that dynamically-typed code maintains statically-typed invariants. In this dissertation, we frequently use contracts to connect session types and refinements of those types.

Wadler and Findler [39] define a type system with casts, called a blame calculus, where casts represent contracts. In their system, a contract is modeled as a cast from a source type S to a target type T with a blame label p. The source term s has type S while the whole term has type T.

$$\langle T \Leftarrow S \rangle^p s$$

In this situation, blame is assigned to the label p when the term contained in the cast, s in this example, fails to satisfy the contract associated with the cast. Conversely, the complement of p, written $\overline{p}$ is blamed, when the context containing the cast fails to satisfy the contract. A cast will be dynamically checked to validate whether a given value can be coerced to the required type.

Consider the following cast which takes a function with a domain and range of type int to a more precise domain and range.

$$(\langle \mathsf{posInt} \to \mathsf{posInt} \Leftarrow \mathsf{int} \to \mathsf{int} \rangle^{p_1} f)x$$

When this cast is evaluated, it will be broken into two casts, one for the range and one for the domain. The cast for the range of the function will attempt to cast the range of the source to the range of the target as follows: $\langle \mathsf{posInt} \Leftarrow \mathsf{int} \rangle^{p_1}$. The cast for the domain of the function will attempt to cast the domain of the target to the domain of the source as follows: $\langle \mathsf{int} \Leftarrow \mathsf{posInt} \rangle^{\overline{p_1}}$.

Preserving order for the range and reversing order for the domain is similar to the standard approach to function subtyping which is covariant in the range and contravariant in the domain.

The range cast retains the blame label p_1 because if this cast fails it is the fault of the function f. For example, a function f that maps all integer inputs to the integer -1 will cause the cast to fail and trigger the blame label. The blame label for the domain cast is the complement of the original blame label p_1 because if this cast fails, then it is the fault of the context for supplying an invalid argument x to the function f. However, we note that the type checker will guarantee that x has type posInt and the cast from posInt to int will always suceed. This means that the blame label $\overline{p_1}$ will never be blamed. The only situation where blame can occur is if the range cast fails to cast the less precise type int to the more precise type posInt with blame label p_1.

Consider the following cast which takes a function with a domain and range of type posInt to a less precise domain and range.

$$(\langle \mathsf{int} \to \mathsf{int} \Leftarrow \mathsf{posInt} \to \mathsf{posInt} \rangle^{p_2} f)x$$

As shown above, this cast will decompose into a cast for the range, $\langle \mathsf{int} \Leftarrow \mathsf{posInt} \rangle^{p_2}$, and domain, $\langle \mathsf{posInt} \Leftarrow \mathsf{int} \rangle^{\overline{p_2}}$, of the function. The range cast will always succeed, so the blame label p_2 will never be blamed. The only situation where blame can occur is if the domain cast fails to cast the less precise type int to the more precise type posInt with blame label $\overline{p_2}$. In this situation, the blame lies with the context containing the cast, as opposed to the cast itself.

In both of these instances, Wadler and Findler [39] prove that blame always lies with the less-precisely typed code. When validating contracts expressed as type refinements, described in Section 5, we prove a similar theorem. More comprehensive theorems about the correctness of blame assignment have been proposed by Dimoulas et al. [10, 11]. Subsequent work on gradual typing that considers systems with both static and dynamic typing also uses "blame always lies with the less-precisely typed code" as a criteria for correctness. For instance, Ahmed et al. [2] developed a blame calculus for a language that integrates parametric polymorphism with static and dynamic typing. Fennell and Thiemann [13] proved a blame theorem for a linear lambda calculus with type Dynamic. Most recently, Wadler [38] surveys the history of the blame calculus and presents the latest developments. Keil and Thiemann [22] develop a blame assignment for higher order contracts that includes intersection and union contracts. Siek et al. [31] develop three calculi for gradual typing and relate them in an effort to unite the concepts of blame and coercion. In the next chapter, we explore how we can adapt notions of blame assignment gleaned from a functional setting to a session-typed setting.

cut	:	$\mathsf{proc}(c, x{:}A \leftarrow\ P\ ;\ Q)$ $\longrightarrow \mathsf{proc}(a, [a/x]P), \mathsf{proc}(c, [a/x]Q)$ *(a fresh)*
plus_s	:	$\mathsf{proc}(c, c.k\ ;\ P)$ $\longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, c.k\ ;\ c \leftarrow c')$ *(c' fresh)*
plus_r	:	$\mathsf{msg}(c, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \longrightarrow \mathsf{proc}(d, [c'/c]Q_k)$
with_s	:	$\mathsf{proc}(d, c.k\ ;\ Q)$ $\longrightarrow \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q)$ (c' fresh)
with_r	:	$\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P_k)$
tensor_s	:	$\mathsf{proc}(c, \mathsf{send}\ c\ a\ ;\ P)$ $\longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c')$ *(c' fresh)*
tensor_r	:	$\mathsf{msg}(c, \mathsf{send}\ c\ a\ ;\ c \leftarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c][a/x]Q)$
lolli_s	:	$\mathsf{proc}(d, \mathsf{send}\ c\ a\ ;\ Q)$ $\longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q)$ (c' fresh)
lolli_r	:	$\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ a\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][a/x]P)$
close	:	$\mathsf{proc}(c, \mathsf{close}\ c) \longrightarrow \mathsf{msg}(c, \mathsf{close}\ c)$
wait	:	$\mathsf{msg}(c, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, Q)$
exists_s	:	$\mathsf{proc}(c, \mathsf{send}\ c\ v\ ;\ P)$ $\longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ v\ ;\ c \leftarrow c')$
exists_r	:	$\mathsf{msg}(c, \mathsf{send}\ c\ v\ ;\ c \leftarrow c'), \mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c][v/n]Q)$
forall_s	:	$\mathsf{proc}(d, \mathsf{send}\ c\ v\ ;\ Q)$ $\longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ v\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q)$
forall_r	:	$\mathsf{proc}(c, n \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ v\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c][v/n]P)$
down_s	:	$\mathsf{proc}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ P)$ $\longrightarrow \mathsf{proc}(c', [c'/c]P), \mathsf{msg}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c')$ (c' fresh)
down_r	:	$\mathsf{msg}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c'/c]Q)$
up_s	:	$\mathsf{proc}(d, \mathsf{send}\ d\ \mathsf{shift}\ ;\ Q)$ $\longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c'/c]Q)$
up_r	:	$\mathsf{proc}(c, \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c'/c]P)$
assert_f	:	$\mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{False}; Q) \longrightarrow \mathsf{abort}(l)$
assert_s	:	$\mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{True}; Q) \longrightarrow \mathsf{proc}(c, Q)$
abort	:	$\mathsf{proc}(c, \mathsf{abort}\ l) \longrightarrow \mathsf{abort}(l)$

Figure 2.2: Linear Process Semantics

$$\frac{}{\Psi\ ;\ b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id} \qquad \frac{\Psi\ ;\ \Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Psi\ ;\ \Delta, \Delta' \vdash x{:}A \leftarrow P\ ;\ Q :: (c : C)}\ \mathsf{cut}$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^+)}{\Psi\ ;\ \Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \uparrow A^+)}\ \uparrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^+ \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \uparrow A^+ \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q :: (d : D)}\ \uparrow L$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^-)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ P :: (c : \downarrow A^-)}\ \downarrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^- \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \downarrow A^- \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \downarrow L$$

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Psi; \Delta \vdash Q :: (d : D)}{\Psi; \Delta, c : \mathbf{1} \vdash \mathsf{wait}\ c\ ;\ Q :: (d : D)}\ \mathbf{1}L$$

$$\frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R \qquad \frac{\Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{:}\tau.\ A \vdash n \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L$$

$$\frac{\Psi, n{:}\tau\ ;\ \Delta \vdash P :: (c : A)}{\Psi\ ;\ \Delta \vdash n \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \forall n{:}\tau.\ A)}\ \forall R \qquad \frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \forall n{:}\tau.\ A \vdash \mathsf{send}\ c\ v\ ;\ Q :: (d : D)}\ \forall L$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : B)}{\Psi\ ;\ \Delta, a : A \vdash \mathsf{send}\ c\ \ a\ ;\ P :: (c : A \otimes B)}\ \otimes R$$

$$\frac{\Psi; \Delta, x : A, c : B \vdash Q :: (d : D)}{\Psi; \Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \otimes L$$

$$\frac{\Psi; \Delta, x : A \vdash P :: (c : B)}{\Psi; \Delta \vdash x \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : A \multimap B)}\ \multimap R$$

$$\frac{\Psi\ ;\ \Delta, c : B \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, a : A, c : A \multimap B \vdash \mathsf{send}\ c\ a\ ;\ Q :: (d : D)}\ \multimap L$$

$$\frac{\Psi; \Delta \vdash P_\ell :: (c : A_\ell) \quad \text{for every } \ell \in L}{\Psi; \Delta \vdash \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (c : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L \quad \Psi; \Delta, c : A_k \vdash Q :: (d : D)}{\Psi; \Delta, c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\ ;\ Q :: (d : D)}\ \&L$$

$$\frac{k \in L \quad \Psi; \Delta \vdash P :: (c : A_k)}{\Psi; \Delta \vdash c.k\ ;\ P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Psi; \Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Psi; \Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

Figure 2.3: Typing Linear Process Expressions

$\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s}$: $\mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q)$
$\longrightarrow \mathsf{proc}(a, [c_L/x_L]Q), \mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L) \quad (c_L\ fresh)$

$\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{r}$: $\mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L), !\mathsf{proc}(c_U, \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P)$
$\longrightarrow \mathsf{proc}(c_L, [c_L/x_L]P)$

$\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s}$: $\mathsf{proc}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q)$
$\longrightarrow !\mathsf{proc}(c_U, [c_U/x_U]Q), \mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U) \quad (c_U\ fresh)$

$\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{r}$: $\mathsf{msg}(c_U, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_k; c_U \leftarrow x_U), \mathsf{proc}(a, \mathsf{shift}\ x_U \leftarrow \mathsf{recv}\ c_L; P)$
$\longrightarrow \mathsf{proc}(a, [c_U/x_U]P)$

id_{U} : $\mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L), !\mathsf{proc}(a_U, a_U \leftarrow b_U)$
$\longrightarrow \mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ b_U; a_L \leftarrow x_L)$

$\mathsf{cut}_{\mathsf{U}}$: $\mathsf{proc}(c, x_U \leftarrow P; Q)$
$\longrightarrow \mathsf{proc}(c, [a_U/x_U]Q), !\mathsf{proc}(a_U, [c_U/x_U]P) \quad (a_U\ fresh)$

Figure 2.4: Shared Process Semantics

$$\frac{}{\Phi, y_U{:}A_U \vdash (x_U \leftarrow y_U) :: (x_U{:}A_U)}\ \mathsf{id}_{\mathsf{U}}$$

$$\frac{\Phi \geq U \geq r \quad \Phi \vdash P :: (x_U{:}A_U) \quad \Phi', x_U{:}A_U \vdash Q :: (z_r{:}C)}{\Phi, \Phi' \vdash (x_U : A \leftarrow P\ ;\ Q) :: (z_r{:}C)}\ \mathsf{cut}_{\mathsf{U}}$$

$$\frac{\Phi \vdash P :: (x_L{:}A^+_L)}{\Phi \vdash (\mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ x_U\ ;\ P) :: (x_U{:}\uparrow^U_L A^+_L)}\ \uparrow\mathsf{R}$$

$$\frac{\Phi, x_L{:}A^+_L \vdash Q :: (z_L{:}C)}{\Phi, x_U{:}\uparrow^U_L A^+_L \vdash (\mathsf{shift}\ x_L \leftarrow \mathsf{send}\ x_U\ ;\ Q) :: (z_L{:}C)}\ \uparrow\mathsf{L}$$

$$\frac{\Phi \vdash Q :: (x_U{:}A^-_U)}{\Phi \vdash (\mathsf{shift}\ x_U \leftarrow \mathsf{send}\ x_L\ ;\ Q) :: (x_L{:}\downarrow^U_L A^-_U)}\ \downarrow\mathsf{R}$$

$$\frac{\Phi, x_U{:}A^-_U \vdash P :: (z_r{:}C)}{\Phi, x_L{:}\downarrow^U_L A^-_U \vdash (\mathsf{shift}\ x_U \leftarrow \mathsf{recv}\ x_L\ ;\ P) :: (z_r{:}C)}\ \downarrow\mathsf{L}$$

All judgments $\Phi \vdash P :: (x_r : A_r)$ presuppose $\Phi \geq r$
Φ, Φ' contexts allows unrestricted $x_{\mathsf{U}}{:}A_{\mathsf{U}}$ to be shared between Φ and Φ'

Figure 2.5: Shared Process Typing

Chapter 3

Session Types as Contracts

In a concurrent setting, there are two important reasons to consider dynamic monitoring of communication. The first is that when spawning a new process, part of the execution of a program now escapes immediate control of the original process. If the new process is compromised by a malicious intruder, then incorrect yet unchecked messages can wreak havoc on the original process. Second, session types allow us to safely connect communicating processes written in a variety of different languages, as long as they (dynamically!) adhere to the session protocol and basic data formats. Static checking would require having access to the internal code of these diverse processes, which is impractical.

In this chapter, we present a model that dynamically monitors communication to enforce adherence to session types in a higher-order setting. We place a monitor on each channel that checks whether messages are consistent with the communication contract on that channel. If the message is determined to violate the contract, the monitor raises an alarm. When an alarm is raised, we are able to assign blame and prove one of an indicated set of possible culprits must have been compromised. We are able to provide blame assignment in a higher-order setting where channels pass channels of arbitrary type to other channels. We also prove that dynamic monitoring does not change system behavior for well-typed processes.

3.1 Model

In this section, we discuss the adversary and trust model and explain the monitor design. We then formally define the operational semantics for the monitoring mechanism and for the blame assignment.

Trust Model We assume that processes are distributed across the network and communicate with each other by message-passing. We assume that there is a secure (trusted) network layer which ensures that messages are sent and received without error. In contrast, *all processes are untrusted*; any process could be compromised by an attacker. We define an "attack" scenario when a process deviates from its prescribed session type. We use runtime monitors to detect such deviations and attribute blame to rogue processes.

Monitor Capabilities We assume that the monitor can inspect communications between processes to check session fidelity, but it cannot observe internal operations of the executing processes. Only send, receive, spawn (cut), and forward (identity) requests can be seen by the monitor. This design decision is important because it allows our monitoring techniques to be applied in the situation where we make no assumptions about the internal structure of the communicating processes. The monitor is also trusted. We note that our monitor cannot detect when linearity has been violated in the system, which means that a channel may have multiple clients without the monitor's knowledge. In our system, we do make the assumption that every channel has a unique provider.

Monitors can raise alarms and assign blame when messages sent over channels are of the *wrong type*, which we explain in detail below. If a protocol violation is detected and an alarm is raised, the computation is aborted.

Adversary Capabilities We assume that channels are *private* in that only the processes at the two endpoints of a channel can send to or receive from it. Further, channel names are capabilities that are hard to forge. An attacker only knows the channel names that are given to it by the trusted runtime (e.g., through spawning a new process).

We define the following transition rule (named havoc) to represent an attacker's action of taking control of a process. The attacker replaces the original process with one of the attacker's choice. However, the attacker cannot forge channel names, and therefore, the set of free channel names in Q is a subset of that in P.

$$\mathsf{havoc} : \mathsf{proc}(c, P) \otimes !(\mathsf{fn}(P) \supseteq \mathsf{fn}(Q)) \longrightarrow \mathsf{proc}(c, Q)$$

Finally, because processes are untrusted, they cannot raise an alarm.

Monitor Design First, we examine several possible design choices for the monitor and explain our chosen design. Figure 3.1 illustrates multiple monitoring architectures. In this graphic, two processes communicate over channel c by exchanging messages. Trusted monitoring components have a grey background. We assume that process P is offering a service along channel a. The monitor is tasked with mediating communication that flows through channel c. The first row of the figure shows an architecture with no monitoring.

The second row of the figure shows a design where a partial identity process acts as the monitor and mediates communication between the two processes. Here, the monitoring process M relays messages between processes P and Q. Because process P is providing a service of type A, the process $\mathsf{proc}(m, M)$ is a partial identity process from A to A. In this design, as soon as a new process is spawned, an accompanying monitoring process is also generated. The advantage of this design is that the monitor is a process defined within the same language as the rest of the system. The drawback of this approach is that blame assignment becomes difficult. We explore the benefits of this monitoring architecture in Chapter 4.

The third row of the figure demonstrates a design where a monitor is placed directly on the communication channel c. The monitor keeps track of the type of the channel and checks the message pattern. The advantage of this approach is that when a process attempts to send

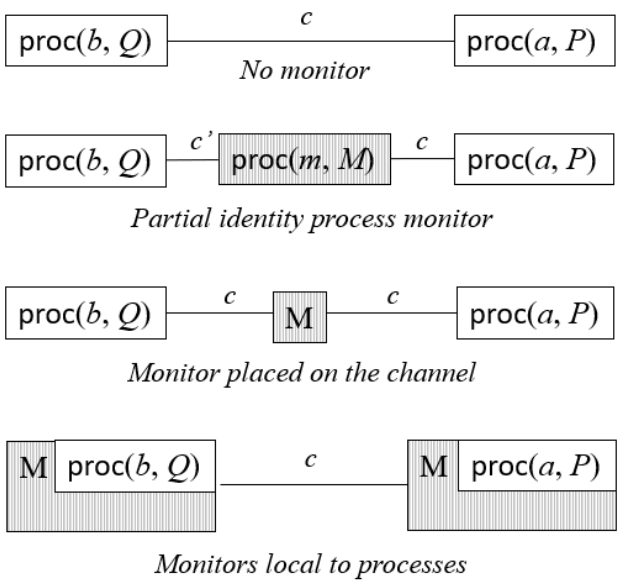

Figure 3.1: Monitoring Architectures

messages of the wrong type, the monitor will raise an alarm before the message reaches its destination. This leads to more precise blame attribution than the first approach.

The last row of the figure presents a design where monitors are local to each process, which may be beneficial in a mutually untrusting environment. When a process receives a message from an untrusted source, its own monitor checks the messages before allowing the process to have access to the messages. This approach fits the distributed model more naturally, as the monitor is local to each process. However, this approach suffers from a similar problem as the partial identity approach: the blame assignment is difficult. Moreover, since the monitoring infrastructure is distributed, a separate mechanism would be needed to somehow verify alarms and blame processes which complicates the monitor design. After considering these different approaches, we chose the approach that places monitors on the communication channels (shown in the third row of the figure) because it is simple and provides relatively precise blame assignment. Next, we define the formal monitor semantics.

Monitor Semantics We have two variations of the monitoring semantics that allow us to provide different levels of precision in our blame assignment based on the assumptions we make. In the *Verified-Spawn* semantics, we assume that a spawned process can be statically typechecked against a given type to ensure that its type matches the type of the channel it is being spawned on. If the spawned processes is not well-typed, all computation aborts. If this check succeeds, then we are able to immediately absolve the spawning process because our monitor has verified that the spawned process adheres to its prescribed type. The ability to determine if a process has come into existence with the right type allows us to provide more precise blame assignment.

More specifically, once an alarm is raised, instead of having to concern ourselves with whether a particular child process process may have been spawned incorrectly by the parent process, and having to blame both the parent and the child processes, we can just blame the child process. In the *Unverified-Spawn* semantics, we assume that we do not have access to the source code of a process being spawned before it starts executing. This assumption is conservative and treats all processes as black boxes. With these assumptions, our blame assignment consists of a set of processes, where one of the indicated processes must have made a havoc transition and triggered the alarm. Once an alarm is raised, we cannot assume that a child process was spawned correctly by its parent, but rather we must place all of its ancestor processes into the blame set. This chapter will focus on the *Verified-Semantics*; a discussion of the *Unverified Semantics* can be found in Section 3.4.

We write m to denote message patterns. We define $m \rhd A$ to mean that a message pattern m is compatible with type A. It describes the pattern of the message that a channel of type A expects to receive (defined below).

$$
\begin{array}{ll}
\mathsf{ch}(a) \rhd \mathsf{A}^+ \otimes \mathsf{B}^+ & \mathsf{ch}(a) \rhd \mathsf{A}^+ \multimap \mathsf{B}^- \\
\mathsf{lab}(\mathit{lab}_j) \rhd \oplus\{\mathit{lab}_i : A_i^+\}_i & \mathsf{lab}(\mathit{lab}_j) \rhd \&\{\mathit{lab}_i : A_i^-\}_i \\
\mathsf{shift} \rhd \downarrow A^- & \mathsf{shift} \rhd \uparrow A^+ \\
\mathsf{shift} \rhd \downarrow_L^U A_U & \mathsf{shift} \rhd \uparrow_L^U A_L \\
\mathsf{end} \rhd \mathbf{1} & \\
v \rhd \forall x.\tau.A(x) \text{ if } v : \tau & v \rhd \exists x.\tau.A(x) \text{ if } v : \tau
\end{array}
$$

We augment the operational semantics presented in Section 2.1 (shown in Figure 2.2 and Figure 2.4) to include monitor actions. The rules are shown in Figure 3.3. The monitoring actions, denoted $!(m)$, check that the condition m is true. We use the predicate $\mathsf{typecheck}(P :: x : A)$ to perform a static check that verifies that the process P is providing a service compatible with the channel x of type A. The application of the havoc rule is an internal transition and is not observable by the computation. The havoc rule contains a monitor condition, highlighted in dark gray, that is used to prove properties of the monitor. When the $\mathsf{havoc}(a)$ action executes, the channel a is added to the context H which tracks havoced channels.

In the id rule, the monitor checks that channels a and b have the same type. In the $\mathsf{id_a}$ rule, if any of the above conditions are not met, the system will raise an alarm. In the lolli_s rule, the monitor ensures that both the channels a and c_i have the appropriate types. In order to make it easier to track linear channel usage, the monitor also renames channel $c_i : A_1 \multimap A_2$ to channel $c_{i+1} : A_2$ once the sending step of the computation is complete. Similarly, in the lolli_r rule, the monitor renames channel $c_i : A_1 \multimap A_2$ to channel $c_{i+1} : A_2$. In the cut rule, the monitor validates the spawned process. If the conditions are met, a new process P is spawned on channel a_0. The spawning process continues to execute. In the $\mathsf{cut_a}$ rule, if either the process P is of the wrong type, or (denoted by the disjoint sum $\oplus$) P uses inappropriate channels, an alarm is raised.

Monitoring Shared Channels The monitoring rules for shared channels are shown in Figure 3.4. In the $\mathsf{cut_U}$ rule, the monitor checks that the spawned unrestricted process P is providing a service compatible with the channel x of type A. In the $\mathsf{up}_\mathsf{L}^\mathsf{U}$_s rule, the monitor ensures that the

unrestricted channel c_U has the appropriate type. In the $\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}$_s rule, the monitor ensures that the linear channel c_L has the appropriate type. We note that in the $\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}$_r rule, a new linear process P is spawned.

Blame assignment When an alarm ($\mathsf{alarm}(a)$) is raised, the monitor assigns blame to exactly one process that provides a service along channel a. Informally, exactly that one process must have "havoced"; otherwise, type preservation will ensure that no alarm is raised.

Though we rename channels at every step of the computation, we are able to blame a single channel. This is the case because each time a channel a_i is renamed, its index is incremented and it is now called a_{i+1}. Therefore, when channel a_i is blamed, we can collapse all of the a_i's by just erasing the index and just blame channel a.

$\mathsf{CameraFun} : \{\mathsf{Cam}\}$
$c \leftarrow \mathsf{CameraFun} =$
  $\mathsf{case}\ c\ \mathsf{of}$
  $\mid\ \mathsf{take} \Rightarrow pm \leftarrow \mathsf{recv}\ c\ ;$
    $\mathsf{case}\ pm\ \mathsf{of}$
    $\mid\ \mathsf{once} \Rightarrow x \leftarrow \mathsf{recv}\ pm\ ;\ \mathsf{wait}\ pm\ ;\ picH \leftarrow \mathsf{takePic}\ ;\ \mathsf{send}\ c\ picH\ ;\ c \leftarrow \mathsf{CameraFun}$

$\mathsf{ToSnap} : \{\mathsf{Snap} \leftarrow \mathsf{User}\ ;\ \mathsf{Cam}\}$
$s \leftarrow \mathsf{ToSnap} \leftarrow u, c =$
  $c.\mathsf{take}\ ;\ u.\mathsf{picPerm}\ ;$
  $\mathsf{case}\ u\ \mathsf{of}$
  $\mid\ \mathsf{fail} \Rightarrow c.\mathsf{fail}\ ;\ s \leftarrow \mathsf{ToSnap} \leftarrow u, c$
  $\mid\ \mathsf{succ} \Rightarrow c.\mathsf{succ}\ ;\ s.\mathsf{share}\ ;\ perm \leftarrow \mathsf{recv}\ u\ ;\ \mathsf{send}\ c\ perm\ ;$
    $picH \leftarrow \mathsf{recv}\ c\ \ ;\ \mathsf{send}\ s\ picH\ ;\ s \leftarrow \mathsf{ToSnap} \leftarrow u, c$

Figure 3.2: Snapchat

3.2 Examples

In this example, we illustrate monitoring with a mobile photosharing application, `Snapchat`, that takes and shares a user's photos and sends them to some remote entity. To take photos, `Snapchat` needs to operate the camera. To prevent the `Snapchat` application from continuously taking and sharing the user's photos, the camera requires that the user grant `Snapchat` permission every time `Snapchat` wants to take and share a photo. This example contains three main processes: the `Snapchat` application process, the camera process, and the user process.

Types and Encoding We encode the expected behavior of each process as a session type declaration below.

$\mathsf{stype}\ \ \mathsf{Cam} = \&\{\mathsf{take} : \mathsf{photoPerm}$
$\qquad\qquad\qquad\qquad \multimap(\mathsf{picHandle} \otimes \mathsf{Cam})\}$

$\mathsf{stype}\ \ \mathsf{User} = \&\{\mathsf{picPerm} : \oplus\{\mathsf{fail} : \mathsf{User};$
$\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{succ} : \mathsf{photoPerm} \otimes \mathsf{User}\}\}$
$\mathsf{stype}\ \ \mathsf{photoPerm} = \oplus\{\mathsf{once} : \exists x : \diamond_U \mathsf{ok}.\mathbf{1}\}$
$\mathsf{stype}\ \ \mathsf{Snap} = \oplus\{\mathsf{share} : \mathsf{picHandle} \otimes \mathsf{Snap}\}$

Camera After the client selects take, the camera process waits for the client to send a photo permission channel (of type photoPerm). Upon receiving a channel *pm*, the camera process receives a signature from the channel *pm*. In this example, the camera process does not validate the signature of the permission by itself, but instead relies on the monitor to check the signature, which we will explain when we discuss the monitoring scenarios. After the request succeeds and the picture handle is sent, the camera process continues to offer a service of type Cam.

User When a process needs permission to access the camera, it communicates with the user process and selects the picPerm label. If the user sends the fail label, the user process continues to offer a service of type User, without granting its client permission to use the camera. If the user process sends a succ label, it then spawns a new process that provides a service of type photoPerm and sends the new process' channel to its client. The type photoPerm is an internal choice, labeled once. The newly spawned process first sends the label once, then sends a digital signature of a token ok (of type ok) using the camera's private key, before it terminates. The digital signature serves as an unforgeable authentication token for a permission to access the camera once.

The code snippet that corresponds to the permission process spawned by the user is given below.

send *pm* once ;
send (*sign* K_priv(U) ok) ;
close *pm* ;

The function *sign* will use the user's private key to sign the abstract type ok. We assume that this permission process has access to the user's private key as it is spawned by the user process.

Snapchat The ToSnap process uses channel c to communicate with the camera process and channel u to communicate with the user process and offers the picture sharing service along channel s. The process first instructs the camera to take a picture and then asks the user process for permission. If the user does not grant the permission, no picture is sent and the `ToSnap` process continues to try and send a picture. If the user grants the permission, the ToSnap process sends its client the label share. It then receives a channel connecting to a permission process from the user, and forwards this channel to the camera. Finally, the ToSnap process receives a picture handle from the camera, and sends it to `ToSnap`'s client.

The CameraFun and ToSnap processes are shown in Figure 3.2. The takePic process provides access to the picture via a picture handle, which is also modeled by a channel.

Monitoring Scenarios We show two monitoring scenarios to demonstrate how our monitor can detect violations of invariants specified by the session types. In these scenarios, an attacker

tries to take pictures without being granted permissions required by the camera.

Scenario 1 The `ToSnap` process is compromised by an attacker. The havoced process does not ask for permission from the user and instead of sending a permission to the camera, sends an integer value (i.e. replacing lines $perm \leftarrow \mathsf{recv}\ u\ ;\ \mathsf{send}\ c\ perm$ of the ***ToSnap*** process with $\mathsf{send}\ c\ n$).

The monitor on channel c is expecting a value of type $\mathsf{PhotoPerm}$ which should be a channel. The monitor will try to typecheck the integer n as a channel and fail. It will then raise an alarm ($\mathsf{alarm}(s)$). Here, blame is assigned to one process, ToSnap (offering along channel s).

Scenario 2 `Snapchat` is again compromised. Instead of asking for permission, it tries to spawn a permission process that will match the signature of an authentic permission process. The desired signature is $\mathsf{photoPerm} = \oplus\{\mathsf{once} : \exists x : \diamond_U \mathsf{ok}.\mathbf{1}\}$. When this process is spawned, the monitor will typecheck it against this signature. Because the Snapchat process does not have the user's key, it is unable to generate a process that will be able to send a value of type $\diamond_U \mathsf{ok}$. Therefore, the monitor will raise an alarm and assign blame to the ToSnap which is offering a service along channel s. In the *Unverified Spawn* semantics, the blame assignment will be different. We will return to this example in Section 3.4.

3.3 Metatheory

We identify three high-level properties that the monitor should satisfy: correctness of the blame assignment, the fact that well-behaved processes are not blamed, and transparency of the monitor.

The correctness of the blame assignment is defined as follows. Let the context Ω be the multiset of processes and messages describing the current state of computation. The context H stores channels that have been havoced. Due to channel renaming, if $a_i \in H$, then $a \in H$. We write $\models \Omega : wf$ to denote that all the process and messages comprising the context Ω must be typed using the typing rules in Figure 2.3 and Figure 2.5.

Definition 1 (Correctness of blame)**.** *A channel a is correct to be blamed w.r.t. the execution trace $\mathcal{T} = \Omega \longrightarrow^* \Omega', \mathsf{alarm}(a)$ with $\models \Omega : wf$ if the process providing a service on a_i where $i \in \mathbb{N}$ has made a havoc transition in $\mathcal{T}$.*

The fact that our monitor assigns blame correctly (Definition 1) is a corollary of Theorem 1.

We now present the configuration typing in order to state the theorem. We assume that the comma operator is associative with $\cdot$ as the unit. The context Λ can either be the context Δ which exclusively consists of linear channels, or the context Φ which can also contain unrestricted channels.

We note that when typechecking $H; \Lambda \Vdash \mathsf{proc}(c, P)$ the context Λ contains a mapping of channels to their types as well as persistent monitor conditions such as $!(c : A)$. However, when the context Λ is used to typecheck a process such as $\Lambda \vdash P :: (c : A)$, we use the mapping of channels to their types and do not duplicate the persistent conditions.

$$\Lambda = \Delta \mid \Phi$$

$$\mathcal{C} = \cdot \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, P) \mid \mathcal{C}_1, \mathcal{C}_2$$

$$\frac{}{H; \cdot \Vdash \cdot} \quad \frac{H; \Lambda \Vdash \mathcal{C}_1 \quad H; \Lambda \Vdash \mathcal{C}_2}{H; \Lambda \Vdash \mathcal{C}_1, \mathcal{C}_2} \quad \frac{\Lambda_{|\mathsf{fn}(P)} \vdash P :: (c : \Lambda(c))}{H; \Lambda \Vdash \mathsf{msg}(c, P)}$$

$$\frac{c \notin H \quad \Lambda_{|\mathsf{fn}(P)} \vdash P :: (c : \Lambda(c))}{H; \Lambda \Vdash \mathsf{proc}(c, P)} \quad \frac{c \in H}{H; \Lambda \Vdash \mathsf{proc}(c, P)}$$

Theorem 1 (Alarm)**.**

1. *If* $\emptyset; \cdot; \cdot \models \Omega$ *and* $\Omega, \emptyset, \longrightarrow^* \Omega'; H; \Lambda$ *then* $H; \Lambda \models \Omega'$
2. *If* $\emptyset; \cdot; \cdot \models \Omega$ *and* $\Omega, \emptyset, \longrightarrow^* \_, H, \mathsf{alarm}(a)$ *then* $a \in H$.

The above theorem states that from an initial configuration, a well-typed configuration can make a series of transitions to either another well-formed configuration, or a state where an alarm is raised on a process a that has been havoced.

Another corollary of Theorem 1 is that well-typed processes are not blamed: if the configuration is well-formed and no process is compromised, then the monitor will not raise any alarm. This is easy to prove because if there is an alarm associated with a, then a must be in H. However, when no process havocs, H remains empty; a contradiction.

The correctness proof for the blame assignment is similar to that of a preservation proof. The key lemma is Lemma 2, which states that if a well-typed configuration makes a transition, then it either steps to another well-formed configuration, or an alarm is raised on a process a that has been havoced. Using this lemma, we can prove Theorem 1 which considers a sequence of transitions. The proof is done by induction on the length of the trace.

Lemma 2 (One-step alarm)**.** *If* $H; \Lambda \Vdash \Omega$ *and* $\mathsf{proc}(a_i, P) \in \Omega$ *then either:*

1. $H, \Omega, \Lambda \rightarrow H, \Omega', \Lambda'$ *and* $H; \Lambda' \Vdash \Omega'$ *or*
2. $H, \Omega, \Lambda \rightarrow \mathsf{alarm}(a)$ *where* $a \in H$ *and* $\Lambda_{|\mathsf{fn}(P)} \not\vdash \mathsf{proc}(a_i, P) :: (a : \Lambda(a))$.

Proof. We prove the lemma by examining each monitoring rule, inverting the typing configuration and applying the typing rules. The proof cases for the linear and shared setting can be found in Appendix A.3 and Appendix A.4 respectively. □

Second, if all processes are well-typed to begin with and no process is compromised at runtime, then the monitor should not raise an alarm. This property shows that a havoc transition is necessary for the monitor to halt the execution and assign blame.

Theorem 3 (Well-typed configurations do not raise alarms)**.** *Given any* $\mathcal{T} = \Omega \longrightarrow^* \Omega'$ *such that* $\models \Omega : wf$ *and* $\mathcal{T}$ *does not contain any havoc transitions, there does not exist an* a *such that* $\mathsf{alarm}(a) \in \Omega'$.

Finally, the monitor should not change the behavior of well-typed processes. We write $\longrightarrow^-$ to denote the operational semantics without the monitor actions, and Ω^- to denote a configuration with the monitor artifacts erased. If the initial configuration is well-typed and no process is compromised, then executing the configuration with and without the monitor should yield the same result.

Theorem 4 (Monitor transparency)**.**

1. *Given a trace $\mathcal{T} = \Omega \longrightarrow^* \Omega'$ such that $\models \Omega : wf$ and $\mathcal{T}$ does not contain any havoc transitions then $\Omega(\longrightarrow^-)^*\Omega'^-$.*
2. *Given a trace $\mathcal{T} = \Omega(\longrightarrow^-)^*\Omega'$ such that $\models \Omega : wf$ and $\mathcal{T}$ does not contain any havoc transitions, then there exists a trace $\mathcal{T}'$ such that $\mathcal{T}' = \Omega \longrightarrow^* \Omega'^-$.*

Our monitor is transparent; it does not alter the behavior of well-formed configurations. The proof is done by examining how each monitor check is applied to well-formed configurations. Since well-formed configurations do not have havoced processes (H is empty), all processes and messages are well-typed. The fact that the monitor checks never fail can be obtained by inverting the typing judgments of the relevant messages and processes.

We note that we define transparency in terms of traces – a trace observed in the monitored semantics implies the same trace must be observed in the non-monited semantics. The first direction of the implication states that for any trace without a havoc transition, executing the trace without the monitoring actions will yield the same result as executing the trace with the monitored actions. The second direction of the implication states that for any trace without a havoc transition that is executed without the monitoring actions, there exists a trace augmented with the monitoring actions that will yield the same result. A more fine grained approach to transparency would consist of defining a bisimulation and requiring that a correspondence between the two semantics be maintained for each individual step of a given trace.

3.4 The Unverified-Spawn Semantics

We now return to the *Unverified-Spawn* system mentioned previously in this chapter. In this system, we cannot check that a newly spawned process is spawned at the correct type. The *Unverified-Spawn* system described in this section is a reformulation of the system presented in Jia et al [21].

Monitoring Rules The *Unverified-Spawn* system differs from the *Verified-Spawn* system only in a few key monitoring rules. These rules are highlighted in Figure 3.5. The core difference consists of needing a graph G to track when a process spawns another process. In the linear cut rule, the monitor is not able to verify that the process P will adhere to the appropriate type, so it adds the fresh linear channel a_0 to the graph G. Similarly, in the unrestricted cut_U rule, the fresh unrestricted channel a_0 is added to the graph G. The $\mathsf{down}^\mathsf{U}_\mathsf{k}\_\mathsf{s}$ rule creates a new persistent process providing a service on the unrestricted channel c_U which replaces the process providing a service on the linear channel c_k. This unrestricted channel is added to the graph G. The most interesting rule is the $\mathsf{up}^\mathsf{U}_\mathsf{k}\_\mathsf{r}$ rule because usually monitoring conditions are only placed on sending rules. However, in this case the persistent process providing a service on channel c_U spawns the linear process providing a service on channel c_k. This linear channel is added to the graph G. As before, the dark gray monitoring blocks are only used to prove the conditions of the monitor.

Blame assignment To assign blame, the monitor maintains a graph data structure that records process spawns throughout the execution of the entire system. We write G to denote the graph

(defined below). The nodes in the graph, denoted N, are provider channel names. After a process offering along channel a spawns a process offering along channel b, an edge $a \rightarrow_{sp} b$ is added to the graph. G is a set of trees.

$$\begin{array}{lrcl} \textit{Process graph} & G & ::= & (N, E) \\ \textit{Edges} & E & ::= & \cdot \mid E, a \rightarrow_{sp} b \end{array}$$

When an alarm ($\mathsf{alarm}(a)$) is raised, the monitor assigns blame to all the direct ancestors of a in the graph G. That is, we find the tree in G that contains a, and let $c_1 \rightarrow_{sp} c_2 \cdots c_n \rightarrow_{sp} a$ be the path in that tree from the root to a. Then, the processes in the set $\{c_1, \cdots, c_n, a\}$ are jointly blamed. Informally, at least one of the processes in that set must have "havoced"; otherwise, type preservation will ensure that no alarm is raised. We write $\models \Omega : wf$ to denote that all the process and messages comprising the context Ω must be typed using the typing rules in Figure 2.3 and Figure 2.5.

As an example, recall Scenario 2 descibed in Section 3.2. In this case, instead of asking for permission, Snapchat (which is offering a service along channel s) tries to spawn a permission process that will match the signature of an authentic permission process. The desired signature is $\mathsf{photoPerm} = \oplus\{\mathsf{once} : \exists x : \diamond_U \mathsf{ok}.\mathbf{1}\}$. This spawn will succeed and spawn a new process:

$$\mathsf{proc}(d, d.\mathsf{once}\ ;\ \mathsf{send}\ d\ (\textit{sign}\ \mathsf{ok})\ ;\ \mathsf{close}\ d)$$

At this point, the graph G is augmented with $s \rightarrow_{sp} d$. When the process d tries to send a signature, because it does not have the user's key, it cannot generate a value v such that $v \rhd \diamond_U \mathsf{ok}$. When the value $(\textit{sign}\ \mathsf{ok})$ is send over channel d, the monitor's check fails and raises an alarm ($\mathsf{alarm}(d)$). The blame is assigned to the set $\{s, d\}$ as G includes $s \rightarrow_{sp} d$, and s is the root.

Definition 2 (Correctness of blame – Unverified-Spawn Semantics)**.** *A set of channels N is correct to be blamed w.r.t. the execution trace $\mathcal{T} = \Omega \longrightarrow^* \Omega', \mathsf{alarm}(a_i)$ with $\models \Omega : \mathsf{wf}$ if there exists a channel $b \in N$ such that the process providing a service on b_i has made as made a havoc transition in $\mathcal{T}$.*

The fact that our monitor assigns blame correctly (Definition 2) is a corollary of Theorem 5.

We now present an augmented version of the configuration typing described earlier in order to state the theorem in the *Unverified-Spawn* setting. As before, the context H stores havoced channels, and the context Λ can either be the context Δ which exclusively consists of linear channels, or the context Φ which can also contain unrestricted channels. In this configuration, we need to add additional rules to be able to check whether a given channel is a descendant of a havoced channel. We also add a rule to type a message process when the channel it is providing on is havoced.

$$\Lambda = \Delta \mid \Phi$$

$$\mathcal{C} = \cdot \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, P) \mid \mathcal{C}_1, \mathcal{C}_2$$

$$\frac{a \in H}{G \vdash a : \mathsf{havoc}} \qquad \frac{\exists a.G \vdash a : \mathsf{havoc} \wedge G(a) \rightarrow_{sp} b}{G \vdash b : \mathsf{havoc}}$$

$$\frac{}{G; H; \cdot \Vdash \cdot} \qquad \frac{G; H; \Lambda \Vdash \mathcal{C}_1 \quad G; H; \Lambda \Vdash \mathcal{C}_2}{G; H; \Lambda \Vdash \mathcal{C}_1, \mathcal{C}_2}$$

$$\frac{G \not\vdash c : \mathsf{havoc} \quad \Lambda_{|\mathsf{fn}(P)} \vdash P :: (c : \Lambda(c))}{G; H; \Lambda \Vdash \mathsf{proc}(c, P)} \qquad \frac{G \vdash c : \mathsf{havoc}}{G; H; \Lambda \Vdash \mathsf{proc}(c, P)}$$

$$\frac{G \not\vdash c : \mathsf{havoc} \quad \Lambda_{|\mathsf{fn}(P)} \vdash P :: (c : \Lambda(c))}{G; H; \Lambda \Vdash \mathsf{msg}(c, P)} \qquad \frac{G \vdash c : \mathsf{havoc}}{G; H; \Lambda \Vdash \mathsf{msg}(c, P)}$$

Theorem 5 (Alarm)**.**

1. *If* $\emptyset; \emptyset; \cdot \models \Omega$ *and* $\Omega, \emptyset, \emptyset \longrightarrow^* \Omega'; G; H$ *then* $G; H; \Lambda \models \Omega'$
2. *If* $\emptyset; \emptyset; \cdot \models \Omega$ *and* $\Omega, \emptyset, \emptyset \longrightarrow^* \_, H, \mathsf{alarm}(a)$ *then* $a \in H$.

The above theorem states that from an initial configuration, a well-typed configuration can make a series of transitions to either another well-formed configuration, or a state where an alarm is raised on a process a that has been havoced.

The correctness proof for the blame assignment is similar to that of a preservation proof. The key lemma is Lemma 6, which states that if a well-typed configuration makes a transition, then it either steps to another well-formed configuration, or an alarm is raised on a process a that has been havoced. Using this lemma, we can prove Theorem 5 which considers a sequence of transitions. The proof is done by induction on the length of the trace.

Lemma 6 (One-step alarm)**.** *If* $G; H; \Lambda \Vdash \Omega$ *and* $\mathsf{proc}(a_i, P) \in \Omega$ *then either:*

1. $G; H, \Omega, \Lambda \to G; H, \Omega', \Lambda'$ *and* $G; H; \Lambda' \Vdash \Omega'$ *or*
2. $G; H, \Omega, \Lambda \to \mathsf{alarm}(a)$ *where* $a \in H$ *and* $\Lambda_{|\mathsf{fn}(P)} \not\vdash \mathsf{proc}(a_i, P) :: (a : \Lambda(a))$.

Proof. We prove the lemma by examining each monitoring rule, inverting the typing configuration and applying the typing rules. The proof cases can be found in Appendix A.6. □

While the blame assigment is completely different for the *Verified-Spawn* and *Unverified Spawn* systems, the proofs cases for Lemma 6 and Lemma 2 are remarkably similar. This is the case because the differences in blame assignment (namely the addition of the graph G to the system) are encapsulated in a few key monitoring rules and two different typing rules for configurations.

3.5 Related Work

Compared to the body of work mentioned in Section 2.2, our work focuses on systems where processes communicate with each other via message-passing. At a high-level, we can relate our adversary model to the work on blame assignment as follows. Each process can be viewed as a program written in dynamically typed language. Our monitor enforces the coercion of session types by observing the communications between the processes. Our blame assignment always

includes the compromised process. If we view the compromised process as a less-precisely-typed program, our correctness of blame property is similar to the notion proposed in Wadler and Findler [39]: blame always falls on less-precisely-typed programs.

The work most closely related to ours is on multi-party session types. Bocchi et al. [5] and Chen et al. [9] assume a similar asynchronous message passing model as ours. Their monitor architecture is also similar to ours; monitors are placed at the ends of the communication channels and monitor communication patterns. One key difference is that their monitors do not raise alarms; instead, the monitors suppress bad messages and move on. Our monitors halt the execution and assign blame. Consequently, this work does not have theorems about blame assignment which are central to our work. Using global types, their monitors can additionally enforce global properties such as deadlock freedom, which our monitors cannot. Our work supports higher-order processes, that is, processes that can spawn other processes and delegate communication to other processes, while their work does not.

The recently-developed Whip system [41] addresses a similar problem our work, but does not use session types. They use a dependent type system to implement a contract monitoring system that can connect services written in different languages. Their system is also higher order, and allows processes that are monitored by Whip to interact with unmonitored processes.

$(c_i)^+ = c_{i+1}$

id	:	$\mathsf{proc}(a, a \leftarrow b), \mathcal{C}, \boxed{!(a : A)}, \boxed{!(b : A)} \longrightarrow [b/a]\mathcal{C}$
$\mathsf{id_a}$	:	$\mathsf{proc}(a, a \leftarrow b), \mathcal{C}, (\nexists A. \boxed{!(a : A)}, \boxed{!(b : A)}) \longrightarrow \mathsf{alarm}(a)$
cut	:	$\mathsf{proc}(c, x : A \leftarrow P; Q), \boxed{!(\mathsf{typecheck}(P :: x : A))}, \boxed{!(\mathsf{fn}(P) \cap \mathsf{fn}(Q) = \emptyset)}$ $\longrightarrow \mathsf{proc}(c, [a_0/x]Q), \mathsf{proc}(a_0, [a_0/x]P), \boxed{!(a_0 : A)} \quad (a_0\ fresh)$
$\mathsf{cut_a}$	:	$\mathsf{proc}(c, x : A \leftarrow P; Q), (\boxed{!(\mathsf{typecheck}(P :: x \not: A))} \oplus \boxed{!(\mathsf{fn}(P) \cap \mathsf{fn}(Q) \neq \emptyset)}$ $\longrightarrow \mathsf{alarm}(c)$
lolli_s	:	$\mathsf{proc}(d, \mathsf{send}\ c_i\ a; Q), \boxed{!(c_i : A_1 \multimap A_2)}, \boxed{!(a : A_1)})$ $\longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q), \boxed{!(c_i^+ : A_2)}$
$\mathsf{lolli\_s_a}$	:	$\mathsf{proc}(d, \mathsf{send}\ c_i\ a; Q), (\nexists A_1. \boxed{!(c_i : A_1 \multimap A_2)}, \boxed{!(a : A_1)})$ $\longrightarrow \mathsf{alarm}(d)$
lolli_r	:	$\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i), \mathsf{proc}(c_i, x \leftarrow \mathsf{recv}\ c_i; P)$ $\longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][a/x]P)$
tensor_s	:	$\mathsf{proc}(c_i, \mathsf{send}\ c_i\ a; P), \boxed{!(c_i : A_1 \otimes A_2)}, \boxed{!(a : A_1)}$ $\longrightarrow \mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+), \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \boxed{!(c_i^+ : A_2)}$
$\mathsf{tensor\_s_a}$	:	$\mathsf{proc}(c_i, \mathsf{send}\ c_i\ a; P), (\nexists A_1. \boxed{!(c_i : A_1 \otimes A_2)}, \boxed{!(a : A_1)})$ $\longrightarrow \mathsf{alarm}(c_i)$
tensor_r	:	$\mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c_i; Q)$ $\longrightarrow \mathsf{proc}(d, [c_i^+/c_i][a/x]Q)$
one_s	:	$\mathsf{proc}(a, \mathsf{close}\ a), \boxed{!(a : 1)} \longrightarrow \mathsf{msg}(a, \mathsf{close}\ a)$
$\mathsf{one\_s_a}$	:	$\mathsf{proc}(a, \mathsf{close}\ a), \boxed{!(a \not: 1)} \longrightarrow \mathsf{alarm}(a)$
one_r	:	$\mathsf{msg}(a, \mathsf{close}\ a), \mathsf{proc}(d, \mathsf{wait}\ a; Q) \longrightarrow \mathsf{proc}(d, Q)$
$\mathsf{alarm_r}$	:	$\mathsf{proc}(c, m \leftarrow \mathsf{recv}\ a; R), \boxed{!(a : A)}, \boxed{!(m \not: A)} \longrightarrow \mathsf{alarm}(c)$
$\mathsf{alarm_{r'}}$	:	$\mathsf{proc}(c, m \leftarrow \mathsf{recv}\ c; R), \boxed{!(c : A)}, \boxed{!(m \not: A)} \longrightarrow \mathsf{alarm}(c)$
havoc	:	$\mathsf{proc}(c, P), \boxed{!\mathsf{fn}(P) \supseteq \mathsf{fn}(Q)} \longrightarrow \mathsf{proc}(c, Q), \boxed{!(\mathsf{havoc}(c))}$

Figure 3.3: Verified-Spawn Monitor Rules

$$
\begin{array}{lcl}
\mathsf{with\_s} & : & \mathsf{proc}(d, c_i.k; Q),\ \boxed{!(c_i : \&\{\ell : A_\ell\})},\ \boxed{!(k \triangleright \&\{\ell : A_\ell\})} \\
& & \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q), \mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i),\ \boxed{!(c_i^+ : A_k)} \\
\mathsf{with\_s_a} & : & \mathsf{proc}(d, c_i.k; Q), (\ \boxed{!(c_i :\not\& \{\ell : A_\ell\})} \oplus \boxed{!(k \not\triangleright \&\{\ell : A_\ell\})}\ ) \longrightarrow \mathsf{alarm}(d) \\
\mathsf{with\_r} & : & \mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i), \mathsf{proc}(c_i, \mathsf{case}\ c_i\{\ell \Rightarrow P_\ell\}_{\ell \in L}) \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P_k) \\
\mathsf{plus\_s} & : & \mathsf{proc}(c_i, c_i.k; P),\ \boxed{!(c_i : \oplus\{\ell : A_\ell\})},\ \boxed{!(k \triangleright \oplus\{\ell : A_\ell\})} \\
& & \longrightarrow \mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+), \mathsf{proc}(c_i^+, [c_i^+/c_i]P),\ \boxed{!(c_i^+ : A_k)} \\
\mathsf{plus\_s_a} & : & \mathsf{proc}(c_i, c_i.k; P), (\ \boxed{!(c_i :\not\oplus\{\ell : A_\ell\})} \oplus \boxed{!(k \not\triangleright \oplus\{\ell : A_\ell\})}\ ) \longrightarrow \mathsf{alarm}(c_i) \\
\mathsf{plus\_r} & : & \mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+), \mathsf{proc}(d, \mathsf{case}\ c_i\{\ell \Rightarrow Q_\ell\}_{\ell \in L}) \\
& & \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q_k) \\
\mathsf{down\_s} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ P),\ \boxed{!(c_i :\downarrow A^-)} \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+),\ \boxed{!(c_i^+ : A^-)} \\
\mathsf{down\_s_a} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ P),\ \boxed{!(\not\exists A^-.c_i :\downarrow A^-)} \longrightarrow \mathsf{alarm}(c_i) \\
\mathsf{down\_r} & : & \mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q) \\
& & \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q) \\
\mathsf{up\_s} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q),\ \boxed{!(c_i :\uparrow A^+)} \\
& & \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q),\ \boxed{!(c_i^+ : A^+)} \\
\mathsf{up\_s_a} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q), (\ \boxed{!(\not\exists A^+.c_i :\uparrow A^+)}\ ) \longrightarrow \mathsf{alarm}(d) \\
\mathsf{up\_r} & : & \mathsf{proc}(c_i, \mathsf{shift} \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i) \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P) \\
\mathsf{exists\_s} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P),\ \boxed{!(c_i : \exists n : \tau.A)},\ \boxed{!(v : \tau)} \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+),\ \boxed{!(c_i^+ : [v/n]A)} \\
\mathsf{exists\_s_a} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P), (\not\exists \tau.\ \boxed{!(c_i : \exists n : \tau.A)},\ \boxed{!(v : \tau)}\ ) \longrightarrow \mathsf{alarm}(c_i) \\
\mathsf{exists\_r} & : & \mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \\
& & \longrightarrow \mathsf{proc}(d, [c_i^+/c_i][v/n]Q) \\
\mathsf{forall\_s} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q),\ \boxed{!(c_i : \forall n : \tau.A)},\ \boxed{!(v : \tau)} \\
& & \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q),\ \boxed{!(c_i^+ : [v/n]A)} \\
\mathsf{forall\_s_a} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q), (\not\exists \tau.\ \boxed{!(c_i : \forall n : \tau.A)},\ \boxed{!(v : \tau)}\ ) \longrightarrow \mathsf{alarm}(d) \\
\mathsf{forall\_r} & : & \mathsf{proc}(c_i, n \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i) \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P)
\end{array}
$$

Figure 3.3: Verified-Spawn Monitor Rules (continued)

$\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s}$: $\mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q),\ !(c_U :\uparrow^U_L A^+_L)\ (c_L\ fresh)$
$\longrightarrow \mathsf{proc}(a, [c_L/x_L]Q), \mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L),\ !(c_L : A^+_L)$

$\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s_a}$: $\mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q),\ !(c_U :\not\uparrow^U_L A^+_L) \longrightarrow \mathsf{alarm}(a)$

$\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{r}$: $\mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L), !\mathsf{proc}(c_U, \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P)$
$\longrightarrow \mathsf{proc}(c_L, [c_L/x_L]P)$

$\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s}$: $\mathsf{proc}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q),\ !(c_L :\downarrow^U_L A^-_U)\ (c_U\ fresh)$
$\longrightarrow !\mathsf{proc}(c_U, [c_U/x_U]Q), \mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U),\ !(c_U : A^-_U)$

$\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s_a}$: $\mathsf{proc}(c_L; \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q),\ !(c_L :\not\downarrow^U_L A^-_U) \longrightarrow \mathsf{alarm}(c_L)$

$\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{r}$: $\mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U), \mathsf{proc}(a, \mathsf{shift}\ x_U \leftarrow \mathsf{recv}\ c_L; P)$
$\longrightarrow \mathsf{proc}(a, [c_U/x_U]P)$

id_{U} : $\mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L), !\mathsf{proc}(a_U, a_U \leftarrow b_U),$
$!(a_U : A)\ , !(b_U : A)$
$\longrightarrow \mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ b_U; a_L \leftarrow x_L)$

$\mathsf{id}_{\mathsf{Ua}}$: $\mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L), !\mathsf{proc}(a_U, a_U \leftarrow b_U),$
$\nexists A.(\ !(a_U : A)\ , !(b_U : A)\ )$
$\longrightarrow \mathsf{alarm}(a_U)$

$\mathsf{cut}_{\mathsf{U}}$: $\mathsf{proc}(c, x_U : A \leftarrow P; Q),\ !(\mathsf{typecheck}(P :: x_U : A))$
$\longrightarrow \mathsf{proc}(c, [a_U/x_U]Q), !\mathsf{proc}(a_U, [a_U/x_U]P) \quad (a_U\ fresh)$

$\mathsf{cut}_{\mathsf{Ua}}$: $\mathsf{proc}(c, x_U : A \leftarrow P; Q),\ !(\mathsf{typecheck}(P :: x_U \not: A)) \longrightarrow \mathsf{alarm}(c)$

Figure 3.4: Verified-Spawn Shared Monitor Rules

cut : $\mathsf{proc}(c, x : A \leftarrow P; Q)$
$\longrightarrow \mathsf{proc}(c, [a_0/x]Q), \mathsf{proc}(a_0, [a_0/x]P),\ !(G(c \rightarrow_{sp} a_0)) \quad (a_0\ fresh)$

$\mathsf{up}^{\mathsf{U}}_{\mathsf{k}}\_\mathsf{r}$: $\mathsf{msg}(c_k, \mathsf{shift}\ x_k \leftarrow \mathsf{send}\ c_U; c_k \leftarrow x_k), !\mathsf{proc}(c_U, \mathsf{shift}\ x_k \leftarrow \mathsf{recv}\ c_U; P)$
$\longrightarrow \mathsf{proc}(c_k, [c_k/x_k]P),\ !(G(c_U \rightarrow_{sp} c_k))$

$\mathsf{down}^{\mathsf{U}}_{\mathsf{k}}\_\mathsf{s}$: $\mathsf{proc}(c_k, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_k; Q),\ !(c_k :\downarrow^U_k A^-) \quad (c_U\ fresh)$
$\longrightarrow !\mathsf{proc}(c_U, [c_U/x_U]Q), \mathsf{msg}(c_k, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_k; c_U \leftarrow x_U),$
$!(G(c_k \rightarrow_{sp} c_U))$

$\mathsf{cut}_{\mathsf{U}}$: $\mathsf{proc}(c, x_U : A \leftarrow P; Q) \quad (a_U\ fresh)$
$\longrightarrow \mathsf{proc}(c, [a_U/x_U]Q), !\mathsf{proc}(a_U, [a_U/x_U]P),\ !(G(c \rightarrow_{sp} a_U))$

Figure 3.5: Unverified-Spawn Monitor Rules Highlights (linear and shared)

Chapter 4

Partial Identity Processes as Contracts

The previous chapter of this thesis presented contracts, specified as session types, that enforced communication protocols between processes. In that setting, we assigned each channel a monitor to detect whether messages observed along the channel adhere to the prescribed session type. The monitor identified any deviant behavior exhibited by the communicating processes and triggered alarms. However, contracts based solely on session types are inherently limited in their expressive power. Many contracts that we would like to enforce cannot even be stated using session types alone. As a simple example, consider a "factorization service" which may be sent a (possibly large) integer x and is supposed to respond with a list of prime factors. Session types can only express that the request is an integer and the response is a list of integers, which is insufficient.

By generalizing the class of monitors beyond those derived from session types, we can enforce, for example, that multiplying the numbers in the response yields the original integer x. To handle these contracts, we have designed a model where our monitors execute as transparent processes alongside the computation. They are able to maintain internal state which allows us to check complex properties. These monitoring processes act as partial identities, which do not affect the computation, except possibly raising an alarm, and merely observe the messages flowing through the system. They then perform whatever computation is needed, for example, they can compute the product of the factors, to determine whether the messages are consistent with the contract. If the message violates the contract, they stop the computation. In this chapter, we present a method for checking that monitors are truly partial identities, prove this method correct and explore examples illustrating the breadth of contracts that our monitors can enforce.

4.1 Model

In this section, we explore and define what it means for a process to be a partial identity. This definition, which provides structure to our monitoring processes, is the cornerstone of our model.

As a first simple example, let's take a process that receives one positive integer n and factors it into two integers p and q that are sent back where $p \leq q$. The part of the specification that is *not* enforced is that if n is not prime, p and q should be proper factors, but we at least enforce that all numbers are positive and $n = p * q$. Since a minimal number of shifts can be inferred during

elaboration of the syntax [29], we suppress them in the examples presented in this chapter.

$$
\begin{array}{l}
\mathsf{factor\_t} = \forall n{:}\mathsf{int}.\, \exists p{:}\mathsf{int}.\, \exists q{:}\mathsf{int}.\, \mathbf{1} \\
\mathsf{factor\_monitor} : \{\mathsf{factor\_t} \leftarrow \mathsf{factor\_t}\} \\
c \leftarrow \mathsf{factor\_monitor} \leftarrow d = \\
\quad n \leftarrow \mathsf{recv}\ c\ ;\ \mathsf{assert}\ l_1\ (n > 0)\ ;\ \mathsf{send}\ d\ n\ ; \\
\quad p \leftarrow \mathsf{recv}\ d\ ;\ \mathsf{assert}\ l_2\ (p > 0)\ ;\ q \leftarrow \mathsf{recv}\ d\ ;\ \mathsf{assert}\ l_3\ (q > 0)\ ;\ \mathsf{assert}\ l_4\ (p \leq q)\ ; \\
\quad \mathsf{assert}\ l_5\ (n = p * q)\ ;\ \mathsf{send}\ c\ p\ ;\ \mathsf{send}\ c\ q\ ;\ c \leftarrow d
\end{array}
$$

This is a one-time interaction (the session type factor_t is not recursive), so the monitor terminates. It terminates here by forwarding, but we could equally well have replaced it by its identity-expanded version at type $\mathbf{1}$, which is $\mathsf{wait}\ d\ ;\ \mathsf{close}\ c$.

The contract could be invoked by the provider or by the client. Let's consider how a provider factor might invoke it. In this case, factor_raw is a process that provides a factoring service on channel c'. We assume that channel c' will have the type factor_t. The channel c' is then passed to the factor_monitor process which yields a monitored factoring service on channel c. The provider factor is now providing a monitored factoring service over the channel c.

$$
\begin{array}{l}
\mathsf{factor} : \{\mathsf{factor\_t}\} \\
c \leftarrow \mathsf{factor} = \\
\quad c' \leftarrow \mathsf{factor\_raw}\ ;\ c \leftarrow \mathsf{factor\_monitor} \leftarrow c'
\end{array}
$$

Buffering Values To check that factor_monitor is a partial identity we need to track that p and q are received from the provider, in this order. In general, for any received message, we need to enter it into a message queue q and we need to check that the messages are passed on in the correct order. We use the context Ψ to track values. As a first cut (to be generalized several times), we write for negative types:

$$[q](b : B^-)\ ;\ \Psi \vdash P :: (a : A^-)$$

which expresses that the two endpoints of the monitor are $a : A^-$ and $b : B^-$ (both negative), and we have already received the messages in queue q along channel a.

A monitor, at the top level, is defined with

$$
\begin{array}{l}
\mathit{mon} : \tau_1 \to \ldots \to \tau_n \to \{A \leftarrow A\} \\
a \leftarrow \mathit{mon}\ x_1 \ldots x_n \leftarrow b = P
\end{array}
$$

where the x_i are values. The body of the process P here is type-checked as one of (depending on the polarity of A)

$$[\,](b : A^-)\ ;\ \Psi \vdash P :: (a : A^-) \quad \text{or} \quad (b : A^+)\ ;\ \Psi \vdash P :: [\,](a : A^+)$$

where $\Psi = (x_1{:}\tau_1) \cdots (x_n{:}\tau_n)$.

In general, queues have the form $q = m_1 \cdots m_n$ with

m	::=	l_k	labels	$\oplus, \&$				
	$\mid$	c	channels	$\otimes, \multimap$	$\mid$	n	values	$\exists, \forall$
	$\mid$	end	close	$\mathbf{1}$	$\mid$	shift	shifts	$\uparrow, \downarrow$

where m_1 is the front of the queue and m_n the back.

When the process P receives a message, we add it to the back of the queue q. We also need to add it to the context Ψ to remember its type. In the factoring example $\tau = \mathsf{int}$.

$$\frac{[q \cdot n](b : B)\ ;\ \Psi, n{:}\tau \vdash P :: (a : A^-)}{[q](b : B)\ ;\ \Psi \vdash n \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : \forall n{:}\tau.\, A^-)}\ \forall R$$

Conversely, when we *send* along channel b the message must be equal to the one at the front of the queue (and therefore it must be a variable). The value m remains in the context so it can be reused for later assertion checks. However, it can never be sent again since it has been removed from the queue.

$$\frac{[q](b : [m/n]B)\ ;\ \Psi, m{:}\tau \vdash Q :: (a : A^-)}{[m \cdot q](b : \forall n{:}\tau.\, B)\ ;\ \Psi, m{:}\tau \vdash \mathsf{send}\ b\ m\ ;\ Q :: (a : A)^-}\ \forall L$$

All the other send and receive rules for negative types ($\forall$, $\multimap$, $\&$) follow exactly the same pattern. For positive types, a queue must be associated with the channel along which the monitor provides (the succedent of the sequent judgment).

$$(b : B^+)\ ;\ \Psi \vdash Q :: [q](a : A^+)$$

Moreover, when end has been received along b the corresponding process has terminated and the channel is closed, so we generalize the judgment to

$$\omega\ ;\ \Psi \vdash Q :: [q](a : A^+) \qquad \text{with } \omega = \cdot \mid (b : B).$$

The shift messages change the direction of communication. They therefore need to switch between the two judgments and also ensure that the queue has been emptied before we switch direction. The two rules for $\uparrow$, which appears in our factoring example, are provided below.

$$\frac{[q \cdot \mathsf{shift}](b : B^-)\ ;\ \Psi \vdash P :: (a : A^+)}{[q](b : B^-)\ ;\ \Psi \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : \uparrow A^+)}\ \uparrow R$$

We notice that after receiving a shift, the channel a already changes polarity (we now have to send along it), so we generalize the judgment, allowing the succedent to be either positive or negative. And conversely for the other judgment.

$$\begin{array}{l}[q](b : B^-)\ ;\ \Psi \vdash P :: (a : A) \\ \omega\ ;\ \Psi \vdash Q :: [q](a : A^+) \quad \text{where } \omega = \cdot \mid (b : B)\end{array}$$

When we *send* the final shift, we initialize a new empty queue. Because the queue is empty the two sides of the monitor must have the same type.

$$\frac{(b : B^+)\ ;\ \Psi \vdash Q :: [\,](a : B^+)}{[\mathsf{shift}](b : \uparrow B^+)\ ;\ \Psi \vdash \mathsf{send}\ b\ \mathsf{shift}\ ;\ Q :: (a : B^+)}\ \uparrow L$$

The rules for forwarding are also straightforward. Both sides need to have the same type, and the queue must be empty. As a consequence, the immediate forward is always a valid monitor at a given type.

$$\frac{}{(b : A^+)\ ;\ \Psi \vdash a \leftarrow b :: [\,](a : A^+)}\ \mathsf{id}^+ \qquad \frac{}{[\,](b : A^-)\ ;\ \Psi \vdash a \leftarrow b :: (a : A^-)}\ \mathsf{id}^-$$

Rule Summary The current rules allow us to communicate *only along the channels a and b that are being monitored*. If we send channels along channels, however, these channels must be recorded in the typing judgment, but we are not allowed to communicate along them directly. On the other hand, if we spawn internal (local) channels, say, as auxiliary data structures, we should be able to interact with them since such interactions are not externally observable. Our judgment thus requires two additional contexts: Θ for channels internal to the monitor, and Γ for externally visible channels that may be sent along the monitored channels. Our full judgments therefore are

$$\begin{array}{l}[q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: (a : A) \\ \omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash Q :: [q](a : A^+) \quad \text{where } \omega = \cdot \mid (b : B)\end{array}$$

So far, it is given by the following rules

$$\frac{(\forall \ell \in L) \quad (b : B_\ell)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash Q_\ell :: [q \cdot \ell](a : A^+)}{(b : \oplus\{\ell : B_\ell\}_{\ell \in L})\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{case}\ b\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: [q](a : A^+)}\ \oplus L$$

$$\frac{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: [q](a : B_k) \quad (k \in L)}{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash a.k\ ;\ P :: [k \cdot q](a : \oplus\{\ell : B_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{(\forall \ell \in L) \quad [q \cdot \ell](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P_\ell :: (a : A_\ell)}{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{case}\ a\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (a : \&\{\ell : A_\ell\}_{\ell \in L})}\ \& R$$

$$\frac{[q](b : B_k)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: (a : A) \quad (k \in L)}{[k \cdot q](b : \oplus\{\ell : B_\ell\}_{\ell \in L})\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash b.k\ ;\ P :: (a : A)}\ \& L$$

$$\frac{(b : B)\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Theta \vdash Q :: [q \cdot x](a : A)}{(b : C \otimes B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash x \leftarrow \mathsf{recv}\ b\ ;\ Q :: [q](a : A)}\ \otimes L$$

$$\frac{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: [q](a : A)}{\omega\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Theta \vdash \mathsf{send}\ a\ x\ ;\ P :: [x \cdot q](a : C \otimes A)}\ \otimes R^*$$

$$\frac{[q \cdot x](b : B)\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Theta \vdash P :: (a : A)}{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash x \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : C \multimap A)}\ \multimap R$$

$$\frac{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash Q :: (a : A)}{[x \cdot q](b : C \multimap B)\ ;\ \Psi\ ;\ \Gamma, x{:}C\ ;\ \Theta \vdash \mathsf{send}\ b\ x\ ;\ Q :: (a : A)}\ \multimap L^*$$

$$\frac{\cdot\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash Q :: [q \cdot \mathsf{end}](a : A)}{(b : \mathbf{1})\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{wait}\ b\ ;\ Q :: [q](a : A)}\ \mathbf{1}L$$

$$\frac{}{\cdot\ ;\ \Psi\ ;\ \cdot\ ;\ \cdot \vdash \mathsf{close}\ a :: [\mathsf{end}](a : \mathbf{1})}\ \mathbf{1}R$$

$$\frac{(b : B)\ ;\ \Psi, n{:}\tau\ ;\ \Gamma\ ;\ \Theta \vdash Q :: [q \cdot n](a : A)}{(b : \exists n{:}\tau.\, B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash n \leftarrow \mathsf{recv}\ b\ ;\ Q :: [q](a : A)}\ \exists L$$

$$\frac{\omega\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Theta \vdash P :: [q](a : [m/n]A)}{\omega\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{send}\ a\ m\ ;\ P :: [m \cdot q](a : \exists n{:}\tau.\, A)}\ \exists R$$

$$\frac{[q \cdot n](b : B)\ ;\ \Psi, n{:}\tau\ ;\ \Gamma\ ;\ \Theta \vdash P :: (a : A^-)}{[q](b : B)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash n \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : \forall n{:}\tau.\, A^-)}\ \forall R$$

$$\frac{[q](b : [m/n]B)\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Theta \vdash P :: (a : A)}{[m \cdot q](b : \forall n{:}\tau.\, B)\ ;\ \Psi, m{:}\tau\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{send}\ b\ m\ ;\ Q :: (a : A)}\ \forall L$$

$$\frac{(b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash Q :: [q \cdot \mathsf{shift}](a : A^+)}{(b : {\downarrow}B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ b\ ;\ Q :: [q](a : A^+)}\ {\downarrow}L$$

$$\frac{[\,](b : A^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: (a : A^-)}{(b : A^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{send}\ a\ \mathsf{shift}\ ;\ P :: [\mathsf{shift}](a : {\downarrow}A^-)}\ {\downarrow}R$$

$$\frac{[q \cdot \mathsf{shift}](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: (a : A^+)}{[q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ a\ ;\ P :: (a : {\uparrow}A^+)}\ {\uparrow}R$$

$$\frac{(b : B^+)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash Q :: [\,](a : A^+)}{[\mathsf{shift}](b : {\uparrow}B^+)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash \mathsf{send}\ b\ \mathsf{shift}\ ;\ Q :: (a : A^+)}\ {\uparrow}L$$

$$\frac{}{(b : A^+)\ ;\ \Psi \vdash a \leftarrow b :: [\,](a : A^+)}\ \mathsf{id}^+ \qquad \frac{}{[\,](b : A^-)\ ;\ \Psi \vdash a \leftarrow b :: (a : A^-)}\ \mathsf{id}^-$$

Spawning New Processes The most complex part of checking that a process is a valid monitor involves spawning new processes. As an example, consider the following implementation of a binary tree:

$$\mathsf{tree} = \exists x : \mathsf{int}.\mathsf{tree} \otimes \mathsf{tree} \otimes \mathbf{1}$$

In this implementation, an interaction with the tree involves being sent an integer value, a channel representing the left subtree and a channel representing the right subtree. When defining a monitor to verify a property of the tree, monitors to handle the left and right subtrees would need to be spawned.

In order to be able to spawn and use local (private) processes, we have introduced the (so far unused) context Θ that tracks such channels. We use it here only in the following two rules:

$$\frac{\Psi\ ;\ \Theta \vdash P :: (c : C) \quad \omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta', c{:}C \vdash Q :: [q](a : A^+)}{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta, \Theta' \vdash (c : C) \leftarrow P\ ;\ Q :: [q](a : A^+)}\ \mathsf{cut}_1^+$$

$$\frac{\Psi\ ;\ \Theta \vdash P :: (c : C) \quad [q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta', c{:}C \vdash Q :: (a : A)}{[q](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta, \Theta' \vdash (c : C) \leftarrow P\ ;\ Q :: (a : A)}\ \mathsf{cut}_1^-$$

The second premise (that is, the continuation of the monitor) remains the monitor, while the first premise corresponds to a freshly spawned local process accessible through channel c. All the ordinary left rules for sending or receiving along channels in Θ are also available for the two monitor validity judgments. By the strong ownership discipline of intuitionistic session types, none of this information can flow out of the monitor.

It is also possible for a single monitor to decompose into two monitors that operate concurrently, in sequence. In that case, the queue q may be split anywhere, as long as the intermediate type has the right polarity. Note that Γ must be chosen to contain all channels in q_2, while Γ' must contain all channels in q_1.

$$\frac{\omega\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: [q_2](c : C^+) \quad (c : C^+)\ ;\ \Psi\ ;\ \Gamma'\ ;\ \Theta' \vdash Q :: [q_1](a : A^+)}{\omega\ ;\ \Psi\ ;\ \Gamma, \Gamma'\ ;\ \Theta, \Theta' \vdash c : C^+ \leftarrow P\ ;\ Q :: [q_1 \cdot q_2](a : A^+)}\ \mathsf{cut}_2^+$$

Why is this correct? The first messages sent along a will be the messages in q_1. If we receive messages along c in the meantime, they will be first the messages in q_2 (since P is a monitor), followed by any messages that P may have received along b if $\omega = (b : B)$. The second rule is entirely symmetric, with the flow of messages in the opposite direction.

$$\frac{[q_1](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta \vdash P :: (c : C^-) \quad [q_2](c : C^-)\ ;\ \Psi'\ ;\ \Gamma'\ ;\ \Theta' \vdash Q :: (a : A)}{[q_1 \cdot q_2](b : B^-)\ ;\ \Psi\ ;\ \Gamma, \Gamma'\ ;\ \Theta, \Theta' \vdash c : C^- \leftarrow P\ ;\ Q :: (a : A)}\ \mathsf{cut}_2^-$$

The next two rules allow a monitor to be attached to a channel x that is passed between a and b. The monitored version of x is called x', where x' is chosen fresh. This apparently violates our property that we pass on all messages exactly as received, because here we pass on a monitored version of the original. However, if monitors are partial identities, then the original x and the new x' are indistinguishable (unless a necessary alarm is raised), which will be a tricky part of the correctness proof.

$$\frac{\begin{array}{c}(x : C^+)\ ;\ \Psi\ ;\ \cdot\ ;\ \Theta \vdash P :: [\,](x' : C^+) \\ \omega\ ;\ \Psi\ ;\ \Gamma, x'{:}C^+\ ;\ \Theta' \vdash Q :: [q_1 \cdot x' \cdot q_2](a : A^+)\end{array}}{\omega\ ;\ \Psi\ ;\ \Gamma, x{:}C^+\ ;\ \Theta, \Theta' \vdash x' \leftarrow P\ ;\ Q :: [q_1 \cdot x \cdot q_2](a : A^+)}\ \mathsf{cut}_3^{++}$$

$$\frac{\begin{array}{c}[\,](x : C^-)\ ;\ \Psi\ ;\ \cdot\ ;\ \Theta \vdash P :: (x' : C^-) \\ [q_1 \cdot x' \cdot q_2](b : B^-)\ ;\ \Psi\ ;\ \Gamma, x'{:}C^-\ ;\ \Theta' \vdash Q :: (a : A)\end{array}}{[q_1 \cdot x \cdot q_2](b : B^-)\ ;\ \Psi\ ;\ \Gamma\ ;\ \Theta, \Theta' \vdash x' \leftarrow P\ ;\ Q :: (a : A)}\ \mathsf{cut}_3^{--}$$

There are two more versions of these rules, depending on whether the types of x and the monitored types are positive or negative. These rules play a critical role in monitoring higher-order processes, because monitoring $c : A^+ \multimap B^-$ may require us to monitor the continuation $c : B^-$ (already covered) but also communication along the channel $x : A^+$ received along c.

In actual programs, we mostly use cut $x \leftarrow P\ ;\ Q$ in the form $x \leftarrow p\ \overline{e} \leftarrow \overline{d}\ ;\ Q$ where p is a defined process. The rules are completely analogous, except that for those rules that require splitting a context in the conclusion, the arguments $\overline{d}$ will provide the split for us. When a new sub-monitor is invoked in this way, we remember and eventually check that the process p must

also be a partial identity process, unless we are already checking it. This has the effect that recursively defined monitors with proper recursive calls are in fact allowed. This is important, because monitors for recursive types usually have a recursive structure. An illustration of this can be seen in pos_mon in Figure 4.1.

4.2 Examples

In this section, we present a variety of monitoring processes that can enforce various contracts. Our examples will mainly be concerned with lists where a list of integers is defined as

$$\mathsf{list} = \{\mathsf{cons} : \exists x : \mathsf{int}.\mathsf{list}; \mathsf{nil} : \mathbf{1}\}$$

Any monitor that enforces a contract on a list must peel off each layer of the type one step at a time (by sending or receiving over the channel as dictated by the type), perform the required checks on values or labels, and then reconstruct the original type (again, by sending or receiving as appropriate). All the examples described in this section can be verified to be partial identities based on the definition presented in the previous section.

Refinement The simplest kind of monitoring process we can write is one that models a refinement of an integer type (shown in Figure 4.1); for example, a process that checks whether every element in the list is positive. This is a recursive process that receives the head of the list from channel b, checks whether it is positive (if yes, it continues to the next value, if not it aborts), and then sends the value along to reconstruct the monitored list over channel a.

$\mathsf{pos\_mon} : \{\mathsf{list} \leftarrow \mathsf{list}\}$
$a \leftarrow \mathsf{pos\_mon} \leftarrow b =$
  $\mathsf{case}\ b\ \mathsf{of}$
  $\mid\ \mathsf{nil} \Rightarrow a.\mathsf{nil}\ ;\ \mathsf{wait}\ b\ ;\ \mathsf{close}\ a$
  $\mid\ \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{assert}\ l\ (x > 0)\ ;\ a.\mathsf{cons}\ ;\ \mathsf{send}\ a\ x\ ;\ a \leftarrow \mathsf{pos\_mon} \leftarrow b$

Figure 4.1: Positive Integer List Monitor

$\mathsf{empty\_mon} : \{\mathsf{list} \leftarrow \mathsf{list}\}$
$a \leftarrow \mathsf{empty\_mon} \leftarrow b =$
  $\mathsf{case}\ b\ \mathsf{of}$
  $\mid\ \mathsf{nil} \Rightarrow \mathsf{wait}\ b\ ;\ a.\mathsf{nil}\ ;\ \mathsf{close}\ a$
  $\mid\ \mathsf{cons} \Rightarrow \mathsf{abort}\ l$

$\mathsf{nonempty\_mon} : \{\mathsf{list} \leftarrow \mathsf{list}\}$
$a \leftarrow \mathsf{nonempty\_mon} \leftarrow b =$
  $\mathsf{case}\ b\ \mathsf{of}$
  $\mid\ \mathsf{nil} \Rightarrow \mathsf{abort}\ l$
  $\mid\ \mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ b\ ;\ a.\mathsf{cons}\ ;\ \mathsf{send}\ a\ x\ ;\ a \leftarrow b$

Figure 4.2: Nonempty and Empty List Monitors

Our monitors can also exploit information contained in the labels present in the external and internal choices. We show two examples of monitors that model label refinement in Figure 4.2. The empty_mon process checks whether the list offered over channel b is empty and aborts if channel b sends the label cons. Similarly, the nonempty_mon monitor checks whether the list

offered over channel b is not empty and aborts if channel b sends the label nil. These two monitors enforce refinements: $\{\mathsf{nil}\} \subseteq \{\mathsf{nil}, \mathsf{cons}\}$ and $\{\mathsf{cons}\} \subseteq \{\mathsf{nil}, \mathsf{cons}\}$.

Monitors with internal state We now move beyond refinement contracts, and model contracts that have to maintain some internal state. We first present a monitor that checks whether a set of right and left parentheses match (shown in Figure 4.3). The match_mon monitor uses its internal state to push every left parenthesis it sees on its stack and to pop it off when it sees a right parenthesis. For brevity, we model our list of parentheses by marking every left parenthesis with a 1 and right parenthesis with a -1. So the sequence ()()) would look like $1, -1, 1, -1, -1$. As we can see, this is not a proper sequence of parentheses because adding all of the integer representations does not yield 0. In a similar vein, we can implement a process that checks that a tree is serialized correctly, which is related to recent work on context-free session types by Thiemann and Vasconcelos [33].

We can also write a monitor that checks whether a given list is sorted in ascending order (shown in Figure 4.4). The ascending_mon monitor uses its internal state to enforce a lower bound on subsequent elements of the list. In order to represent this bound, we add an option int type to our language. This value can either be None if no bound has yet been set, or Some b if b is the current bound.

```
match_mon : int → {list ← list}
a ← match_mon count ← b =
  case b of
  | nil ⇒ assert l1(count = 0) ; a.nil ; wait b ; close a
  | cons ⇒ x ← recv b ; a.cons ;
    if (x = 1) then send a x ; a ← match_mon (count + 1) ← b
    else if (x = −1) then assert l2 (count > 0) ; send a x ; a ← match_mon (count − 1) ← b
    else abort l3   //invalid input
```

Figure 4.3: Parenthesis Matching Monitor

```
ascending_mon : option int → {list ← list}
m ← ascending_mon bound ← n =
  case n of
  | nil ⇒ m.nil ; wait n ; close m
  | cons ⇒ x ← recv n ;
          case bound of
        | None ⇒ m.cons ; send m x ; m ← ascending_mon (Some x) ← n
        | Some a ⇒ assert l (x ≥ a) ; m.cons ; send m x ;
          m ← ascending_mon (Some x) ← n
```

Figure 4.4: Ascending Monitor

If the list is empty, there is no bound to check, so no contract failure can happen. If the list is nonempty, we check to see if a bound has already been set. If not, we set the bound to be the

first received element. If there is already a bound in place, then we check if the received element is greater or equal to the bound. If it is not, then the list must be unsorted, so we abort with a contract failure. We note that the input list n remains unchanged because every element that we examine we pass along unchanged to m.

To take the above example one step further, assume we have an actual sorting procedure that takes a stream of integers as input and produces a sorted list. We can use the ascending_mon process to verify that the elements of the output list are in sorted order. However, we still need to verify that the elements in the output list are in fact a permutation of the elements that were sent to the sorting procedure. Given a reasonable hash function h, we can write monitors to accomplish this goal (shown in Figure 4.5). We first introduce the sorter type which is an external choice with two options – next which receives an integer and continues behaving as a sorter, or done which completes the sorting and returns a list. We can write a monitor sorter_mon that hashes each element as it is sent to the sorting procedure and keeps track of a running total of the sum of the hashes. Once the sorting procedure has generated the resulting sorted list, we can use the result_mon to compute the hash of each element and subtract it from the total. After all of the elements are received, the monitor checks that the total is 0 – if it is, with high probability, the input stream and output list are permutations of each other. This example is an instance of *result checking* and is inspired by Wasserman and Blum [40].

In order to provide a nonprobabalistic guarantee, we need to carry around an index associated with each integer throughout the computation, so that we can verify that the two lists contain the same indices and are therefore permutations of each other. Currently, our monitoring processes do not maintain enough state to enforce these contracts because the monitoring process must send and receive extra messages, just to verify the contract. We call these messages ghosts because they are only used for contract-checking and cannot affect the actual computation taking place. Integrating ghost messages into monitoring infrastructure poses significant theoretical challenges – it is necessary to prove that ghost messages are transparent and do not influence "real" computation.

```
result_mon : int → {list ← list}
k ← result_mon total ← l =
  case l of
  | nil ⇒ assert l (total = 0) ; k.nil ; wait a ; close k
  | cons ⇒ x ← recv l ; k.cons ; send k x ; k ← result_mon (total − h(x)) ← l

sorter : &{next : ∀n : int.sorter; done : list}
sorter_mon : int → {sorter ← sorter}
a ← sorter_mon total ← b =
  case a of
  | next ⇒ x ← recv a ; b.next ; send b x ; a ← sorter_mon (total + h(x)) ← b ;
  | done ⇒ b ← result_montotal ← a
```

Figure 4.5: Result Checking Monitor

Mapper Finally, we can also define monitors that check higher-order contracts, such as a contract

for a mapping function. Consider the mapper process shown in Figure 4.6 which receives an integer, doubles it, sends it back and recurses. Looking at the code, we can see that any positive integer that the mapper has processed will be strictly larger than the original integer. This contract can be imposed on the mapper itself, which is done in the mapper_mon process. This process first examines each integer the mapper receives and asserts it is positive. If this precondition is met, then the value is sent to the mapper and once the mapped value is received, an assertion confirms that it is larger than the original input. We would like to monitor the mapper as it is applied to the list l in the map process, so we write a wrapper process that starts a monitor on the mapper m and passes the monitored mapper m' to the map process. This higher-order example concludes our example section.

```
mapper_t = &{done : 1 ; next : ∀n : int.∃n : int.mapper_t}
mapper_proc : {mapper_t}
m ← mapper_proc =
  case m of
  | done ⇒ close m
  | next ⇒ x ← recv m ; send m (2 * x) ; m ← mapper_proc

mapper_mon : {mapper_t ← mapper_t}
n ← mapper_mon ← m =
  case n of
  | done ⇒ m.done ; wait m ; close n
  | next ⇒ x ← recv n ;  assert l1 (x > 0)    //checks precondition
    m.next ; send m x ; y ← recv m ;  assert l2 (y > x)    //checks postcondition
    n.next ; send n y ; n ← mapper_mon ← m

map : {list ← mapper_t ; list}
k ← map ← m l =
  case l of
  | nil ⇒ m.done ; k.nil ; wait l ; wait m ; close k
  | cons ⇒ x ← recv l ; m.next ; send m x ; y ← recv m ;
    k.cons ; send k y ; k ← map ←  m l

wrapper : {list ← mapper_t ; list}
k ← wrapper ← m l =
  m' ← mapper_mon ← m;    //run monitor
  k ← map m' l
```

Figure 4.6: Higher-Order Monitor

4.3 Metatheory

We prove that the partial-identity criterion presented in Section 4.1 guarantees that session-typed monitoring processes are observationally equivalent to partial identity processes. A simplified

version of the theorem is stated below (a more complete version and proof details can be found in Gommerstadt et al [16]).

The notion of observational equivalence we need does not observe nontermination, that is, it only compares messages that are actually received. Since messages can flow in two directions, we need to observe messages that arrive at either end. We therefore do not require, as is typical for a bisimulation, for the configurations to be in step with each other. In other words, if one configuration takes a step, it is not necessarily the case that the other configuration also took a step. Instead we say if both configurations send an externally visible message, then the messages must be equivalent. For two configurations C_1 and C_2 we write $C_1 \sim C_2$ for our notion of observational equivalence.

Theorem 7 (Transparency)**.** *Let P be a process that adheres to the typing rules presented in Section 4.1. Then, $b : B \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, P) :: (a : A)$.*

Proof Sketch. We prove the theorem by first defining a notion of observational equivalence for messages. We then contruct a partial bisimulation. We note that this bisimulation is not standard because the process P can terminate due to a failed assertion. □

4.4 Related Work

Most closely related to our is the work by Disney et al. [12], which investigates behavioral contracts that enforce temporal properties for modules. Our contracts (i.e., session types) enforce temporal properties as well; the session types specify the order in which messages are sent and received by the processes. Our contracts can also make use of internal state, as those of Disney et al. do, but our system is concurrent, while their system does not consider concurrency.

Recently, Melgratti and Padovani have developed chaperone contracts for higher-order session types [23]. Their work is based on a classic interpretation of session types, instead of an intuitionistic one like ours; therefore, they do not handle spawning or forwarding processes. While their contracts also inspect messages passed between processes, unlike ours, they cannot model contracts which rely on the monitor making use of internal state (e.g., the parenthesis matching). They proved a blame theorem relying on the notion of locally correct modules, which is a semantic categorization of whether a module satisfies the contract. We did not prove a general blame theorem; instead, we prove a somewhat standard safety theorem for cast-based contracts (discussed in Chapter 5).

The Whip system [41] uses a dependent type system to implement a contract monitoring system. However, Whip cannot handle stateful contracts. Another distinguishing feature of our monitors is that they are partial identity processes encoded in the same language as the processes to be monitored.

Chapter 5

Refinement Types as Contracts

In the previous chapter, we introduced the notion of partial-identity monitors. The advantage of this monitoring approach is its generality, which allows us to express a variety of contracts. However, a drawback of this approach is that due to its generality, it is difficult to assign blame when a contract has been breached. In this chapter, we restrict our partial-identity monitors to model a particular category of contracts in order to provide blame assignment for this category.

That is, we show how to check refinement types dynamically using our partial-identity monitors. We encode refinements as type casts, which allows processes to remain well-typed with respect to the non-refinement type system (described in Section 2.1). These casts are translated at run time to monitors that validate whether the cast expresses an appropriate refinement. If so, the monitors behave as identity processes; otherwise, they raise an alarm and abort. We prove that our translation generates monitors that are well-typed and that are valid partial identity processes. For refinement contracts, we can also prove a safety theorem, analogous to the classic "Well-typed Programs Can't be Blamed" [39], stating that if a monitor enforces a contract that casts from type A to type B, where A is a subtype of B, then this monitor will never raise an alarm.

5.1 Model

Surface Language We first augment messages and processes to include casts as follows. We write $\langle A \Leftarrow B \rangle^{\rho}$ to denote a cast from type B to type A, where ρ is a unique label for the cast. The cast for values is written as $(\langle \tau \Leftarrow \tau' \rangle^{\rho})$. In this section, we treat the types τ' and τ as refinement types of the form $\{n{:}t \mid b\}$, where b is a boolean expression that expresses simple properties of the value n. We express any unrefined type τ by writing $\{n : t \mid \mathsf{true}\}$ which is compatible with earlier sections of this thesis.

$$P \quad ::= \quad \cdots \mid x \leftarrow \langle \tau \Leftarrow \tau' \rangle^{\rho}\, v \,;\, Q \mid a \leftarrow \langle A \Leftarrow B \rangle^{\rho}\, b$$

Both of the additional rules to type casts are shown below. We only allow casts between two types that are compatible with each other (written $A \sim B$), which is co-inductively defined based on the structure of the types. The full definition is shown in Figure 5.1.

$$\frac{A \sim B}{\Psi\ ;\ b : B \vdash a \leftarrow \langle A \Leftarrow B \rangle^{\rho}\ b :: (a : A)}\ \mathsf{id\_cast}$$

$$\frac{\Psi \vdash v : \tau' \quad \Psi, x : \tau\ ;\ \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi\ ;\ \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^{\rho}\ v\ ;\ Q :: (c : C)}\ \mathsf{val\_cast}$$

$$\frac{}{\{n{:}t \mid b_1\} \sim \{n{:}t \mid b_2\}}\ \mathsf{base} \qquad \frac{}{1 \sim 1}\ 1$$

$$\frac{A \sim A' \quad B \sim B'}{A \otimes B \sim A' \otimes B'}\ \otimes \qquad \frac{A' \sim A \quad B \sim B'}{A \multimap B \sim A' \multimap B'}\ \multimap$$

$$\frac{A_k \sim A'_k\ \text{for all}\ k \in I \cap J}{\oplus\{\ell_k : A_k\}_{k \in I} \sim \oplus\{\ell_k : A'_k\}_{k \in J}}\ \oplus \qquad \frac{A_k \sim A'_k\ \text{for all}\ k \in I \cap J}{\&\{\ell_k : A_k\}_{k \in I} \sim \&\{\ell_k : A'_k\}_{k \in J}}\ \&$$

$$\frac{A \sim B}{\downarrow A\ \sim \downarrow B}\ \downarrow \qquad \frac{A \sim B}{\uparrow A\ \sim \uparrow B}\ \uparrow$$

$$\frac{A \sim B \quad \tau_1 \sim \tau_2}{\exists n : \tau_1 . A \sim \exists n : \tau_2 . B}\ \exists \qquad \frac{A \sim B \quad \tau_1 \sim \tau_2}{\forall n : \tau_1 . A \sim \forall n : \tau_2 . B}\ \forall$$

Figure 5.1: Compatibility

Translation to Monitors At runtime, casts are translated into monitoring processes. A cast $a \leftarrow \langle A \Leftarrow B \rangle^{\rho}\ b$ is implemented as a monitor. This monitor ensures that the process that offers a service on channel b behaves according to the prescribed type A. Because of the typing rules, we are assured that channel b must adhere to the type B.

Figure 5.2 is a summary of all the translation rules, except recursive types. The translation is of the form: $[\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} = P$, where A, B are types; the channels a and b are the offering channel and monitoring channel (respectively) for the resulting monitoring process P; and ρ is the label of the monitor (i.e., the contract).

In a functional setting, when a cast fails, either the cast itself is blamed, or the environment is blamed. That is, blame would be assigned to the label ρ if channel B could not be coerced to to type A. Conversely, blame would be assigned to the label $\overline{\rho}$ if the channel b did not have type B. In our setting, our typing rules will ensure that channel b will have type B making the label $\overline{\rho}$ unecessary. Further, unlike in a functional setting where inputs are provided to a function, communication between processes is bi-directional. While blame is always triggered by processes sending messages to the monitor, our contracts may depend on a set of values received from both processes, so it does not make sense to blame one party. Finally, consider the case of forwarding where the processes at either end of the channel are behaving according to the types (contracts) assigned to them, but the cast may connect two processes that have incompatible

types. In this case, it is unfair to blame either one of the processes. Instead, we blame the label of the failed contract.

$$
\begin{array}{lcl}
\mathsf{one} & : & [\![\langle \mathbf{1} \Leftarrow \mathbf{1} \rangle^{\rho}]\!]_{a,b} = \mathsf{wait}\ b; \mathsf{close}\ a \\
\mathsf{lolli} & : & [\![\langle A_1 \multimap A_2 \Leftarrow B_1 \multimap B_2 \rangle^{\rho}]\!]_{a,b} = \\
 & & x \leftarrow \mathsf{recv}\ a\ ;\ y \leftarrow [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x\ ;\ \mathsf{send}\ b\ y\ ;\ [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b} \\
\mathsf{tensor} & : & [\![\langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2 \rangle^{\rho}]\!]_{a,b} = \\
 & & x \leftarrow \mathsf{recv}\ b\ ;\ y \leftarrow [\![\langle A_1 \Leftarrow B_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x\ ;\ \mathsf{send}\ a\ y\ ;\ [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b} \\
\mathsf{forall} & : & [\![\langle \forall\{n : t \mid e\}.\, A \Leftarrow \forall\{n : t' \mid e'\}.\, B \rangle^{\rho}]\!]_{a,b} = \\
 & & n \leftarrow \mathsf{recv}\ a\ ;\ \mathsf{assert}\ \rho\, e'(n)\ \ ;\ \mathsf{send}\ b\ n; [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{exists} & : & [\![\langle \exists\{n : t \mid e\}.\, A \Leftarrow \exists\{n : t' \mid e'\}.\, B \rangle^{\rho}]\!]_{a,b} = \\
 & & n \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{assert}\ \rho\, e(n)\ \ ;\ \mathsf{send}\ a\ n; [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{up} & : & [\![\langle \uparrow A \Leftarrow \uparrow B \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{shift} \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{send}\ a\ \mathsf{shift}\ ;\ [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{down} & : & [\![\langle \downarrow A \Leftarrow \downarrow B \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{shift} \leftarrow \mathsf{recv}\ a\ ;\ \mathsf{send}\ b\ \mathsf{shift}\ ;\ [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b} \\
\mathsf{plus} & : & [\![\langle \oplus\{\ell : A_\ell\}_{\ell \in I} \Leftarrow \oplus\{\ell : B_\ell\}_{\ell \in J} \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{case}\ b\ (\ell \Rightarrow Q_\ell)_{\ell \in J} \\
 & & \mathsf{where}\ \forall \ell, \ell \in I \cap J,\ a.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell \rangle^{\rho}]\!]_{a,b} = Q_\ell \\
 & & \mathsf{and}\ \forall \ell, \ell \in J \wedge \ell \notin I,\ Q_\ell = \mathsf{abort}\ \rho \\
\mathsf{with} & : & [\![\langle \&\{\ell : A_\ell\}_{\ell \in I} \Leftarrow \&\{\ell : B_\ell\}_{\ell \in J} \rangle^{\rho}]\!]_{a,b} = \\
 & & \mathsf{case}\ a\ (\ell \Rightarrow Q_\ell)_{\ell \in I} \\
 & & \mathsf{where}\ \forall \ell, \ell \in I \cap J,\ b.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell \rangle^{\rho}]\!]_{a,b} = Q_\ell \\
 & & \mathsf{and}\ \forall \ell, \ell \in I \wedge \ell \notin J,\ Q_\ell = \mathsf{abort}\ \rho
\end{array}
$$

Figure 5.2: Cast Translation

The translation is defined following the structure of the types. The tensor rule generates a process that first receives a channel (x) from the channel being monitored (b), then spawns a new monitor to monitor x, making sure that it behaves as type A_1. Then, it passes the new monitor's offering channel y to channel a. Finally, the monitor continues to monitor channel b to make sure that it behaves as type A_2. The lolli rule is similar to the tensor rule, except that the monitor first receives a channel from its offering channel. Similar to the higher-order function case, the argument position is contravariant, so the newly spawned monitor checks that the received channel behaves as type B_1. The exists rule generates a process that first receives a value from the channel b, then checks the boolean condition e to validate the contract. The forall rule is similar, except the argument position is contravariant, so the boolean expression e' is checked on the offering channel a. The with rule generates a process that checks that all of the external choices promised by the type $\&\{\ell : A_\ell\}_{\ell \in I}$ are offered by the process being monitored. If a label in the set I is not implemented, then the monitor aborts with the label ρ. The plus rule requires that, for internal choices, the monitor checks that the monitored process only offers

choices within the labels in the set $\oplus\{\ell : A_\ell\}_{\ell \in I}$.

We translate casts with recursive types as follows. For each pair of compatible recursive types A and B, we generate a unique monitor name f and record its type $f : \{A \leftarrow B\}$ in a context Σ. The translation algorithm needs to take additional arguments, including Σ to generate and invoke the appropriate recursive process when needed. For instance, when generating the monitor process for $f : \{\mathsf{list} \leftarrow \mathsf{list}\}$, we follow the rule for translating internal choices. For $[\![\langle \mathsf{list} \Leftarrow \mathsf{list} \rangle^\rho]\!]_{y,x}$ we apply the exists rule in the translation to get $y \leftarrow f \leftarrow x$.

5.2 Examples

We present two examples – one for an integer refinement and one for a label refinement. Consider the below cast:

$$\langle \exists n : \mathsf{int} \mid n > 2.A \Leftarrow \exists n : \mathsf{int} \mid m > 0.B \rangle^\rho$$

where channel a has type $\{\exists n : \mathsf{int} \mid n > 2.A\}$ and channel b has type $\{\exists n : \mathsf{int} \mid m > 0.B\}$. To validate this cast, we first receive the integer value from channel b. We then assert that this value meets channel a's refinement contract by checking if it is larger than two. We then send this value along to channel a and continue checking the rest of the type. The monitor would look like this:

$[[\langle \{\exists n : \mathsf{int} | n > 2.A\} \Leftarrow \{\exists m : \mathsf{int} | m > 0.B\} \rangle^\rho]]_{a,b} =$
$\quad x \leftarrow \mathsf{recv}\ b\ ;$
$\quad \mathsf{assert}\ \rho\ (x > 2)\ ;$
$\quad \mathsf{send}\ a\ x\ ;$
$\quad [[\langle A \Leftarrow B \rangle^\rho]]_{a,b}\ ;$

Figure 5.3: Integer Cast Translation

We can also handle casts with label refinements like the following:

$$\langle \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}\} \Leftarrow \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}; \mathsf{nil} : 1\} \rangle^\rho$$

In this example, we have to case on the possible labels received from channel b. If the $cons$ label is received, we send the $cons$ label along to channel a. Otherwise, we abort. In order for this monitor to adhere to the partial identity criterion defined in the previous chapter, the type of the monitoring process would be $\mathsf{list} \leftarrow \mathsf{list}$.

$[[\langle \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}; \mathsf{nil} : 1\} \Leftarrow \oplus\{\mathsf{cons} : \exists n : \mathsf{int.list}\} \rangle^\rho]]_{a,b} =$
$\quad \mathsf{case}\ b\ \mathsf{of}$
$\quad | \ \mathsf{cons} \Rightarrow a.\mathsf{cons}\ ;\ [[\langle \exists n : \mathsf{int.list} \Leftarrow \exists n : \mathsf{int.list} \rangle^\rho]]_{a,b}$
$\quad | \ \mathsf{nil} \Rightarrow \mathsf{abort}\ \rho$

Figure 5.4: Label Cast Translation

$$\frac{}{\Sigma \vdash 1 \leq 1}\ 1 \qquad \frac{\Sigma \vdash A \leq A' \quad \Sigma \vdash B \leq B'}{\Sigma \vdash A \otimes B \leq A' \otimes B'}\ \otimes \qquad \frac{\Sigma \vdash A' \leq A \quad \Sigma, B \leq B'}{\Sigma \vdash A \multimap B \leq A' \multimap B'}\ \multimap$$

$$\frac{\Sigma \vdash A_k \leq A'_k \text{ for } k \in J \quad J \subseteq I}{\Sigma \vdash \oplus\{\ell_k : A_k\}_{k\in J} \leq \oplus\{\ell_k : A'_k\}_{k\in I}}\ \oplus \qquad \frac{\Sigma \vdash A_k \leq A'_k \text{ for } k \in J \quad I \subseteq J}{\Sigma \vdash \&\{\ell_k : A_k\}_{k\in J} \leq \&\{\ell_k : A'_k\}_{k\in I}}\ \&$$

$$\frac{\Sigma \vdash A \leq B}{\Sigma \vdash \downarrow A\ \leq \downarrow B}\ \downarrow \qquad \frac{\Sigma \vdash A \leq B}{\Sigma \vdash \uparrow A\ \leq \uparrow B}\ \uparrow$$

$$\frac{\Sigma \vdash A \leq B \quad \tau_1 \leq \tau_2}{\Sigma \vdash \exists n : \tau_1.A \leq \exists n : \tau_2.B}\ \exists \qquad \frac{\Sigma \vdash A \leq B \quad \tau_2 \leq \tau_1}{\Sigma \vdash \forall n : \tau_1.A \leq \forall n : \tau_2.B}\ \forall$$

$$\frac{\Sigma, A \leq B \vdash \mathsf{def}(A) \leq \mathsf{def}(B)}{\Sigma \vdash A \leq B}\ \mathsf{def} \qquad \frac{}{\Sigma, A \leq B \vdash A \leq B}\ \mathsf{cycle}$$

$$\frac{\forall v{:}t, [v/x]b_1 \mapsto^* \mathsf{true} \text{ implies } [v/x]b_2 \mapsto^* \mathsf{true}}{\Sigma \vdash \{x{:}t \mid b_1\} \leq \{x{:}t \mid b_2\}}\ \mathsf{refine}$$

Figure 5.5: Subtyping

5.3 Metatheory

We prove two formal properties of cast-based monitors: safety and transparency. We also prove preservation in the presence of well-typed casts.

Because of the expressiveness of our contracts, a general safety (or blame) theorem is difficult to achieve. However, for cast-based contracts, we can prove that a cast which enforces a subtyping relation, and the corresponding monitor, will not raise an alarm.

We first define our subtyping relation in Figure 5.5. In addition to the subtyping between refinement types, we also include label subtyping for our session types. A process that offers more external choices can always be used as a process that offers fewer external choices. Similarly, a process that offers fewer internal choices can always be used as a process that offers more internal choices (e.g., non-empty list can be used as a list). The subtyping rules for internal and external choices are drawn from work by Gay and Hole [15]. For recursive types, we directly examine their definitions. Therefore, the natural formulation of the rules would be coinductive. However, to make our proofs simpler, we transform this coinductive judgement into an inductive judgement. We proceed by adding a context Σ to track the subtyping information that has already been encountered in order to model circular proofs.

Our safety theorem guarantees that well-typed casts do not raise alarms. The key is to show that the monitor process generated from the translation algorithm in Figure 5.2 is well-typed under a typing relation which guarantees that no $\mathsf{abort}(l)$ state can be reached.

We refer to the type system presented thus far in the thesis as T, where monitors that may evaluate to $\mathsf{abort}(l)$ can be typed. We define a stronger type system S which consists of the rules in T with the exception of the abort rule and we replace the assert rule with the assert_strong

rule. This new rule verifies that the condition b is true using the fact that the refinements are stored in the context Ψ. The two type systems are summarized in Figure 5.6.

Theorem 8 (Refinement monitors are well-typed)**.**

1. *$b : B \vdash_T [\![\langle A \Leftarrow B\rangle^\rho]\!]^\Psi_{a,b}$ :: (a : A).*
2. *If $B \leq A$, then $b : B \vdash_S [\![\langle A \Leftarrow B\rangle^\rho]\!]^\Psi_{a,b} :: (a : A)$.*

Proof. The proof is by induction over the monitor translation rules. To prove (1) we use type system T. To prove (2) we use type system T and the sub-typing relation to show that (a) for the internal and external choice cases, no branches that include abort are generated; and (b) for the forall and exists cases, the assert never fails (i.e., the assert_strong rule applies). The proof cases are presented in Appendix B.1. □

As a corollary, we can show that when executing in a well-typed context, a monitor process translated from a well-typed cast will never raise an alarm. To state the corollary, we present the configuration typing. This configuration typing is similar to the one defined in Chapter 3, but without a context to track havoced channels. We assume that the comma operator is associative with $\cdot$ as the unit.

$$\mathcal{C} = \cdot \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, P) \mid \mathcal{C}_1, \mathcal{C}_2$$

$$\frac{}{\cdot \Vdash \cdot} \qquad \frac{\Delta \Vdash \mathcal{C}_1 \quad \Delta \Vdash \mathcal{C}_2}{\Delta \Vdash \mathcal{C}_1, \mathcal{C}_2} \qquad \frac{\Delta_{|\mathsf{fn}(P)} \vdash_S P :: (c : \Delta(c))}{\Delta \Vdash \mathsf{proc}(c, P)} \qquad \frac{\Delta_{|\mathsf{fn}(P)} \vdash_S P :: (c : \Delta(c))}{\Delta \Vdash \mathsf{msg}(c, P)}$$

Corollary 1 (Well-typed casts cannot raise alarms)**.** *$\Delta \Vdash \mathcal{C} :: (b : B)$ and $B \leq A$ implies $\mathcal{C}, \mathsf{proc}(a, [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}) \not\longrightarrow^* \mathsf{abort}(\rho)$.*

Proof. By Theorem 8 we have that $\Psi \; ; \; b : B \vdash_S [\![\langle A \Leftarrow B\rangle^\rho]\!]^\Psi_{a,b} :: (a : A)$. Type system S does not allow aborts, so $\mathcal{C}, \mathsf{proc}(a, [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}) \not\longrightarrow^* \mathsf{abort}(\rho)$. □

Next, we prove that monitors translated from casts are partial identity processes.

Theorem 9 (Casts are transparent)**.**
$b : B \vdash \mathsf{proc}(b, a \leftarrow b) \sim \mathsf{proc}(a, [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}) :: (a : A)$.

Proof. We just need to show that the translated process passes the partial identity checks. We can show this by induction over the translation rules and by applying the rules in Section 4.1. Some representative cases are shown in Appendix B.2. □

Finally, we prove preservation for the system. Due to the assumption that any cast $\langle A \Leftarrow B\rangle^\rho$ will be well-typed (defined as $B \leq A$), we have to define substitution rules (shown in Figure 5.7) and prove a substitution lemma in the presence of subtyping. We define $|E|$ to be the size of the derivation of E.

Lemma 10 (Subtype-Substitution)**.**

1. *If $E = \Psi; \Delta, x : A \vdash P :: (c : C)$ and $B \leq A$ then for any fresh variable $y : B$ we have $E' = \Psi; \Delta, y : B \vdash [y : B/x : A]P :: (c : C)$ and $|E| = |E'|$.*
2. *If $E = \Psi, n : \tau; \Delta \vdash P :: (c : C)$ and $\tau' \leq \tau$ then for any fresh variable $m : \tau'$ we have $E' = \Psi, m : \tau'; \Delta \vdash [m : \tau'/n : \tau]P :: (c : C)$ and $|E| = |E'|$*

3. *If* $E = \Psi; \Delta \vdash P :: (x : A)$ *and* $A \leq B$ *then for any fresh variable* $y : B$ *we have* $E' = \Psi; \Delta \vdash [y : B/x : A]P :: (y : B)$ *and* $|E| = |E'|$.

Proof. We prove the lemma by inducting over the derivation of $\Psi; \Delta \vdash P :: (c : C)$. We note that we implicitly encode any channel $c : A$ or value $v : \tau$ as $\langle A \Leftarrow A \rangle^{\rho} c : A$ and $\langle \tau \Leftarrow \tau \rangle^{\rho} v : \tau$ respectively. Representative cases are provided in Appendix B.4. □

We also augment the semantics presented in (Section 2.1) with types and casts as appropriate. The typed semantics are shown in Figure 5.8.

Theorem 11 (Subtyping-Preservation)**.** *If* $\Delta \Vdash \mathcal{C}$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$ *then* $\Delta \Vdash \mathcal{C}'$.

Proof. We prove the theorem by examining the semantics, and invoking Lemma 10. The proof cases are provided in Appendix B.6. □

Proving preservation for this system was unexpectedly more challenging than proving the safety and transparency theorems presented earlier in the section. The main challenge arose when defining and proving a substitution lemma for a session-type system with subtyping. This preservation proof concludes our metatheory section.

5.4 Related Work

Many of the contracts studied in the context of the lambda calculus [2, 10, 11, 14, 22, 38] are based on refinement types. Our contracts are able to encode refinement-based contracts. Our safety theorem supports Walder and Findler's [39] claim that the less-precisely typed code is always to blame. When we cast from a type to its supertype, the cast can never be at fault, and remains well-typed. A cast can only be at fault when coercing a less-precise type to a more-precise type. In this situation, we generate a monitor to validate the cast.

Recently, gradual typing for two-party session-type systems has been developed [20, 32]. Even though it is a different formalism, the way untyped processes are gradually typed at runtime resembles how we monitor type casts. Because of dynamic session types, their system has to keep track of the linear use of channels, which is not needed for our monitors.

Both System T and S

$$\frac{}{\Psi\ ;\ b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id} \qquad \frac{\Psi\ ;\ \Delta \vdash P :: (x : A) \quad x : A, \Delta' \vdash Q :: (c : C)}{\Psi\ ;\ \Delta, \Delta' \vdash x{:}A \leftarrow P\ ;\ Q :: (c : C)}\ \mathsf{cut}$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^+)}{\Psi\ ;\ \Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \uparrow A^+)}\ \uparrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^+ \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \uparrow A^+ \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q :: (d : D)}\ \uparrow L$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : A^-)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ P :: (c : \downarrow A^-)}\ \downarrow R \qquad \frac{\Psi\ ;\ \Delta, c : A^- \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \downarrow A^- \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \downarrow L$$

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: (c : \mathbf{1})}\ \mathbf{1}R \qquad \frac{\Psi; \Delta \vdash Q :: (d : D)}{\Psi; \Delta, c : \mathbf{1} \vdash \mathsf{wait}\ c\ ;\ Q :: (d : D)}\ \mathbf{1}L$$

$$\frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R \qquad \frac{\Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{:}\tau.\ A \vdash n \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L$$

$$\frac{\Psi, n{:}\tau\ ;\ \Delta \vdash P :: (c : A)}{\Psi\ ;\ \Delta \vdash n \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \forall n{:}\tau.\ A)}\ \forall R \qquad \frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \forall n{:}\tau.\ A \vdash \mathsf{send}\ c\ v\ ;\ Q :: (d : D)}\ \forall L$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: (c : B)}{\Psi\ ;\ \Delta, a : A \vdash \mathsf{send}\ c\ \ a\ ;\ P :: (c : A \otimes B)}\ \otimes R$$

$$\frac{\Psi; \Delta, x : A, c : B \vdash Q :: (d : D)}{\Psi; \Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \otimes L$$

$$\frac{\Psi; \Delta, x : A \vdash P :: (c : B)}{\Psi; \Delta \vdash x \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : A \multimap B)}\ \multimap R$$

$$\frac{\Psi\ ;\ \Delta, c : B \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, a : A, c : A \multimap B \vdash \mathsf{send}\ c\ a\ ;\ Q :: (d : D)}\ \multimap L$$

$$\frac{\Psi; \Delta \vdash P_\ell :: (c : A_\ell) \quad \text{for every } \ell \in L}{\Psi; \Delta \vdash \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (c : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

$$\frac{k \in L \quad \Psi; \Delta, c : A_k \vdash Q :: (d : D)}{\Psi; \Delta, c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\ ;\ Q :: (d : D)}\ \&L$$

$$\frac{k \in L \quad \Psi; \Delta \vdash P :: (c : A_k)}{\Psi; \Delta \vdash c.k\ ;\ P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

$$\frac{\Psi; \Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Psi; \Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

Figure 5.6: System T and S Typing

Both System T and S

$$\frac{\Psi \vdash v : \tau' \quad \Psi, x : \tau \,;\, \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi \,;\, \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^{\rho}\, v \,;\, Q :: (c : C)}\ \mathsf{val\_cast}$$

$$\frac{A \sim B}{\Psi, b \,;\, b : B \vdash a \leftarrow \langle A \Leftarrow B \rangle^{\rho}\, b :: (a : A)}\ \mathsf{id\_cast}$$

System T only

$$\frac{\Psi \vdash b : \mathsf{bool} \quad \Psi \,;\, \Delta \vdash Q :: (x : A)}{\Psi \,;\, \Delta \vdash \mathsf{assert}\ \rho\ b; Q :: (x : A)}\ \mathsf{assert}$$

$$\frac{}{\Psi \,;\, \Delta \vdash \mathsf{abort}\ \rho :: (x : A)}\ \mathsf{abort}$$

System S only

$$\frac{\Psi \vDash b\ \mathsf{true} \quad \Psi \,;\, \Delta \vdash Q :: (x : A)}{\Psi \,;\, \Delta \vdash \mathsf{assert}\ \rho\ b; Q :: (x : A)}\ \mathsf{assert\_strong}$$

Figure 5.6: System T and S Typing (continued)

$$[c : C/b : B](\mathsf{proc}(a, a \leftarrow \langle A \Leftarrow B\rangle^{\rho}\ b)) = \mathsf{proc}(a, a \leftarrow \langle A \Leftarrow C\rangle^{\rho}\ c)$$

$$[c : C/a : A](\mathsf{proc}(a, a \leftarrow \langle A \Leftarrow B\rangle^{\rho}\ b)) = \mathsf{proc}(c, c \leftarrow \langle C \Leftarrow B\rangle^{\rho}\ b)$$

$$[h : A' \otimes B'/c : A \otimes B]\mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c; Q) = \mathsf{proc}(d, g \leftarrow \mathsf{recv}\ h; [h : B'/c : B][g : A'/x : A]Q)$$

$$[f : A'' \otimes B'/c : A \otimes B]\mathsf{proc}(c, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ (a : A'); P) = \mathsf{proc}(f, \mathsf{send}\ f\ \langle A'' \Leftarrow A'\rangle^{\rho}\ a\ ;\ [f : B'/c : B]P)$$

$$[h : A' \multimap B'/c : A \multimap B]\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c; P) = \mathsf{proc}(h, g \leftarrow \mathsf{recv}\ h; [h : B'/c : B][g : A'/x : A]P)$$

$$[f : A'' \multimap B/c : A \multimap B]\mathsf{proc}(d, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ (a : A'); Q) = \mathsf{proc}(d, \mathsf{send}\ f\ \langle A'' \Leftarrow A'\rangle^{\rho}\ a\ ;\ [f : B'/c : B]Q)$$

$$[f : \oplus\{\ell : A'_{\ell}\}_{\ell\in L}/c : \oplus\{\ell : A_{\ell}\}_{\ell\in L}]\mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_{\ell})_{\ell\in L}) = \mathsf{proc}(d, \mathsf{case}\ f\ (\ell \Rightarrow [f : A'_{\ell}/c : A_{\ell}]Q_{\ell})_{\ell\in L})$$

$$[f : \oplus\{\ell : A'_{\ell}\}_{\ell\in L}/c : \oplus\{\ell : A_{\ell}\}_{\ell\in L}]\mathsf{proc}(c, c.k; P) = \mathsf{proc}(f, f.k; [f : A'_{\ell}/c : A_{\ell}]P)$$

$$[f : \&\{\ell : A'_{\ell}\}_{\ell\in L}/c : \&\{\ell : A_{\ell}\}_{\ell\in L}]\mathsf{proc}(d, c.k; Q) = \mathsf{proc}(d, f.k; [f : A'_{\ell}/A_{\ell}]Q)$$

$$[f : \&\{\ell : A'_{\ell}\}_{\ell\in L}/c : \&\{\ell : A_{\ell}\}_{\ell\in L}]\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_{\ell})_{\ell\in L}) = \mathsf{proc}(f, \mathsf{case}\ f\ (\ell \Rightarrow [f : A'_{\ell}/c : A_{\ell}]P_{\ell})_{\ell\in L}))$$

$$[f : \exists n : \tau'.A'/c : \exists n : \tau.A]\mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c; Q) = \mathsf{proc}(d, m \leftarrow \mathsf{recv} f; [m : \tau'/n : \tau][f : A'/c : A]Q)$$

$$[f : \exists n : \tau''.A'/c : \exists n : \tau.A]\mathsf{proc}(c, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho}\ (v : \tau'); P) = \mathsf{proc}(f, \mathsf{send}\ f\ \langle \tau'' \Leftarrow \tau'\rangle^{\rho}\ (v : \tau'); [f : A'/c : A]P)$$

$$[f : \forall n : \tau''.A'/c : \forall n : \tau.A]\mathsf{proc}(d, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho}\ (v : \tau'); Q) = \mathsf{proc}(d, \mathsf{send}\ f\ \langle \tau'' \Leftarrow \tau'\rangle^{\rho}(v : \tau'); [f : A'/c : A]Q)$$

$$[f : \forall n : \tau'.A'/c : \forall n : \tau.A]\mathsf{proc}(c, n \leftarrow \mathsf{recv}\ c; P) = \mathsf{proc}(f, m \leftarrow \mathsf{recv} f; [m : \tau'/n : \tau][f : A'/c : A]\ ;\ P$$

$$[f : \downarrow A'/c : \downarrow A]\mathsf{proc}(c, \mathsf{send}\ c\ \mathsf{shift}; P) = \mathsf{proc}(f, \mathsf{send}\ f\ \mathsf{shift}; [f : A'/c : A]P)$$

$$[f : \downarrow A'/c : \downarrow A]\mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d; Q) = \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d; [f : A'/c : A]Q)$$

$$[f : \uparrow A'/c : \uparrow A]\mathsf{proc}(d, \mathsf{send}\ d\ \mathsf{shift}; Q) = \mathsf{proc}(d, \mathsf{send}\ d\ \mathsf{shift}; [f : A'/c : A]Q)$$

$$[f : \uparrow A'/c : \uparrow A]\mathsf{proc}(c, \mathsf{shift}\ \mathsf{recv}\ c; P) = \mathsf{proc}(f, \mathsf{shift}\ \mathsf{recv}\ f; [f : A'/c : A]P)$$

$$[m : \tau''/v : \tau'](\mathsf{proc}(a, x \leftarrow \langle \tau \Leftarrow \tau'\rangle^{\rho}\ v; Q) = \mathsf{proc}(a, x \leftarrow \langle \tau \Leftarrow \tau''\rangle^{\rho}\ m; Q)$$

$$[k : A''/a : A']\mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ;\ c \leftarrow c') = \mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A''\rangle^{\rho} k\ ;\ c \leftarrow c')$$

$$[k : A''/c : A]\mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho} a\ ;\ c \leftarrow c') = \mathsf{msg}(k, \mathsf{send}\ k\ \langle A'' \Leftarrow A'\rangle^{\rho} a\ ;\ k \leftarrow c')$$

$$[m : \tau''/v : \tau']\mathsf{msg}(c, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau'\rangle^{\rho} v\ ;\ c' \leftarrow c) = \mathsf{msg}(c, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau''\rangle^{\rho} m\ ;\ c' \leftarrow c)$$

Figure 5.7: Substitution Rules

$$
\begin{array}{lcl}
\mathsf{id\_cast} & : & \mathsf{proc}(a, a \leftarrow \langle A \Leftarrow A' \rangle^{\rho} b), \mathcal{C} \longrightarrow [b : A'/a : A]\mathcal{C} \\
\mathsf{val\_cast} & : & \mathsf{proc}(a, x \leftarrow \langle \tau' \Leftarrow \tau \rangle^{\rho}\ v; Q) \longrightarrow \mathsf{proc}(a, Q[v : \tau'/x : \tau]) \\
\mathsf{cut} & : & \mathsf{proc}(c, x{:}A \leftarrow \langle A \Leftarrow A' \rangle^{\rho}\ P\ ;\ Q) \\
 & & \longrightarrow \mathsf{proc}(a, [a : A'/x : A']P), \mathsf{proc}(c, [a : A'/x : A]Q) \quad \textit{(a fresh)} \\
\mathsf{tensor\_s} & : & \mathsf{proc}(c, \mathsf{send}\ c\ (a : A')\ ;\ P) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : B/c : B]P), \mathsf{msg}(c : B, \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^{\rho} a\ ;\ c \leftarrow c') \quad \textit{(c' fresh)} \\
\mathsf{tensor\_r} & : & \mathsf{msg}(c : B, \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^{\rho} a\ ;\ c \leftarrow c'), \mathsf{proc}(d, (x : A) \leftarrow \mathsf{recv}\ c\ ;\ Q) \\
 & & \longrightarrow \mathsf{proc}(d, [c' : B/c : B][a : A'/x : A]Q) \\
\mathsf{lolli\_s} & : & \mathsf{proc}(d, \mathsf{send}\ c\ (a : A')\ ;\ Q) \\
 & & \longrightarrow \mathsf{msg}(c' : B, \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^{\rho} a\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c' : B/c : B]Q) \quad (c' \text{ fresh}) \\
\mathsf{lolli\_r} & : & \mathsf{proc}(c, (x : A) \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c' : B, \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^{\rho} a\ ;\ c' \leftarrow c) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : B/c : B][a : A'/x : A]P) \\
\mathsf{exists\_s} & : & \mathsf{proc}(c, \mathsf{send}\ c\ (v : \tau')\ ;\ P) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : [v/n]A/c : [v/n]A]P), \\
 & & \mathsf{msg}(c : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^{\rho} v\ ;\ c \leftarrow c') \\
\mathsf{exists\_r} & : & \mathsf{msg}(c : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^{\rho} v\ ;\ c \leftarrow c'), \mathsf{proc}(d, (n : \tau) \leftarrow \mathsf{recv}\ c\ ;\ Q) \\
 & & \longrightarrow \mathsf{proc}(d, [c' : A/c : A][v : \tau'/n : \tau]Q) \\
\mathsf{forall\_s} & : & \mathsf{proc}(d, \mathsf{send}\ c\ (v : \tau')\ ;\ Q) \\
 & & \longrightarrow \mathsf{msg}(c' : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^{\rho} v\ ;\ c' \leftarrow c), \\
 & & \mathsf{proc}(d, [c' : [v/n]A/c : [v/n]A]Q) \\
\mathsf{forall\_r} & : & \mathsf{proc}(c, (n : \tau) \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c' : [v/n]A, \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^{\rho} v\ ;\ c' \leftarrow c) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : A/c : A][v : \tau'/n : \tau]P) \\
\mathsf{assert\_f} & : & \mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{False}; Q) \longrightarrow \mathsf{abort}(l) \\
\mathsf{assert\_s} & : & \mathsf{proc}(c, \mathsf{assert}\ l\ \mathsf{True}; Q) \longrightarrow \mathsf{proc}(c, Q)
\end{array}
$$

Figure 5.8: Typed Semantics with Casts

$$
\begin{array}{lcl}
\mathsf{plus\_s} & : & \mathsf{proc}(c, c.k\ ;\ P) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : A_k / c : A_k]P), \mathsf{msg}(c : A_k, c.k\ ;\ c \leftarrow c') \quad \textit{(}c' \textit{ fresh)} \\
\mathsf{plus\_r} & : & \mathsf{msg}(c : A_k, c.k\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L}) \\
 & & \longrightarrow \mathsf{proc}(d, [c' : A_\ell / c : A_\ell]Q_k) \\
\mathsf{with\_s} & : & \mathsf{proc}(d, c.k\ ;\ Q) \\
 & & \longrightarrow \mathsf{msg}(c' : A_k, c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c' : A_k / c : A_k]Q) \quad (c' \text{ fresh}) \\
\mathsf{with\_r} & : & \mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c' : A_k, c.k\ ;\ c' \leftarrow c) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : A_\ell / c : A_\ell]P_k) \\
\mathsf{close} & : & \mathsf{proc}(c, \mathsf{close}\ c) \longrightarrow \mathsf{msg}(c : \mathbf{1}, \mathsf{close}\ c) \\
\mathsf{wait} & : & \mathsf{msg}(c : \mathbf{1}, \mathsf{close}\ c), \mathsf{proc}(d, \mathsf{wait}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, Q) \\
\mathsf{down\_s} & : & \mathsf{proc}(c, \mathsf{send}\ c\ \mathsf{shift}\ ;\ P) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : A^- / c : A^-]P), \mathsf{msg}(c : A^-, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c') \quad (c' \text{ fresh}) \\
\mathsf{down\_r} & : & \mathsf{msg}(c : A^-, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c \leftarrow c'), \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q) \\
 & & \longrightarrow \mathsf{proc}(d, [c' : A^- / c : A^-]Q) \\
\mathsf{up\_s} & : & \mathsf{proc}(d, \mathsf{send}\ d\ \mathsf{shift}\ ;\ Q) \\
 & & \longrightarrow \mathsf{msg}(c' : A^+, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c' : A^+ / c : A^+]Q) \\
\mathsf{up\_r} & : & \mathsf{proc}(c, \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c' : A^+, \mathsf{send}\ c\ \mathsf{shift}\ ;\ c' \leftarrow c) \\
 & & \longrightarrow \mathsf{proc}(c', [c' : A^+ / c : A^+]P)
\end{array}
$$

Figure 5.8: Typed Semantics with Casts (continued)

Chapter 6

Dependent Types as Contracts

While Chapter 5 discussed monitoring type refinements, a special case of dependent types, monitoring arbitrary dependent types presents unique challenges. Dependent type theories, such as Coq, Agda, and Nuprl, encode dependent types by integrating programs and proof objects. These proof objects must be generated, communicated over the system, and verified. Although verifying a proof object is analogous to type checking, transmitting proof objects requires significant infrastructure [25]. Due to this challenge, an approach to dependent contract checking that avoids communicating proof objects when possible is desirable. In certain cases, proof objects encode constraints that are possible to validate at runtime, such as the fact that an integer is positive. In these situations, it is unnecessary to send a proof object. We can only avoid sending a proof object if the rest of the computation does not depend on it being sent. The concept of *proof irrelevance* [28] captures the idea that some proofs play no computational role in the program. If a proof object is irrelevant, we can avoid sending it and check the truth value of the proposition directly. In this chapter, we encode dependent session types with proof objects [30, 36]. We then integrate proof irrelevance into our type system and explore the class of properties that can be monitored successfully. We also formalize a notion of erasure to verify that irrelevant terms do not need to be communicated. Finally, we prove that blame assignment (as defined in Chapter 3) remains correct with the addition of irrelevant terms and that our notion of erasure is correct.

6.1 Model

Proof irrelevance allows us to selectively hide portions of a proof or program. We express irrelevant expressions in our language by using the bracket operator, $[v]$, meaning that there is a term v that is irrelevant from a computational point of view. We add a new category of assumptions $x \div \tau$ meaning that x stands for a term of type τ that is not computationally available. We then extend our session types to include quantifiers over irrelevant types $\forall x \div \tau.A$ and $\exists x \div \tau.A$. We note that for $\forall x \div \tau.A$ and $\exists x \div \tau.A$ to be well-formed, x cannot appear in A.

$$A ::= ... \mid \forall x : \tau.A(x) \mid \exists x : \tau.A(x) \mid \forall x \div \tau.A \mid \exists x \div \tau.A$$

For example, consider the following type:

$$\forall x : \mathsf{int}.\forall p : (x > 0).\exists y : \mathsf{int}.\exists q : (y > 0).\mathbf{1}$$

This type models an interaction where the integer x is first received. Then, a proof p of the fact that x is positive, is received. Next, the integer y is sent. Similarly, a proof q of the fact that y is positive is sent. Finally, the interaction terminates. By looking at the proof objects, we notice that they are not relevant to the rest of the computation. Moreover, given an integer, it is possible to check whether it is positive directly, and sending a proof of its positivity is unnecessary. Therefore, we can rewrite the type as follows:

$$\forall x : \mathsf{int}.\forall p \div [x > 0].\exists y : \mathsf{int}.\exists q \div [y > 0].\mathbf{1}$$

There are two irrelevant proof objects, $[x > 0]$ and $[y > 0]$, present in this type. Instead of receiving an actual proof p of the fact that x is positive, the condition $[x > 0]$ is checked dynamically and a unit element [] is received instead. Similarly, the constraint that $[y > 0]$ is dynamically verified, and the unit element [] is sent.

We define the context Ψ to contain both standard assumptions of the form $b : \tau$ and irrelevant assumptions of the form $x \div \tau$.

$$\Psi := \cdot \mid \Psi, b \div \tau \mid \Psi, c : \tau$$

The "irrelevant" proof objects must exist in the system for the purpose of type-checking, but they cannot have a computational effect on the program. In order to facilitate this, the typing rules must carefully ensure that hidden proofs are never required to compute something that is itself not hidden. That is, computationally *relevant* portions cannot depend on computationally *irrelevant* portions. For simplicity, we consider irrelevance only for messages to be exchanged, not internally in the functional layer. We define a promotion operation on contexts that transforms computationally irrelevant hypotheses into standard ones to account for type checking within the bracket operator.

Definition 3 (Promotion)**.**

$$(\cdot)^{\oplus} = \cdot \quad (\Psi, x : \tau)^{\oplus} = \Psi^{\oplus}, x : \tau \quad (\Psi, x \div \tau)^{\oplus} = \Psi^{\oplus}, x : \tau$$

The typing rules for both the relevant and irrelevant dependent process expressions are shown in Figure 6.1. We note that in the $\exists R_{[]}$ and $\forall L_{[]}$ rules, the context Ψ is promoted to appropriately typecheck the value v against the type τ. The type τ can be a base type (including a proof) or a dependent type. Values include integers, booleans or proof objects.

$$\tau := b \mid \Pi x : \tau_1.\tau_2 \mid \Sigma x : \tau_1.\tau_2$$

$$v := \mathsf{true} \mid \mathsf{false} \mid n \mid p \mid \ldots$$

In addition, we assume an underlying dependent type theory with the following properties:

- (Substitution) If $\Psi_1 \vdash v : \tau_1$ and $\Psi_1, c : \tau_1, \Psi_2 \vdash e : \tau$ then $\Psi_1, [v/c]\Psi_2 \vdash [v/c]e : [v/c]\tau$.
- (Weakening) If $\Psi_1 \vdash e : \tau$ then $\Psi_1, \Psi_2 \vdash e : \tau$.
- (Irrelevance) If $\Psi_1, c \div \tau', \Psi_2 \vdash e : \tau$ then $\Psi_1, \Psi_2 \vdash e : \tau$.

$$\frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R \qquad \frac{\Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{:}\tau.\ A \vdash n \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L$$

$$\frac{\Psi, n{:}\tau\ ;\ \Delta \vdash P :: (c : A)}{\Psi\ ;\ \Delta \vdash n \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \forall n{:}\tau.\ A)}\ \forall R \qquad \frac{\Psi \vdash v : \tau \quad \Psi\ ;\ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \forall n{:}\tau.\ A \vdash \mathsf{send}\ c\ v\ ;\ Q :: (d : D)}\ \forall L$$

$$\frac{\Psi^{\oplus} \vdash v : \tau \quad \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ [v]\ ;\ P :: (c : \exists n{\div}\tau.\ A)}\ \exists R_{[]} \qquad \frac{\Psi, n{\div}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{\div}\tau.\ A \vdash [n] \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L_{[]}$$

$$\frac{\Psi, n{\div}\tau\ ;\ \Delta \vdash P :: (c : A)}{\Psi\ ;\ \Delta \vdash [n] \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \forall n{\div}\tau.\ A)}\ \forall R_{[]} \qquad \frac{\Psi^{\oplus} \vdash v : \tau \quad \Psi\ ;\ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \forall n{\div}\tau.\ A \vdash \mathsf{send}\ c\ [v]\ ;\ Q :: (d : D)}\ \forall L_{[]}$$

Figure 6.1: Typing Dependent Process Expressions

Monitoring Rules The monitoring rules for both the relevant and irrelevant dependent processes are shown in Figure 6.3. There are two kinds of rules in our semantics, the first kind handles relevant values and the second kind handles erased irrelevant values.

The first category (exists_s, exists_s$_a$, exists_r, forall_s, forall_s$_a$, forall_r) consists of monitoring rules that are similar to those shown in Chapter 3. In these rules, the sent values are relevant. For example, in the exists_s rule, the monitor verifies that both the channel c_i and the value v match at type τ. If that is not the case, the exists_s$_a$ rule fires, and triggers an alarm. We note that the value v could be a proof object of some type ϕ. In that case, the monitor will verify that the proof object matches the type ϕ. This ensures that the proof object corroborates the required condition.

The second category (exists_s$_{[]}$, exists_s_a$_{[]}$, exists_r$_{[]}$, forall_s$_{[]}$, forall_s_a$_{[]}$, forall_r$_{[]}$) contains rules that handle erased terms. For example, in the exists_s$_{[]}$ rule, the monitor does not have access to the value sent over channel c_i so it cannot check that the value is of the appropriate type. The only knowledge the monitor has is the type of the erased proof term ϕ which represents a condition. Using this type, the monitor will attempt to construct a proof p that will match the type ϕ. If a proof cannot be found, then the exists_s_a$_{[]}$ rule executes, and raises an alarm. As before, we assume that the monitor is a trusted component of the system and cannot generate faulty proofs.

In order for our monitor to construct a proof of some proposition, it must be able to check if the proposition is true or false. Some propositions, such as $[x > 0]$, lie in a naturally decidable fragment, such as Presburger arithmetic. However, other propositions, such as $[\mathsf{is\_prime}(x)]$, cannot be expressed in Presburger arithmetic. Many of these propositions, including primality, can be expressed in Peano arithmetic, which is undecidable in general. That is, while a procedure to check primality can be written, an algorithm may not exist to check the truth value of every proposition in this fragment. In this thesis, we call the propositions that can be expressed in Presburger arithmetic "decidable" because there is an algorithm to determine if the proposition is true or false, and we call the propositions that do not lie in this fragment "hard" because such an algorithm may not exist. In our monitoring semantics, the type of the proof term ϕ must be decidable.

The taxonomy of proof objects is summarized in Figure 6.2. When proof objects are decid-

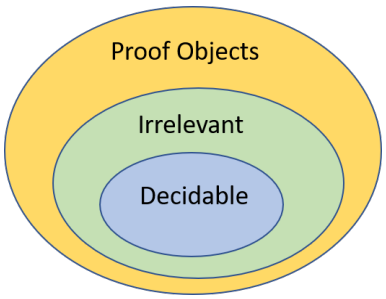

Figure 6.2: Proof Object Taxonomy

able (blue oval) our monitors can construct a proof in order to verify the desired condition. When proof objects are irrelevant, but hard (green oval), our monitors will try to construct a proof and may fail. When a proof cannot be generated, it is necessary to transmit the actual proof object, even though the actual proof object may be irrelevant to the rest of the computation. In this case, the sent proof will trigger one of the relevant monitoring rules. When a proof object is transmitted, our monitors will check that the received proof produces the expected result.

6.2 Examples

In this section, we present examples of various conditions that can be encoded as irrelevant proof objects. Our examples showcase conditions that can be validated by our monitors (proof objects located in the blue oval in Figure 6.2) as well as those that cannot (proof objects located in the green oval in Figure 6.2.

Factoring We first return to the factoring example, first presented in Section 4.1. We have a process that receives one positive integer n and factors it into two integers p and q that are sent back. We want to enforce the facts that $n = p * q$, and $p \leq q$. We write the factor_t type as follows:

$$\mathsf{factor\_t} = \forall n : \mathsf{int}.\forall r \div [n > 0].\exists p : \mathsf{int}.\exists q : \mathsf{int}.\exists s \div [n = p * q \wedge p \leq q].\mathbf{1}$$

In this example, because the specific values of n, p and q are known before the monitor needs to verify the condition $[n = p*q \wedge p \leq q]$, the check is decidable. That is, if $n = 35, p = 5, q = 7$, then the condition becomes $35 = 5 * 7 \wedge 5 \leq 7$ which is easily computed.

Let us now consider a different factoring implementation. In this implementation, we have a process that receives one positive integer n and factors it into two integers. However, in this implementation, the actual factors are not sent back, but instead a boolean acknowledgement of success b is sent. We want to enforce that if the boolean b is true that $n = p * q$ and $p \leq q$.

$$\mathsf{factor\_secret\_t} = \forall n : \mathsf{int}.\forall r \div [n > 0].\exists b : \mathsf{bool}.\exists s \div [b \supset \exists p.\exists q.n = p * q \wedge p \leq q].\mathbf{1}$$

In this situation, the monitor does not have access to the factors p and q, only the original integer n. Because of this, the condition $[b \supset \exists p.\exists q.n = p * q \wedge p \leq q]$ is hard. Even though

this particular proof object is irrelevant, it cannot be checked dynamically. In order to check this condition, an actual proof object will need to be sent.

Lists In Section 4.2 we provided monitors that check whether a list is empty or not. It is frequently useful to go a step further and verify that a list has a certain length. To check this contract, we could index a list by its length where $\mathsf{list}[l]$ has a length of l. We then express the list type as follows:

$$\begin{aligned}\mathsf{list}[l] = \oplus\{&\mathsf{nil} : \exists r \div [l = 0].\mathbf{1};\\ &\mathsf{cons} : \exists s \div [l > 0].\exists q : \mathsf{int}.\mathsf{list}[l-1]\}\end{aligned}$$

If the list is empty, the monitor will check that the length is indeed 0, and then the list will terminate. If the list is nonempty, the monitor will check that the length is positive. The list will then send the element y and continue to behave as a list, but now with the length decremented by one element.

In Section 4.2, we also presented a partial identity monitor that verified whether a list was in ascending order. Another method for validating this contract is to use a dependent type, where a $\mathsf{list}[x]$ is indexed by a bound x on its elements. Using this type, a list will initially have a bound of 0, $\mathsf{list}[0]$.

$$\begin{aligned}\mathsf{list}[x] = \oplus\{&\mathsf{nil} : \mathbf{1};\\ &\mathsf{cons} : \exists y : \mathsf{int}.\exists p \div [y \geq x].\mathsf{list}[y]\}\end{aligned}$$

In this encoding, if the list is nonempty, it will first send the element y. The monitor will then dynamically check the condition that $y \geq x$. Finally, it will continue behaving as a list, but it will now be indexed with the bound y.

Now let us assume that we not only desire a list in ascending order, but one that also consists of prime numbers. We can augment the type with the additional constraint as follows:

$$\begin{aligned}\mathsf{list}[x] = \oplus\{&\mathsf{nil} : \mathbf{1};\\ &\mathsf{cons} : \exists y : \mathsf{int}.\exists p \div [y \geq x \wedge \mathsf{is\_prime}(y)].\mathsf{list}[y]\}\end{aligned}$$

Though we can encode the primality condition as a proof-irrelevent object, it is hard. In order to enforce that the list consists of ascending primes, a primality-proving object will need to be transmitted.

Trees We consider a binary tree implementation that is parametrized by two operations, split and join [1]. In this implementation, all standard tree operations, such as insertion and deletion are implemented in terms of the split and join functions. Consider the following type which represents a tree:

$$\begin{aligned}\mathsf{tree} = \&\{&\mathsf{split} : \forall k : \mathsf{int}.\mathsf{tree} \otimes \exists l : \mathsf{bool}.\mathsf{tree} \otimes \mathbf{1};\\ &\mathsf{join} : \mathsf{tree} \multimap \mathsf{tree} \otimes \mathbf{1}\}\end{aligned}$$

If the split label is selected, an integer key is received. The implementation then splits the original tree of height h into two trees: one that consists of all elements of the original tree that are smaller than the given key (of height m), and one that consists of all elements of the original tree that are larger than the given key (of height n). Both of these trees are sent, along with a boolean value indicating whether the given key is in the tree. Finally, the process terminates. If the join label is selected, then a tree of height v is received and merged into the original tree of height h. The merged tree of height w is sent and then the process terminates.

In this implementation, we assume that trees are balanced, that is, a tree with x nodes has a height of $\lceil \log(x+1) \rceil$. We now consider the invariants we would like to enforce for this implementation. First, we require that keys and heights must be nonnegative. This is encoded by the irrelevant proof objects $[k \geq 0]$ and $[v \geq 0]$.

$$\begin{aligned}\mathsf{tree}[h] = \&\{&\mathsf{split} : \forall k : \mathsf{int}.\forall p \div [k \geq 0].\\ &\qquad \exists m : \mathsf{int}.\mathsf{tree}[m] \otimes \exists q \div [0 \leq m \leq h].\exists l : \mathsf{bool}.\\ &\qquad \exists n : \mathsf{int}.\mathsf{tree}[n] \otimes \exists r \div [0 \leq n \leq h].\mathbf{1};\\ &\mathsf{join} : \forall v : \mathsf{int}.\mathsf{tree}[v] \multimap \forall r \div [v \geq 0].\\ &\qquad \exists w : \mathsf{int}.\mathsf{tree}[w] \otimes \exists s \div [\mathsf{max}(v,h) \leq w \leq v+h].\mathbf{1}\}\end{aligned}$$

Second, we want to impose bounds on the heights of the trees created as a result of the split and join processes. When a tree of height h is split, the resulting trees of height m and n cannot have a height that is larger than that of the original tree, h. However, if the key being split on is smaller or greater than all of the keys in the original tree, then the height of m or n, respectively, could be 0. These bounds are expressed by the irrelevant proof objects $[0 \leq m \leq h]$ and $[0 \leq n \leq h]$.

When a tree of height v is being joined to a tree of height h, the height of the resulting tree w must be at least the height of the larger of the two trees being joined. In constrast, the height of the resulting tree w cannot be larger than the sum of the heights of the two trees being joined. This bound is represented by the irrelevant proof object $[\mathsf{max}(v,h) \leq w \leq v+h]$. Because the integer values k, h, m, n, v and w are all available to the monitor, the monitor is able to verify all of the conditions in this example.

Let us consider the proof object $[\mathsf{max}(v,h) \leq w \leq v+h]$. While the upper bound $v+h$, is a correct upper bound, it is too conservative. In fact, because we assumed our trees to be balanced, we can calculate a more accurate bound. Given that a balanced tree of height v has at most $2^v - 1$ nodes, the resulting tree of height m can have at most $(2^v - 1 + 2^h - 1)$ nodes. This means that the height of the merged tree m is bounded by $\lceil \log(2^v + 2^h - 1) \rceil$. Unfortunately, the condition $[\mathsf{max}(v,h) \leq w \leq \lceil \log(2^v + 2^h - 1) \rceil]$ is hard. Even though this proof object is irrelevant, it cannot be checked dynamically. In order to enforce the more accurate bound, an actual proof object will need to be sent.

6.3 Metatheory

In this section we prove three theorems. We first show that blame is assigned correctly in the presence of irrelevant terms. We then formalize a notion of erasure to verify that irrelevant terms do not need to be communicated at the process level. Finally, we show that our notion of erasure is correct and that erasure preserves the meanings of programs.

We first recall how blame assignment is defined in Chapter 3. Let the context Ω be the multiset of processes and messages describing the current state of computation. We define the context Δ to map linear channels in the system to their types. We define the context H to store channels that have been havoced. Due to channel renaming, if $a_i \in H$, then $a \in H$. We write $\models \Omega : \mathsf{wf}$ to denote that the state Ω is well-typed. This well-typedness requires that all processes and messages in Ω be typed using typing rules in Figure 6.1.

In order to prove that blame is assigned correctly, we state and prove a substitution lemma that applies for irrelevant terms.

Lemma 12 (Irrelevant Substitution)**.** *If* $\Psi^{\oplus} \vdash v : \tau$ *and* $\Psi, b \div \tau; \Delta \vdash Q :: (d : D)$ *then* $\Psi; [v/b]\Delta \vdash [v/b]Q :: (d : [v/b]D)$.

Proof. We prove this lemma by induction on the derivation of $\Psi, b \div \tau; \Delta \vdash Q :: (d : D)$. We show the principal cases in Appendix C.1. □

We now present the configuration typing in order to state the theorem. We assume that the comma operator is associative with $\cdot$ as the unit.

$$\mathcal{C} = \cdot \mid \mathsf{proc}(c, P) \mid \mathsf{msg}(c, P) \mid \mathcal{C}_1, \mathcal{C}_2$$

$$\frac{}{H; \cdot \Vdash \cdot} \qquad \frac{H; \Delta \Vdash \mathcal{C}_1 \quad H; \Delta \Vdash \mathcal{C}_2}{H; \Delta \Vdash \mathcal{C}_1, \mathcal{C}_2} \qquad \frac{\Delta_{|\mathsf{fn}(P)} \vdash P :: (c : \Delta(c))}{H; \Delta \Vdash \mathsf{msg}(c, P)}$$

$$\frac{c \notin H \quad \Delta_{|\mathsf{fn}(P)} \vdash P :: (c : \Delta(c))}{H; \Delta \Vdash \mathsf{proc}(c, P)} \qquad \frac{c \in H}{H; \Delta \Vdash \mathsf{proc}(c, P)}$$

Theorem 13 (Alarm)**.**

1. *If* $\emptyset; \cdot; \cdot \models \Omega$ *and* $\Omega, \emptyset, \longrightarrow^* \Omega'; H; \Delta$ *then* $H; \Delta \models \Omega'$
2. *If* $\emptyset; \cdot; \cdot \models \Omega$ *and* $\Omega, \emptyset, \longrightarrow^* \_, H, \mathsf{alarm}(a)$ *then* $a \in H$.

The above theorem states that from an initial configuration, a well-typed configuration can make a series of transitions to either another well-formed configuration, or a state where an alarm is raised on a process a that has been havoced.

The correctness proof for the blame assignment is similar to that of a preservation proof. The key lemma is Lemma 14, which states that if a well-typed configuration makes a transition, then it either steps to another well-formed configuration, or an alarm is raised on a process a that has been havoced. Using this lemma, we can prove Theorem 13 which considers a sequence of transitions. The proof is done by induction on the length of the trace.

Lemma 14 (One-step alarm)**.** *If* $H; \Delta \Vdash \Omega$ *and* $\mathsf{proc}(a_i, P) \in \Omega$ *then either:*

1. $H, \Omega, \Delta \to H, \Omega', \Delta'$ *and* $H; \Delta' \Vdash \Omega'$ *or*

2. $H, \Omega, \Delta \to \mathsf{alarm}(a)$ *where* $a \in H$ *and* $\Delta_{|\mathsf{fn}(P)} \not\vdash \mathsf{proc}(a_i, P) :: (a : \Delta(a))$.

Proof. We prove the lemma by examining each monitoring rule, inverting the typing configuration and applying the typing rules. The proof cases can be found in Appendix C.2. □

Now that we have augmented our system with irrelevant terms, the next step is to check that irrelevant terms do not need to be communicated at the process level. We formalize this with a notion of erasure that replaces computationally irrelevant terms by unit elements [] and irrelevant types by the unit type **1**.

Definition 4. *We define erasure on configurations and contexts below. Erasure for types and processes is defined in Figure 6.4.*

$$\cdot^\dagger = \cdot \quad (\mathsf{proc}(c, P))^\dagger = \mathsf{proc}(c, P^\dagger) \quad (\mathsf{msg}(c, P))^\dagger = \mathsf{msg}(c, P^\dagger) \quad (\mathcal{C}_1, \mathcal{C}_2)^\dagger = \mathcal{C}_1^\dagger, \mathcal{C}_2^\dagger$$

$$\cdot^\dagger = \cdot \quad (\Psi, x : \tau)^\dagger = \Psi^\dagger, x : \tau^\dagger \quad (\Psi, x \div \tau)^\dagger = \Psi^\dagger \quad (\Delta, c : A)^\dagger = \Delta^\dagger, c : A^\dagger$$

We begin by showing that our notion of erasure is correct. That is, if a process is well-typed with respect to some type, than the erased version of that process must also be well-typed with respect to the erased type.

Theorem 15 (Erasure Correctness)**.** *If* $\Psi, \Delta \vdash P :: (z : A)$ *then* $\Psi^\dagger, \Delta^\dagger \vdash P^\dagger :: (z : A^\dagger)$.

Proof. We prove this theorem by induction over the typing rules. The proof cases are shown in Appendix C.3. □

In order to show that erasure does not affect preservation, we need to show that substitution of irrelevant terms and erasure of irrelevant terms commute. In order to facilitate this, we prove a lemma that states if a process is well-typed with respect to some type given a context with an irrelevant term, then an erased version of that process must be well typed with respect to the erased type given the erased context.

Lemma 16 (Irrelevant Erasure)**.** *If* $\Psi^\oplus \vdash v : \tau$ *and* $\Psi, n \div \tau; \Delta \vdash Q :: (d : D)$ *then* $\Psi^\dagger; \Delta^\dagger \vdash Q^\dagger :: (d : D^\dagger)$.

Proof. We prove this theorem by induction over the derivation of $\Psi, n \div \tau; \Delta \vdash Q :: (d : D)$. The proof cases are shown in Appendix C.4. □

To show that erasure does not affect preservation, we are actually showing that erasure and process evaluation commute. We define two configurations as being equal if they are comprised of identical processes and messages.

Theorem 17 (Erasure Preservation)**.**

1. *If* $\cdot; \Omega \Vdash C$ *and* $C \to C'$ *then* $\exists C''$ *such that* $C^\dagger \to C''$ *and* $(C')^\dagger = C''$.
2. *If* $\cdot; \Omega; \Vdash C$, $\cdot; \Omega \Vdash C^\dagger$ *and* $C^\dagger \to C'^\dagger$ *then* $\exists C''$ *such that* $C \to C''$ *and* $C' = C''$.

Proof. We prove this theorem by induction over the operational semantics and by applying Lemma 16. The proof cases are shown in Appendix C.5. □

6.4 Related Work

Because dependent types subsume refinement types, the related work described in Section 5.4 is also relevant here. Greenberg et al. [17] survey the the literature on dependent contract checking, which focuses on dependent functions, and compares the technical approaches. Most relevant to our work is that of Ou et al. [27] who have developed a language where simply-typed and dependently-typed code is integrated by using coersions. Each coercion is treated as a contract and checked dynamically. Their coercions resemble the proof objects in our system which can be used to integrate simply-typed and dependently-typed code.

Recently, Toninho and Yoshida [35] have presented data-dependent session types, which are a version of dependent types that integrate dependent functions and session-typed processes. Their type theory can express protocols where the choice of the next communication action can depend on specific values of the received data. As an example, consider the below type:

$$\forall x : \mathsf{int}.\mathsf{if}\ (x > 0)(\exists y : \mathsf{int}.\mathbf{1})(\exists z : \mathsf{bool}.\mathbf{1})$$

This type represents a process that receives the integer x and checks whether it is positive. If it is, the process sends an integer y and then terminates. If not, the process sends a boolean z and terminates. In our system, we can represent this interaction as follows:

$$\forall x : \mathsf{int}.\ \oplus \{\mathsf{true} : \forall p \div [x > 0].\exists y : \mathsf{int}.\mathbf{1}; \mathsf{false} : \forall q \div [x \leq 0].\exists z : \mathsf{bool}.\mathbf{1}\}$$

In our version, the if connective has been transformed into an internal choice with two options, true and false. The guard to the if has been converted into two proof irrelevant terms for each branch of the if.

Along similar lines, Neykova et al. [26] design and implement a system for multiparty session types that features interaction refinements. Interaction refinements allow them to refine types of communicated messages and impose message dependent control flow. For example, they could require that first process P sends an positive integer to process Q, and then process Q sends the exact same integer to another process R. Our model is constrained to binary session types, but we are able to express interaction refinements. While their work focuses on providing a practical implementation in F#, our approach centers on theoretical guarantees.

$$
\begin{array}{lcl}
\mathsf{exists\_s} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P),\ !(c_i : \exists n : \tau.A)\ ,\ !(v : \tau) \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+),\ !(c_i^+ : [v/n]A) \\
\mathsf{exists\_s_a} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P), (\not\exists \tau.\ !(c_i : \exists n : \tau.A)\ ,\ !(v : \tau)\ ) \longrightarrow \mathsf{alarm}(c_i) \\
\mathsf{exists\_r} & : & \mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \\
& & \longrightarrow \mathsf{proc}(d, [c_i^+/c_i][v/n]Q) \\
\mathsf{exists\_s_{[]}} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P),\ !(\exists \phi.\exists p.c_i : \exists n \div \phi.A), p : \phi) \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+),\ !(c_i^+ : A) \\
\mathsf{exists\_s\_a_{[]}} & : & \mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P),\ !\neg(\exists \phi.\exists p.c_i : \exists n \div \phi.A), p : \phi) \longrightarrow \mathsf{alarm}(c_i) \\
\mathsf{exists\_r_{[]}} & : & \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [] \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \\
& & \longrightarrow \mathsf{proc}(d, [c_i^+/c_i][v/n]Q) \\
\mathsf{forall\_s} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q),\ !(c_i : \forall n : \tau.A)\ ,\ !(v : \tau) \\
& & \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q),\ !(c_i^+ : [v/n]A) \\
\mathsf{forall\_s_a} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q), (\not\exists \tau.\ !(c_i : \forall n : \tau.A)\ ,\ !(v : \tau)\ \longrightarrow \mathsf{alarm}(d) \\
\mathsf{forall\_r} & : & \mathsf{proc}(c_i, n \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i) \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P) \\
\mathsf{forall\_s_{[]}} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q),\ !(\exists \phi.\exists p.c_i : \forall n \div \phi.A), p : \phi) \\
& & \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q),\ !(c_i^+ : A) \\
\mathsf{forall\_s\_a_{[]}} & : & \mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q),\ !\neg(\exists \phi.\exists p.c_i : \forall n \div \phi.A), p : \phi) \longrightarrow \mathsf{alarm}(d) \\
\mathsf{forall\_r_{[]}} & : & \mathsf{proc}(c_i, [] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i) \\
& & \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P) \\
\mathsf{alarm_r} & : & \mathsf{proc}(c, m \leftarrow \mathsf{recv}\ a; R),\ !(a : A)\ ,\ !(m \not\triangleright A)\ \longrightarrow \mathsf{alarm}(c) \\
\mathsf{alarm_{r'}} & : & \mathsf{proc}(c, m \leftarrow \mathsf{recv}\ c; R),\ !(c : A)\ ,\ !(m \not\triangleright A)\ \longrightarrow \mathsf{alarm}(c)
\end{array}
$$

Figure 6.3: Dependent Monitor Rules

$(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger$ $\quad$ $(A \multimap B)^\dagger = A^\dagger \multimap B^\dagger$

$\&\{\ell : A_\ell\}^\dagger_{\ell \in L} = \&\{\ell : A^\dagger_\ell\}_{\ell \in L}$ $\quad$ $\oplus\{\ell : A_\ell\}^\dagger_{\ell \in L} = \oplus\{\ell : A^\dagger_\ell\}_{\ell \in L}$

$(\exists n : \tau.A)^\dagger = \exists n : \tau.A^\dagger$ $\quad$ $(\forall n : \tau.A)^\dagger = \forall n : \tau.A^\dagger$

$(\exists n \div \tau.A)^\dagger = \exists n \div 1.A^\dagger$ $\quad$ $(\forall n \div \tau.A)^\dagger = \forall n \div 1.A^\dagger$

$(\uparrow A)^\dagger = \uparrow A^\dagger$ $\quad$ $(\downarrow A)^\dagger = \downarrow A^\dagger$

$1^\dagger = 1$

$(\mathsf{close}\ c)^\dagger = \mathsf{close}\ c$ $\quad$ $(\mathsf{wait}\ c\ ;\ Q)^\dagger = \mathsf{wait}\ c\ ;\ Q^\dagger$

$(\mathsf{send}\ c\ a\ ;\ Q)^\dagger = \mathsf{send}\ c\ a\ ;\ Q^\dagger$ $\quad$ $x \leftarrow \mathsf{recv}\ c\ ;\ Q^\dagger$

$(c.\ell_j\ ;\ Q)^\dagger = c.\ell_j\ ;\ Q^\dagger$ $\quad$ $(\mathsf{send}\ c\ v\ ;\ Q)^\dagger = \mathsf{send}\ c\ v\ ;\ Q^\dagger$

$(n \leftarrow \mathsf{recv}\ c\ ;\ Q)^\dagger = n \leftarrow \mathsf{recv}\ c\ ;\ Q^\dagger$ $\quad$ $(\mathsf{send}\ c\ [v]\ ;\ Q)^\dagger = \mathsf{send}\ c\ []\ ;\ Q^\dagger$

$([n] \leftarrow \mathsf{recv}\ c\ ;\ Q)^\dagger = [] \leftarrow \mathsf{recv}\ c\ ;\ Q^\dagger$ $\quad$ $(\mathsf{send}\ c\ \mathsf{shift}\ ;\ Q)^\dagger = \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q^\dagger$

$(\mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q)^\dagger = \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ Q^\dagger$ $\quad$ $(x \leftarrow P\ ;\ Q)^\dagger = x \leftarrow P^\dagger\ ;\ Q^\dagger$

$(c \leftarrow d)^\dagger = c \leftarrow d$

$(\mathsf{case}\ c\ \mathsf{of}\ \{\ell_i \Rightarrow Q_i\}_i)^\dagger = \mathsf{case}\ c\ \mathsf{of}\ \{\ell_i \Rightarrow Q^\dagger_i\}_i)$

Figure 6.4: Erasure for Types and Processes

Chapter 7

Miscellaneous Monitors

In this thesis, we have argued that session-typed monitors allow us to monitor a variety of different classes of contracts. We have considered examples of contracts that are expressed as session types, partial-identity processes, refinement types, and dependent types. With each class of contract, we have provided a monitoring mechanism and shown it to be safe and transparent.

Throughout this thesis, we have returned to the example of factoring twice – once in Chapter 4 and once in Chapter 6. In the former case, we used a partial-identity monitor and in the latter case we were able to express the desired contract using a dependent type. In order to monitor our contracts, we made use of two approaches, shown in Figure 7.1. In the first approach, we increased the complexity of our type system. We first implemented this design in Chapter 3 where used monitors to check that processes were adhering to their prescribed session-types. We later augmented our system to handle proof objects in Chapter 6. In these settings, we were able to prove strong blame theorems. In the second approach, we made our monitors more powerful and increased their operational complexity. In Chapter 4 our monitors were fully-fledged processes that had the ability to spawn other processes. While these monitors were the most general and allowed us the most flexibility, proving a strong blame theorem proved difficult. When we restricted our monitors to refinements in Chapter 5, we were able to prove a safety theorem for that fragment. While refinements can be handled by partial identity monitors, they can also be monitored with the dependent monitors described in Chapter 6.

Despite the variety of contracts presented in this thesis, there are still classes of contracts that our monitors are not capable of validating. In this chapter, we discuss three types of monitors that do not yet fit into our session-typed monitoring model.

7.1 Partial Identity Processes with Unrestricted Channels

We first recall the partial identity processes discussed in Chapter 4. As an example, let us consider how to create a stream that consists of the bitwise logical disjunction (denoted $\vee$) of two streams of bits, implemented on channels x and y. The first method involves defining a process that examines each corresponding bit of x and y and then computes the resulting bit appropriately to output on channel z. The second method uses the negation of logical or (denoted $\downarrow$) and the following fact from propositional logic:

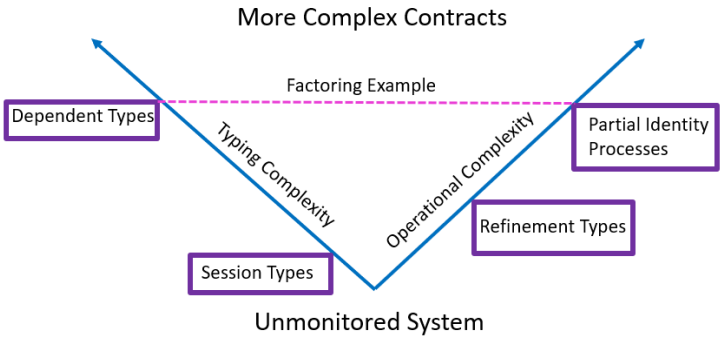

Figure 7.1: Monitoring Summary

$$x \vee y = (x \downarrow y) \downarrow (x \downarrow y)$$

We note that the channels x and y cannot be linear in order to use the above property to implement *or* using nor, because each of them is used twice. Let the standard implementation be called or_std, and the one using nor be called or_nor. The implementation of or_nor is shown in Figure 7.2.

We would like to enforce the contract that the or_nor process produces the same result as or_std process does. A monitor for this contract would need access to the channels x and y in order to pass those channels along to the or_std process. The monitor will also need to make use of a process to check whether two streams of bits have the same bits (referred to as eq). The code for this monitor is shown in Figure 7.2.

$z \leftarrow \mathsf{or\_nor} \leftarrow x, y =$
  $u \leftarrow \mathsf{nor} \leftarrow x, y$;
  $z \leftarrow \mathsf{nor} \leftarrow u, u$

$z \leftarrow \mathsf{or\_mon} \leftarrow x, y, z' =$
  $z'' \leftarrow \mathsf{or\_1} \leftarrow x, y$;
  $z \leftarrow \mathsf{eq} \leftarrow z', z''$

Figure 7.2: Or/Nor Monitor

Unfortunately, this monitor does not meet our criteria for a partial identity monitor because it takes multiple channels as an argument, even though two of them are non-linear. The goal of this future work is to investigate whether we can relax the partial identity rules to allow passing non-linear channels as arguments to monitoring processes. This relaxation would allow us to model another class of contracts, but would necessitate verifying that the changes do not jeopardize the transparency property of our monitors.

7.2 Monitoring Information Flow

Another avenue of future work is monitoring 2-safety properties such as information flow in this concurrent setting. In this setting, the contract we are looking to enforce is noninterference which guarantees that low-security information cannot influence high-security information. We are investigating the use of secure multiexecution to be able to run a process in both a high and low setting to determine whether it breaks noninterference. We have built an initial prototype of a secure multiexection engine which takes in a process, produces a low-security copy (where all high security data is set to some default value) and a high-security copy and then given some input runs the high-security and low-security copies concurrently. While secure-multiexecution provides strong security guarantees, in practice it is inefficient because it requires every block of code to be executed multiple times in parallel, once for each security level. Because we are already working in a concurrent setting, we are interested in investigating whether secure multiexecution can be implemented to have a lower performance overhead by taking advantage of the existing concurrency in our system. The goal of this research direction is to produce of a model of information flow enforcement in a concurrent process-based setting and prove that our enforcement mechanism satisfies noninterference.

7.3 Monitoring Deadlock

In a purely linear setting, session-typed languages guarantee deadlock freedom. Unfortunately, programs that rely on the use of shared resources cannot be expressed in this paradigm. The shared channels introduced in Section 2.1 have a copying semantics and therefore do not allow the sharing of mutable resources. Balzer and Pfenning [3] augment a session-typed language with resource sharing by splitting the language into linear and shared layers with modal operators connecting the layers. They impose an acquire-release discipline on shared channels which grants exclusive access to a process. This language is no longer deadlock-free, and deadlock can arise when executing standard examples such the dining philosophers scenario. More specifically, acquire-release introduces nondeterminism into the languages because any one of multiple clients trying to aquire a shared process could succeed.

Recently, Balzer et al [4] have developed a type system where types constrain the order of resources that a process may acquire. This information is used to determine when cyclic dependencies, which can lead to deadlock, occur. As a result, their system statically guarantees deadlock freedom for session-typed languages with shared resources. We are currently working on designing a monitoring mechanism that can dynamically detect deadlock as the system executes. Because deadlock is a global property, this mechanism involves monitors that must communicate with each other to determine whether a process can get struck trying to acquire a shared resource. Our prototype implementation uses the Mitchell-Merritt algoritm [24] to build a graph of what process is waiting for another process to figure out when deadlock has occured. The goal of this future work is to expand our prototype to model more examples and develop a theory that underlies our monitoring infrastructure.

Appendix A

Proof Cases

A.1 Verified Spawn Configuration Inversion Lemma

Lemma 18 (Configuration-Inversion)**.**

1. *If* $H; \Lambda \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$ *where* $\mathcal{C}_2 = \mathsf{proc}(c, P)$ *then* $H; \Lambda \Vdash \mathcal{C}_1$ *and* $H; \Lambda \Vdash \mathsf{proc}(c, P)$.
2. *If* $H; \Lambda \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ *where* $\mathcal{C}_2 = \mathsf{msg}(c, P)$ *and* $\mathcal{C}_3 = \mathsf{proc}(d, Q)$ *then* $H; \Lambda \Vdash \mathcal{C}_1$ *and* $H; \Lambda \Vdash \mathsf{msg}(c, P)$ *and* $H; \Lambda \Vdash \mathsf{proc}(d, Q)$.
3. *If* $H; \Lambda \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ *where* $\mathcal{C}_2 = \mathsf{proc}(c, P)$ *and* $\mathcal{C}_3 = \mathsf{msg}(d, Q)$ *then* $H; \Lambda \Vdash \mathcal{C}_1$ *and* $H; \Lambda \Vdash \mathsf{proc}(c, P)$ *and* $H; \Lambda \Vdash \mathsf{msg}(d, Q)$.
4. *If* $H; \Lambda \Vdash \mathsf{proc}(c, P)$ *and* $c \notin H$*, then* $\Lambda_{|\mathsf{fn}P} \vdash P :: (c : \Lambda(c))$.
5. *If* $H; \Lambda \Vdash \mathsf{msg}(c, P)$ *then* $\Lambda_{|\mathsf{fn}(P)} \vdash P :: (c : \Lambda(c))$.

Proof. By examining the configuration typing rules. □

A.2 Freename Lemma

Lemma 19 (freename)**.** *If* $\Delta_{|CH} \vdash \mathsf{proc}(c, P) :: (c : \Delta(c))$ *then* $\mathsf{fn}(P) = CH$.

Proof. By induction over the structure of the typing judgement.

Case id

Let $R = a \leftarrow b$. We have that $\Delta_{|CH} = b$. By the id rule, we see that $\mathsf{fn}(R) = b$, which gives us $\mathsf{fn}(R) = CH$.

Case 1R

Let $R = \mathsf{close}\ c$. We have that $\Delta_{|CH} = \cdot$. By the 1R rule, we see that $\mathsf{fn}(R) = \cdot$, which gives us $\mathsf{fn}(R) = CH$.

Case 1L

Let $R = \mathsf{wait}\ c; Q$. We have that $\Delta_{|CH} = \Delta_{|CH \setminus c}, c : 1$. By I.H., $\mathsf{fn}(Q) = CH \setminus c$. By the 1L rule, we see that $\mathsf{fn}(R) = \mathsf{fn}(Q) \cup c$, which gives us $\mathsf{fn}(R) = CH \setminus c \cup c = CH$.

Case cut

Let $R = x : A \leftarrow P; Q$. We have that $\Delta_{|CH} = \Delta_{|CH_1}, \Delta_{|CH_2}$. By I.H., $\mathsf{fn}(P) = CH_1$ and $\mathsf{fn}(Q) = CH_2 \cup x$. By the cut rule, we see that $\mathsf{fn}(R) = \mathsf{fn}(P) \cup \mathsf{fn}(Q)\ x$, which gives us $\mathsf{fn}(R) = CH_1 \cup CH_2 \cup x \setminus x = CH$.

Case $\otimes$R

Let $R = \mathsf{send}\ c\ a; P$. We have that $\Delta_{|CH} = \Delta_{|CH\setminus a}, a : A$. By I.H., $\mathsf{fn}(P) = CH \setminus a$. By the $\otimes$R rule, $\mathsf{fn}(R) = \mathsf{fn}(P) \cup a$, which gives us $\mathsf{fn}(R) = \mathsf{fn}(P) \cup a = CH \setminus a \cup a = CH$.

Case $\otimes$L

Let $R = x \leftarrow \mathsf{recv}\ c; Q$. We have that $\Delta_{|CH} = \Delta_{|CH\setminus c}, c : A \otimes B$. By I.H., $\mathsf{fn}(Q) = \Delta_{|CH\setminus c} \cup c \cup x$. By the $\otimes$L rule, $\mathsf{fn}(R) = \mathsf{fn}(Q) \setminus x$, which gives us $\mathsf{fn}(R) = CH \setminus c \cup c \cup x \setminus x = CH$.

Case $\multimap$R

Let $R = x \leftarrow \mathsf{recv}\ c; P$. By i.h, $\mathsf{fn}(P) = CH \cup x$. By the $\multimap$R rule, $\mathsf{fn}(R) = \mathsf{fn}(P) \setminus x$, which gives us $\mathsf{fn}(R) = CH \cup x \setminus x = CH$.

Case $\multimap$L

Let $R = \mathsf{send}\ c\ a; Q$. We have that $\Delta_{|CH} = \Delta_{CH\setminus a\setminus c}, a : A, c : A \multimap B$. By I.H., $\mathsf{fn}(Q) = CH \setminus a \setminus c \cup c$. By the $\multimap$L rule, $\mathsf{fn}(R) = \mathsf{fn}(Q) \cup a$, which gives us $\mathsf{fn}(R) = CH \setminus a \setminus c \cup c \cup a = CH$.

Case $\uparrow R$

Let $R = \mathsf{shift} \leftarrow \mathsf{recv}\ c; P$. By I.H., $\mathsf{fn}(P) = CH$. By the $\uparrow$R rule, $\mathsf{fn}(R) = \mathsf{fn}(P)$, which gives us $\mathsf{fn}(R) = CH$.

The cases for $\uparrow L$, $\downarrow R$, $\downarrow L$ are similar.

Case $\&$R

Let $R = c.k; Q$. We have that $\Delta_{|CH} = \Delta_{|CH\setminus c}, c : \&\{\ell : A_\ell\}$. By I.H., $\mathsf{fn}(Q) = CH \setminus c \cup c = CH$. By the $\&$R rule, $\mathsf{fn}(R) = \mathsf{fn}(P)$ which gives us $\mathsf{fn}(R) = CH$.

The cases for $\&$L, $\oplus$R, $\oplus$L are similar.

Case $\exists$L

Let $R = n \leftarrow \mathsf{recv}\ c; Q$. We have that $\Delta_{|CH} = \Delta_{CH\setminus c}, c : \exists n : \tau.A$. By i.h, $\mathsf{fn}(Q) = CH \setminus c \cup c = CH$. By the $\exists$L rule, $\mathsf{fn}(R) = \mathsf{fn}(Q)$ which gives us $\mathsf{fn}(R) = CH$.

The cases for $\exists$R, $\forall$L, $\forall$R are similar.

□

A.3 Proof of lemma (one-step) for linear modality

Case id

We have $\mathsf{proc}(a, a \leftarrow b), \mathcal{C}$ $!(a : A)$, $!(b : A) \longrightarrow [b/a]\mathcal{C}$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(a, a \leftarrow b; \mathcal{C})$ and $H; \Delta \Vdash \mathcal{C}$.

Subcase $a \notin H$.

By inversion of the configuration typing, $\Delta_{|b} \vdash a \leftarrow b :: (a : \Delta(a))$. From the id typing rule, we see that $b : \Delta(a)$. Let $\Delta' = [b : \Delta(a)/a : \Delta(a)]$. We then have $H; \Delta' \Vdash [b/a]\mathcal{C}$.

Subcase $a \in H$.

Using the fact that $a : A$ and $b : A$ we can define Δ' as follows: $\Delta' = [b : \Delta(a)/a : \Delta(a)]$. We then have $H; \Delta' \Vdash [b/a]\mathcal{C}$.

Case $\mathsf{id_a}$

We have $\mathsf{proc}(a, a \leftarrow b), \mathcal{C}, (\not\exists A.\, !(a : A)\,, !(b : A)\,)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(a, a \leftarrow b; \mathcal{C})$ and $H; \Delta \Vdash \mathcal{C}$.

Subcase $a \notin H$.

By inversion of the configuration typing, $\Delta_{|b} \vdash a \leftarrow b :: (a : \Delta(a))$. By the id typing rule, we have $b : \Delta(a) \vdash a \leftarrow b :: (a : \Delta(a))$. If $a \not: \Delta(a)$ or $b \not: \Delta(a)$ then $b : \Delta(a) \nvdash a \leftarrow b :: (a : \Delta(a))$. This is a contradiction and because $H; \Delta \Vdash \mathsf{proc}(a, a \leftarrow b)$, it must be the case that $a \in H$.

Subcase $a \in H$.

Assume $\Delta_{|b} \vdash a \leftarrow b :: (a : \Delta(a))$. By the id rule, $b : A \vdash a \leftarrow b :: (a : A)$. However, this contradicts the fact that there does not exist an A such that $a : A$ and $b : A$. Therefore, $\Delta_{|b} \nvdash a \leftarrow b :: (a : \Delta(a))$.

Case cut

We have $\mathsf{proc}(c, x : A \leftarrow P; Q), !(\mathsf{typecheck}(P :: x : A))\,, !(\mathsf{fn}(P) \cap \mathsf{fn}(Q) = \emptyset\,) \longrightarrow \mathsf{proc}(c, [a_0/x]Q), \mathsf{proc}(a_0, [a_0/x]P), !(a_0 : A)\,$ where a_0 is fresh. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$.

Subcase $c \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$. By inversion of the cut typing rule, $\Delta_{|CH'} \vdash P :: (x : \Delta(x))$ and $\Delta_{|CH} \vdash Q :: (c : \Delta(c))$. By the Freename lemma, we know that $CH' = \mathsf{fn}(P)$ and $CH = \mathsf{fn}(Q)$. Let $\Delta' = [a_0 : A/x : A]\Delta$. Let $a_0 : A$, so by the substitution lemma we get $\Delta'_{|\mathsf{fn}(P)} \vdash [a_0 : A/x : A]P :: (a_0 : \Delta'(a_0)))$ and $\Delta'_{|\mathsf{fn}(Q)} \vdash [a_0 : A/x : A]Q :: (c : \Delta'(c))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c, [a_0/x]Q)$ and $H; \Delta' \Vdash \mathsf{proc}(a_0, [a_0/x]P)$.

Subcase $c \in H$

Using the fact that $P :: (x : A)$ we have $\Delta_{|\mathsf{fn}(P)} \vdash P :: (x : \Delta(a))$. Let $\Delta' = [a_0 : A/x : A]\Delta$. We then have $\Delta'_{|\mathsf{fn}(P)} \vdash [a_0/x]P :: (x : \Delta(a))$ which gives us $H; \Delta' \Vdash \mathsf{proc}(a_0, [a_0/x]P)$. Because $c \in H$, by the configuration typing, we have that $H; \Delta' \Vdash \mathsf{proc}(c, [a_0/x]Q)$.

Case $\mathsf{cut_a}$

We have $\mathsf{proc}(c, x : A \leftarrow P; Q), (\ !(\mathsf{typecheck}(P :: x :/ A))\ \oplus\ !(\mathsf{fn}(P) \cap \mathsf{fn}(Q) \neq \emptyset\ ) \longrightarrow \mathsf{alarm}(c)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$.

Subcase $c \notin H$.

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \vdash x : A \leftarrow P; Q :: (c : \Delta(c))$. By inversion of the cut typing rule, $\Delta_{|CH'} \vdash P :: (x : \Delta(x)))$ and $\Delta_{|CH} \vdash Q :: (c : \Delta(c))$. By the Freename lemma, we know that $CH' = \mathsf{fn}(P)$ and $CH = \mathsf{fn}(Q)$. Because Δ is a linear context, it must be the case that $\mathsf{fn}(P) \cap \mathsf{fn}(Q) = \emptyset$. This means $\Delta_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \nVdash x : A \leftarrow P; Q :: (c : \Delta(c))$ which is a contradiction. If $\mathsf{typecheck}(P :: x :/ A)$ then $\Delta_{|fn(P)} \nVdash P :: (x : \Delta(x)))$ and $\Delta_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \nVdash x : A \leftarrow P; Q :: (c : \Delta(c))$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$ it must be the case that $c \in H$.

Subcase $c \in H$

Assume $\Delta_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \vdash \mathsf{proc}(c, x : A \leftarrow P; Q) :: (c : \Delta(c))$. By inversion of the cut typing rule, $\Delta_{|CH'} \vdash P :: (x : \Delta(x)))$ and $\Delta_{|CH} \vdash Q :: (c : \Delta(c))$. By the Freename lemma, we know that $CH' = \mathsf{fn}(P)$ and $CH = \mathsf{fn}(Q)$. Because Δ is a linear context, it must be the case that $\mathsf{fn}(P) \cap \mathsf{fn}(Q) = \emptyset$. This means $\Delta_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \nVdash x : A \leftarrow P; Q :: (c : \Delta(c))$ which is a contradiction. If $\mathsf{typecheck}(P :: x :/ A)$ then $\Delta_{|fn(P)} \nVdash P :: (x : \Delta(x)))$ and $\Delta_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \nVdash x : A \leftarrow P; Q :: (c : \Delta(c))$.

Case $\mathsf{lolli\_s}$

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ a; Q), \ !(c_i : A_1 \multimap A_2)\ ), \ !(a : A_1)\ ) \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q), \ !(c_i^+ : A_2)\ $. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ a; Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup a \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ a; Q :: (d : \Delta(d))$. By inversion of the $\multimap$L rule, we have $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. By using the $\multimap$L rule and the id rule, we can type the message as follows: $a : A_1, c_i : A_2 \vdash \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i :: (c_i^+ : A_2)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|\mathsf{fn}(Q)} \vdash [c_i^+ : A_2/c_i : A_2]Q :: (d : \Delta'(d))$ and $\Delta'_{|a \cup c_i} \vdash \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i) :: (c_i^+ : A_2)$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$ and $H; \Delta' \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i)$.

Subcase $d \in H$

Using the fact that $a : A_1, c_i : A_1 \multimap A_2$ and $c_i^+ : A_2$ we can type the message as follows: $a : A_1, c_i : A_2 \vdash \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i :: (c_i^+ : A_2)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|a \cup c_i} \vdash \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i :: (c_i^+ : A_2)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i)$. Because $d \in H$, by the configuration typing, we have that $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case $\mathsf{lolli\_s_a}$

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ a; Q), (\nexists A_1.\ !(c_i : A_1 \multimap A_2)\ , \ !(a : A_1\ )$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ a; Q)$.

Subcase $d \notin H$.

By inversion of the configuration typing, $\Delta_{|c_i \cup a \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ a; Q :: (d : \Delta(d))$. By the $\multimap$L rule, we have $\Delta_{|\mathsf{fn}(Q)}, a : A_1, c_i : A_1 \multimap A_2 \vdash \mathsf{send}\ c_i\ a; Q :: (d : \Delta(d))$. If $c_i \not: A_1 \multimap A_2$, or $a \not: A_1$ then $\Delta_{|c_i \cup a \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ a; Q :: (d : \Delta(d))$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ a; Q)$, it must be the case that $d \in H$.

Subcase $d \in H$.

Assume $\Delta_{|c_i \cup a \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ a; Q :: (d : \Delta(d))$. By the $\multimap$L rule, $\Delta_{\mathsf{fn}(Q)}, a : A_1, c_i : A_1 \multimap A_2 \vdash \mathsf{send}\ c_i\ a; Q :: (d : \Delta(d))$. However, this contradicts the fact there does not exist A_1 such that $c_i : A_1 \multimap A_2$ and $a : A_1$. Therefore, $\Delta_{|c_i \cup a \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ a; Q :: (d : \Delta(d))$.

Case $\mathsf{lolli_r}$

We have $\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i), \mathsf{proc}(c_i, x \leftarrow \mathsf{recv}\ c_i; P) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][a/x]P)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, x \leftarrow \mathsf{recv}\ c_i; P)$ and $H; \Delta \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i)$.

Subcase $c_i \notin H$.

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash x \leftarrow \mathsf{recv}\ c_i; P :: (c_i : \Delta(c_i))$ and $\Delta_{c_i \cup a} \vdash \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i :: (c_i^+ : \Delta(c_i^+))$. Using the $\multimap$R rule and the id rule, we can type the message as follows: $a : A_1, c_i : A_2 \vdash \mathsf{send}\ c_i\ a; c_i^+ \leftarrow c_i :: (c_i^+ : A_2)$. By inversion of the $\multimap$R typing rule, we have that $\Delta_{|\mathsf{fn}(P)}, x : A_1 \vdash P :: (c_i : A_2))$. Let $\Delta' = [c_i^+ : A_2/c_i : A_2]\Delta \cup a$. We then have $\Delta'_{|\mathsf{fn} \cup a} \vdash [c_i^+/c_i][a/x]P :: (c_i^+ : A_2))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i][a/x]P)$.

Subcase $c_i \in H$

Let $\Delta' = \Delta$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i][a/x]P)$.

Case $\mathsf{tensor\_s}$

We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ a; P), !(c_i : A_1 \otimes A_2), !(a : A_1) \longrightarrow \mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+), \mathsf{proc}(c_i^+, [c_i^+/c_i]P), !(c_i^+ : A_2)$. By the configuration typing $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ a; P)$.

Subcase $c_i \notin H$.

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P) \cup a} \vdash \mathsf{send}\ c_i\ a\ ;\ P :: (c_i : \Delta(c_i))$. By inversion of the $\otimes R$ rule, we have $\Delta_{|\mathsf{fn}(P)} \vdash P :: (c_i : A_2)$. By using the $\otimes R$ rule and the id rule, we can type the message as follows: $a : A_1, c_i^+ : A_2 \vdash \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+ :: (c_i : A_2)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+ : A_2/c_i : A_2]P :: (c_i^+ : A_2)$ and $\Delta'_{|a \cup c_i^+} \vdash \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+ :: (c_i : A_2)$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$ and $H; \Delta' \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+)$.

Subcase $c_i \in H$.

Using the fact that $a : A_1$, $c_i^+ : A_2$, and $c_i : A_1 \otimes A_2$ we can type the message as follows: $a : A_1, c_i^+ : A_2 \vdash \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+ :: (c_i : A_2)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|a \cup c_i^+} \vdash \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+ :: (c_i : A_2)$. Therefore,$H; \Delta' \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+)$. Be-

cause $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$.

Case tensor_s$_a$
We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ a; P), (\not\exists A_1.\ !(c_i : A_1 \otimes A_2)\ ,\ !(a : A_1)\ ) \longrightarrow \mathsf{alarm}(c_i)$. By the configuration typing $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ a; P)$.

Subcase $c_i \notin H$.

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P) \cup a} \vdash \mathsf{send}\ c_i\ a\ ;\ P :: (c_i : \Delta(c_i))$. By the $\otimes$R rule, we have $\Delta_{|\mathsf{fn}(P)}, a : A_1 \vdash \mathsf{send}\ c_i\ a; P :: (c_i : A_1 \multimap A_2)$. If $c_i :\not A_1 \otimes A_2$ or $a :\not A_1$ then $\Delta_{|\mathsf{fn}(P) \cup a} \not\vdash \mathsf{send}\ c_i\ a; P :: (c_i : A_1 \otimes A_2)$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ a; P)$, it must be the case that $c_i \in H$.

Subcase $c_i \in H$.

Assume $\Delta_{|\mathsf{fn}(P) \cup a} \vdash \mathsf{send}\ c_i\ a; P :: \Delta(c_i)$. By the $\otimes$R rule, $\Delta_{|\mathsf{fn}(P)}, a : A_1 \vdash \mathsf{send}\ c_i\ a; P :: (c_i : A_1 \otimes A_2)$. However, this contradicts the fact that there does not exists A_1 such that $c_i : A_1 \otimes A_2$ and $a : A_1$. Therefore, $\Delta_{|\mathsf{fn}(P) \cup a} \not\vdash \mathsf{send}\ c_i\ a; P :: (c_i : \Delta(c_i))$.

Case tensor_r

We have $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+, \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c_i; Q)$,
$\longrightarrow \mathsf{proc}(d, [c_i^+/c_i][a/x]Q)$. By the configuration typing $H; \Delta \Vdash \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c_i; Q)$ and $H; \Delta \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+ : \Delta(c_i)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash x \leftarrow \mathsf{recv}\ c_i; Q :: (d : \Delta(d))$ and $\Delta_{a \cup c_i^+} \vdash \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+ :: (c_i : \Delta(c_i))$. Using the $\otimes$R and id typing rules we can type the message as follows: $a : A_1, c_i^+ : A_2 \vdash \mathsf{send}\ c_i\ a; c_i \leftarrow c_i^+ :: (c_i : A_2)$. By inversion of the $\otimes$L typing rule, we have that $\Delta_{|\mathsf{fn}(Q)}, x : A_1 \vdash Q :: (d : \Delta(d))$. Let $\Delta' = [c_i^+ : A_2/c_i : A_2]\Delta \cup a$. We then have $\Delta'_{|\mathsf{fn}(Q) \cup a} \vdash [c_i^+/c_i][a/x]Q :: (d : \Delta(d))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i][a/x]Q)$.

Subcase $d \in H$

Let $\Delta' = \Delta$. Because $c_i \in H$, by the configuration typing, we have
$H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i][a/x]Q)$.

Case one_s

We have $\mathsf{proc}(a, \mathsf{close}\ a), !(a : \mathbf{1}) \longrightarrow \mathsf{msg}(a, \mathsf{close}\ a)$. By the configuration typing, we have $H; \Delta \Vdash \mathsf{proc}(a, \mathsf{close}\ a)$.

Subcase $a \notin H$.

By inversion of the configuration typing, $\Delta_{|\cdot} \vdash \mathsf{close}\ a :: (a : \Delta(a))$. By the 1R rule, we have $\cdot \vdash \mathsf{close}\ a :: (a : 1)$. Let $\Delta' = \Delta$. We then have $\Delta'_{|\cdot} \vdash \mathsf{close}\ a :: (a : 1)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(a, \mathsf{close}\ a)$.

Subcase $a \in H$.

Using the fact that $a : 1$, we can type the message as follows: $\cdot \vdash \mathsf{close}\ a :: (a : 1)$. Let $\Delta' = \Delta$. We then have $\Delta'_{|\cdot} \vdash \mathsf{close}\ a :: (a : \Delta'(a))$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(a, \mathsf{close}\ a)$.

Case $\mathsf{one\_s_a}$

We have $\mathsf{proc}(a, \mathsf{close}\ a), !(a :\not\mathbf{1}) \longrightarrow \mathsf{alarm}(a)$. By the configuration typing, we have $H; \Delta \Vdash \mathsf{proc}(a, \mathsf{close}\ a)$.

Subcase $a \notin H$.

By inversion of the configuration typing, $\Delta_{|\cdot} \vdash \mathsf{close}\ a :: (a : \Delta(a))$. By the 1R rule, we have $\cdot \vdash \mathsf{close}\ a :: (a : 1)$. If $a :\not 1$, then $\cdot \nvdash \mathsf{close}\ a :: (a : 1)$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(a, \mathsf{close}\ a)$, it must be the case that $a \in H$.

Subcase $a \in H$.

Assume $\Delta_{|\cdot} \vdash \mathsf{close}\ a :: (a : \Delta(a))$. By the 1R rule, it must be the case that $a : 1$. However, this contradicts the fact that $a :\not 1$. Therefore, $\Delta_{|\cdot} \nvdash \mathsf{close}\ a :: (a : \Delta(a))$.

Case $\mathsf{one\_r}$

We have $\mathsf{msg}(a, \mathsf{close}\ a), \mathsf{proc}(d, \mathsf{wait}\ a; Q) \longrightarrow \mathsf{proc}(d, Q)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{wait}\ a; Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|a \cup \mathsf{fn}(Q)} \vdash \mathsf{wait}\ a; Q :: (d : \Delta(d))$. By inversion of the 1L rule, $\Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. Let $\Delta' = \Delta \setminus a$. We then have $\Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$ and $H; \Delta' \Vdash \mathsf{proc}(d, Q)$.

Subcase $d \in H$

Let $\Delta' = \Delta$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, Q)$.

Case alarm_r

We have $\mathsf{proc}(c, m \leftarrow \mathsf{recv}\ a; R), !(a : A), !(m \not\rhd A) \longrightarrow \mathsf{alarm}(c)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c, m \leftarrow \mathsf{recv}\ a; R)$.

Subcase $c \notin H$

By inversion of the configuration typing, $\Delta_{|a \cup \mathsf{fn}(R)} \vdash m \leftarrow \mathsf{recv}\ a; R :: (c : \Delta(c))$. In order to typecheck this process with any receiving rule, it must be the case that $m \rhd A$ which is a contradiction. Therefore, $\Delta_{|a \cup \mathsf{fn}(R)} \nvdash m \leftarrow \mathsf{recv}\ a; R :: (c : \Delta(c))$ and $c \in H$.

Subcase $c \in H$

Assume $\Delta_{|a \cup \mathsf{fn}(R)} \vdash m \leftarrow \mathsf{recv}\ a; R :: (c : \Delta(c))$. In order to typecheck this process with any receiving rule, it must be the case that $m \rhd A$ which is a contradiction. Therefore, $\Delta_{|a \cup \mathsf{fn}(R)} \nvdash m \leftarrow \mathsf{recv}\ a; R :: (c : \Delta(c))$ and $c \in H$.

Case alarm'_r

We have $\mathsf{proc}(c, m \leftarrow \mathsf{recv}\ c; R)$, $!(c : A)$, $!(m \not\triangleright A)$ $\longrightarrow$ $\mathsf{alarm}(c)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c, m \leftarrow \mathsf{recv}\ c; R)$.

Subcase $c \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(R)} \vdash m \leftarrow \mathsf{recv}\ c; R :: (c : \Delta(c))$. In order to typecheck this process with any receiving rule, it must be the case that $m \triangleright A$ which is a contradiction. Therefore, $\Delta_{|\mathsf{fn}(R)} \not\Vdash m \leftarrow \mathsf{recv}\ c; R :: (c : \Delta(c))$ and $c \in H$.

Subcase $c \in H$

Assume $\Delta_{|\mathsf{fn}(R)} \vdash m \leftarrow \mathsf{recv}\ c; R :: (c : \Delta(c))$. In order to typecheck this process with any receiving rule, it must be the case that $m \triangleright A$ which is a contradiction. Therefore, $\Delta_{|\mathsf{fn}(R)} \not\Vdash m \leftarrow \mathsf{recv}\ c; R :: (c : \Delta(c))$ and $c \in H$.

Case $\mathsf{down\_s}$

We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ P)$, $!(c_i :\downarrow A^-)$ $\longrightarrow$ $\mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ; c_i \leftarrow c_i^+)$, $!(c_i^+ : A^-)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ \mathsf{shift}; P :: (c_i : \Delta(c_i))$. By inversion of the $\downarrow$ R rule, we have $\Delta_{|\mathsf{fn}(P)} \vdash P :: (c_i : A^-)$. By using the $\downarrow$R and the id rule, we can type the message as follows: $c_i^+ : A^- \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+ :: (c_i : A^-)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+/c_i]P :: (c_i^+ : \Delta(c_i^+))$ and $\Delta'_{|c_i^+} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+ :: (c_i : A^-)$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$ and $H; \Delta'_{|c_i^+} \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+)$.

Subcase $c_i \in H$

Using the fact that $c_i^+ : A^-$ and $c_i : \downarrow A^-$ we can type the message as follows: $c_i^+ : A^- \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+ :: (c_i : A^-)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+ :: (c_i : A^-)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+)$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$.

Case $\mathsf{down\_s}_a$

We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ P)$, $!(c_i :\not\downarrow A^-)$ $\longrightarrow$ $\mathsf{alarm}(c_i)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ \mathsf{shift}; P :: (c_i : \Delta(c_i))$. By the $\downarrow$R rule, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ \mathsf{shift}; P :: (c_i : \downarrow A^-)$. If $\not\exists A. c_i : \downarrow A^-$ then $\Delta_{|\mathsf{fn}(P)} \not\Vdash \mathsf{send}\ c_i\ \mathsf{shift}; P :: (c_i : \Delta(c_i))$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ P)$, it must be the case that $c_i \in H$.

Subcase $c_i \in H$

Assume $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ \mathsf{shift}; P :: (c_i : \Delta(c_i))$. By the $\downarrow$R rule, it must be the case that

$c_i : \downarrow A^-$ for some A. However, this contradicts the fact that there does not exist A such that $c_i : \downarrow A^-$. Therefore, $\Delta_{|\mathsf{fn}(P)} \not\Vdash \mathsf{send}\ c_i\ \mathsf{shift}; P :: (c_i : \Delta(c_i))$.

Case down_r

We have $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q)$. By the configuration typing, we have that $H; \Delta \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+)$ and $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(Q)} \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ d\ ;\ Q) :: (d : \Delta(d))$ and $\Delta_{|c_i^+} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+ :: (c_i : \Delta(c_i))$. By using the $\downarrow$R and the id rule, we can type the message as follows: $c_i^+ : A^- \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i^+ :: (c_i : A^-)$. By inversion of the $\downarrow$L typing rule we have that $\Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. Let $\Delta' = [c_i^+ : A^-/c_i : A^-]\Delta$. We then have that $\Delta'_{|\mathsf{fn}(Q)} \vdash [c_i^+/c_i]Q :: (d : \Delta(d))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Subcase $d \in H$

Let $\Delta' = \Delta$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case up_s

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q), !(c_i :\uparrow A^+) \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q), !(c_i^+ : A^+)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ \mathsf{shift}; Q :: d : \Delta(d)$. By inversion of the $\uparrow$ L rule, we have $\Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. By using the $\uparrow$L and the id rule, we can type the message as follows: $c_i : A^+ \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i :: (c_i : A^+)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+ \cup \mathsf{fn}(Q)} \vdash [c_i^+/c_i]Q :: (d : \Delta'(d))$ and $\Delta'_{|c_i} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i \leftarrow c_i :: \Delta'(c_i))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$ and $H; \Delta' \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i)$.

Subcase $d \in H$

Using the fact that $c_i^+ : A^+$ and $c_i : \uparrow A^+$ we can type the message as follows: $c_i : A^- \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : A^-)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : A^+)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i)$. Because $d \in H$, by the configuration typing, we have , $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case up_s_a

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q), !(\not\exists A.A^+.c_i :\uparrow A^+) \longrightarrow \mathsf{alarm}(d)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q :: (d : \Delta(d))$. By the $\uparrow$L rule, $\Delta_{|\mathsf{fn}(Q)}, c_i : \uparrow A^+ \vdash \mathsf{send}\ c_i\ \mathsf{shift}; Q :: (d : \Delta(d))$. If $/\exists A.c_i \uparrow A^+$ then

$\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ \mathsf{shift}; Q :: (d : \Delta(d))$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q)$, it must be the case that $d \in H$.

Subcase $d \in H$

Assume $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q :: (d : \Delta(d))$. By the $\uparrow$L rule, $\Delta_{|\mathsf{fn}(Q)}, c_i : \uparrow A^+ \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q :: (d : \Delta(d))$. However, this contradicts the fact that there does not exist A such that $c_i : \uparrow A^+$. Therefore, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ Q :: (d : \Delta(d))$.

Case up_r

We have $\mathsf{proc}(c_i, \mathsf{shift} \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$. By the configuration typing, we have that $H; \Delta \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i)$ and $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{shift} \leftarrow \mathsf{recv}\ c_i\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c_i\ ;\ P :: (c_i : \Delta(c_i))$ and $\Delta_{|c_i} \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : \Delta(c_i^+))$. By using the $\uparrow$L and the id rule, we can type the message as follows: $c_i : A^+ \vdash \mathsf{send}\ c_i\ \mathsf{shift}\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : A^+)$. By inversion of the $\uparrow$R typing rule we have that $\Delta_{|\mathsf{fn}(P)} \vdash P :: (c_i : A^+)$. Let $\Delta' = [c_i^+ : A^+/c_i : A^+]\Delta$. We then have that $\Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+/c_i]P :: (c_i^+ : \Delta'(c_i^+))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$.

Subcase $c_i \in H$

Let $\Delta' = \Delta$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$.

Case exists_s

We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P), !(c_i : \exists n : \tau.A), !(v : \tau) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$, $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+), !(c_i^+ : [v/n]A)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ v\ ;\ P :: (c_i : \Delta(c_i))$. By inversion of the $\exists$R rule, we have $\Delta_{|\mathsf{fn}(P)} \vdash P :: (c_i : \Delta(c_i))$. By using the $\exists$R and the id rule, we can type the message as follows: $c_i^+ : [v/n]A \vdash \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n : \tau.A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+/c_i]P :: (c_i^+ : \Delta(c_i^+))$ and $\Delta'_{|c_i^+} \vdash \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n : \tau.A)$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$ and $H; \Delta'_{|c_i^+} \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+)$.

Subcase $c_i \in H$

Using the fact that $c_i^+ : [v/n]A$ and $c_i : \exists n : \tau.A$ we can type the message as follows: $c_i^+ : [v/n]A \vdash \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n : \tau.A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+} \vdash \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n : \tau.A)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+)$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, CH, [c_i^+/c_i]P)$.

Case exists_s$_a$

We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P), (\not\exists\tau.\ !(c_i : \exists n : \tau.A)\ ,\ !(v : \tau)\ ) \longrightarrow \mathsf{alarm}(c_i)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ v\ ;\ P :: (c_i : \Delta(c_i))$. By the $\exists R$ rule, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ v; P :: (c_i : \exists n : \tau.A)$. If $\not\exists\tau.c_i : \exists n : \tau.A, v : \tau$, then $\Delta_{|\mathsf{fn}(P)} \not\forall \mathsf{send}\ c_i\ v; P :: (c_i : \exists n : \tau.A)$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P)$, it must be the case that $c_i \in H$.

Subcase $c_i \in H$

Assume $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ v\ ;\ P :: (c_i : \Delta(c_i))$. By the $\exists$R rule, it must be the case that $c_i : \exists n : \tau.A$ for some τ. = However, this contradicts the fact that there does not exist τ such that $c_i : \exists n : \tau.A$ and $v : \tau$. Therefore, $\Delta_{|\mathsf{fn}(P)} \not\forall \mathsf{send}\ c_i\ v\ ;\ P :: (c_i : \Delta(c_i))$.

Case exists_r

We have $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c_i^+/c_i][v/n]Q)$. By the configuration typing, we have that $H; \Delta \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+)$ and $H; \Delta \Vdash \mathsf{proc}(d, n \leftarrow \mathsf{recv}\ c_i\ ;\ Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash n \leftarrow \mathsf{recv}\ c_i\ ;\ Q :: (d : \Delta(d))$ and $\Delta_{c_i^+} \vdash \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+ :: (c_i : \Delta(c_i)$. By using the $\exists$R and the id rule, we can type the message as follows: $c_i^+ : [v/n]A \vdash \mathsf{send}\ c_i\ v\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n : \tau.A)$. By inversion of the $\exists L$ typing rule we have that $n : \tau; \Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. By inversion of the $\exists$R typing rule, we have $\cdot \vdash v : \tau$. We now apply standard substitution to get: $[v/n]\Delta_{|\mathsf{fn}(Q)} \vdash [v/n]Q :: (d : [v/n]\Delta(d))$. Let $\Delta' = [c_i^+ : A^-/c_i : A^-]\Delta$ and $\Delta'_{|\mathsf{fn}(Q)} = [c_i^+ : A/c_i : A][v : \tau/n : \tau]\Delta_{|\mathsf{fn}(Q)}$. We then have that $\Delta'_{|\mathsf{fn}(Q)} \vdash [c_i^+/c_i]Q :: (d : \Delta(d))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Subcase $d \in H$

Let $\Delta' = \Delta$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case forall_s

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q), !(c_i : \forall n : \tau.A)\ , !(v : \tau)\ \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q), !(c_i^+ : [v/n]A)\ .$ By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ v\ ;\ Q :: (d : \Delta(d))$. By inversion of the $\forall L$ rule, we have $\Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. By using the $\forall$L and the id rule, we can type the message as follows: $c_i : \forall n : \tau.A \vdash \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : [v/n]A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+ \cup \mathsf{fn}(Q)} \vdash [c_i^+/c_i]Q :: (d : \Delta'(d))$ and $\Delta'_{|c_i} \vdash \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : \Delta'(c_i^+))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$ and $H; \Delta' \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i)$.

Subcase $d \in H$

Using the fact that $c_i^+ : [v/n]A$ and $c_i : \forall n : \tau.A$ we can type the message as follows: $c_i : \forall n : \tau.A \vdash \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : [v/n]A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i} \vdash \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : [v/n]A)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i)$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case forall_$\mathsf{s_a}$

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q),\ \not\exists\tau.\ !(c_i : \forall n : \tau.A)\ ,\ !(v : \tau)\ ) \longrightarrow \mathsf{alarm}(d)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ v\ ;\ P)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ v\ ;\ Q) :: (d : \Delta(d))$. By the $\forall L$ rule, $\Delta_{|\mathsf{fn}(Q)}, c_i : \forall n : \tau.A \vdash \mathsf{send}\ c_i\ v\ ;\ Q :: (d : \Delta(d))$. If $\not\exists \tau.c_i : \forall n : \tau.A, v : \tau$, then $\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ v\ ;\ Q :: (d : \Delta(d))$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ v\ ;\ Q)$, it must be the case $d \in H$.

Subcase $d \in H$

Assume $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ v\ ;\ Q :: (d : \Delta(d))$. By the $\forall$L rule, it must be the case that $c_i : \forall n : \tau.A$ for some τ. However, this contradicts the fact that there does not exist A such that $c_i : \forall n : \tau.A$. Therefore, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ v\ ;\ Q) :: (d : \Delta(d))$.

Case forall_r

We have $\mathsf{proc}(c_i, n \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P)$. By the configuration typing, we have that $H; \Delta \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i)$ and $H; \Delta \Vdash \mathsf{proc}(c_i, n \leftarrow \mathsf{recv}\ c_i\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash n \leftarrow \mathsf{recv}\ c_i\ ;\ P :: (c_i : \Delta(c_i))$ and $\Delta_{|c_i} \vdash \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i :: (c_i : \Delta(c_i))$. By the $\forall$L rule and the id rule, we can type the message as follows: $c_i : \forall n : \tau.A \vdash \mathsf{send}\ c_i\ v\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : [v/n]A)$. By inversion of the $\forall$R typing rule we have that $n : \tau; \Delta_{|\mathsf{fn}(P)} \vdash P :: (c_i : A)$. By inversion of the $\forall L$ typing rule, we have that $\cdot \vdash v : \tau$. We now apply standard substitution to get: $[v/n]\Delta_{\mathsf{fn}(P)} \vdash [v/n]P :: (c_i : [v/n]A))$. Let $\Delta' = [c_i^+ : A^-/c_i : A^-]\Delta$ and $\Delta'_{|\mathsf{fn}(P)} = [c_i^+ : A/c_i : A][v : \tau/n : \tau]\Delta_{|\mathsf{fn}(P)}$. We then have that $\Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+/c_i][v/n]P :: (c_i^+ : \Delta'(c_i^+))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$.

Subcase $c_i \in H$

Let $\Delta' = \Delta$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P)$.

Case havoc

We have $\mathsf{proc}(c, P),\ !\mathsf{fn}(P) \supseteq \mathsf{fn}(Q)\ \longrightarrow \mathsf{proc}(c, Q),\ !(\mathsf{havoc}(c))$. By the configuration typing, we have that $H; \Delta \Vdash \mathsf{proc}(c, P)$.

Subcase $c \notin H$

Let $\Delta' = \Delta$. Because $\mathsf{havoc}(c)$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c, Q)$.

Subcase $c \in H$

Let $\Delta' = \Delta$. Because $c \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c, Q)$.

Case with_s

We have $\mathsf{proc}(d, c_i.k; Q)$, $!(c_i : \&\{\ell : A_\ell\})$, $!(k \triangleright \&\{\ell : A_\ell\}) \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q)$, $\mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i)$, $!(c_i^+ : A_k)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, c_i.k; Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash c_i.k; Q :: (d : \Delta(d))$. By the inversion of the $\&$L rule, $\Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. By using the $\&$L and the id rule, we can type the message as follows: $c_i : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c_i.k; c_i^+ \leftarrow c_i :: (c_i^+ : A_k)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+ \cup \mathsf{fn}(Q)} \vdash [c_i^+/c_i]Q :: (d : \Delta'(d))$ and $\Delta'_{|c_i} \vdash c_i.k; c_i^+ \leftarrow c_i :: (c_i^+ : \Delta'(c_i^+))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q) : \Delta'(d)$ and $H; \Delta' \Vdash \mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i) : \Delta'(c_i^+)$.

Subcase $d \in H$

Because $k \triangleright \&\{\ell : A_\ell\}_{\ell \in L}$, it must be the case that $k \in L$. Using that fact and that $c_i^+ : A_k$, $c_i : \&\{\ell : A_\ell\}_{\ell \in L}$ we can type the message as follows: $c_i : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c_i.k; c_i^+ \leftarrow c_i :: (c_i^+ : A_k)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i} \vdash c_i.k; c_i^+ \leftarrow c_i :: (c_i^+ : A_k)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i)$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case with_s$_a$

We have $\mathsf{proc}(d, c_i.k; Q), (!(c_i \not: \&\{\ell : A_\ell\}) \oplus !(k \not\triangleright \&\{\ell : A_\ell\})) \longrightarrow \mathsf{alarm}(d)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, c_i.k; Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash c_i.k; Q :: (d : \Delta(d))$. By the $\&$L rule, $\Delta_{\mathsf{fn}(Q)}, c_i : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c_i.k; Q :: (d : \Delta(d))$. By inversion of the $\&$L rule, we have that $k \in L$. If $c_i \not: \&\{\ell : A_\ell\}_{\ell \in L}$ then $\Delta_{c_i \cup \mathsf{fn}(Q)} \not\vdash c_i.k; Q :: (d : \Delta(d))$. If $k \not\triangleright \&\{\ell : A_\ell\}_{\ell \in L}$ then $k \notin L$ which means that $\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash c_i.k; Q :: (d : \Delta(d))$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(d, c_i.k; Q)$, it must be the case that $d \in H$.

Subcase $d \in H$

Assume $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash c_i.k; Q :: (d : \Delta(d))$. By the $\&$L rule, we have that $\Delta_{\mathsf{fn}(Q)}, c_i : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c_i.k; Q :: (d : \Delta(d))$. However, this contradicts the fact that $c_i \not: \&\{\ell : A_\ell\}_{\ell \in L}$. By inversion of the $\&$L rule, it must also be the case that $k \in L$, which contradicts the fact that $k \not\triangleright \&\{\ell : A_\ell\}_{\ell \in L}$. Therefore, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash c_i.k; Q :: (d : \Delta(d))$.

Case with_r

We have $\mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i), \mathsf{proc}(c_i, \mathsf{case}\ c_i\{\ell \Rightarrow P_\ell\}_{\ell \in L}) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P_k)$. By the configuration typing, $H; \Delta \Vdash \mathsf{msg}(c_i^+, c_i.k; c_i^+ \leftarrow c_i)$ and $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{case}\ c_i\{\ell \Rightarrow P_\ell\}_{\ell \in L})$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{case}\ c_i\{\ell \Rightarrow P_\ell\}_{\ell \in L} :: (c_i : \Delta(c_i))$ and $\Delta_{|c_i} \vdash c_i.k; c_i^+ \leftarrow c_i :: (c_i^+ : \Delta(c_i^+))$. By using the &L and the id rule, we can type the message as follows: $c_i : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c_i.k; c_i^+ \leftarrow c_i :: (c_i^+ : A_k)$. By inversion of the &R typing rule, we have that $\Delta_{|\mathsf{fn}(P)} \vdash P_\ell :: (c_i : A_\ell)$. Let $\Delta' = [c_i^+ : A_k / c_i : A_k]\Delta$. We then have that $\Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+/c_i]P_k :: (c_i^+ : \Delta'(c_i^+))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P_k)$.

Subcase $c_i \in H$

Let $\Delta' = \Delta$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P_k)$.

Case plus_s

We have $\mathsf{proc}(c_i, c_i.k; P), \boxed{!(c_i : \oplus\{\ell : A_\ell\})}, \boxed{!(k \rhd \oplus\{\ell : A_\ell\})} \longrightarrow \mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+), \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \boxed{!(c_i^+ : A_k)}$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, c_i.k; P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash c_i.k; P :: (c_i : \Delta(c_i))$. By inversion of the $\oplus$R rule, $\Delta_{|\mathsf{fn}(P)} \vdash P :: (c : \Delta(c_i))$. By using the $\oplus$R and the id rule, we can type the message as follows: $c_i^+ : A_k \vdash c_i.k; c_i \leftarrow c_i^+ :: (c_i : \oplus\{\ell : A_\ell\})$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|\mathsf{fn}(P)} \vdash c_i.k; [c_i^+/c_i]P :: (c_i^+ : \oplus\{\ell : A_\ell\})$ and $\Delta'_{c_i^+} \vdash c_i.k; c_i \leftarrow c_i^+ :: (c_i : \oplus\{\ell : A_\ell\})$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$ and $H; \Delta' \Vdash \mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+)$.

Subcase $c_i \in H$

Because $k \rhd \oplus\{\ell : A_\ell\}_{\ell \in L}$, it must be the case that $k \in L$. Using that fact and that $c_i^+ : A_k$, $c_i : \oplus\{\ell : A_\ell\}_{\ell \in L}$ we can type the message as follows: $c_i^+ : A_k \vdash c_i.k; c_i \leftarrow c_i^+ :: (c_i : \oplus\{\ell : A_\ell\})$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+} \vdash c_i.k; c_i \leftarrow c_i^+ :: (c_i : \oplus\{\ell : A_\ell\})$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+)$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$

Case plus_s$_\mathsf{a}$

We have $\mathsf{proc}(c_i, c_i.k; P), (\boxed{!(c_i :\not\oplus\{\ell : A_\ell\})} \oplus \boxed{!(k \not\rhd \oplus\{\ell : A_\ell\})}) \longrightarrow \mathsf{alarm}(c_i)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, c_i.k; P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash c_i.k; P :: (c_i : \Delta(c_i)$. By the $\oplus R$ rule, $\Delta_{|\mathsf{fn}(P)} \vdash c_i.k; P :: (c_i : \oplus\{\ell : A_\ell\})$. By inversion of the $\oplus$R rule, we have that $k \in L$. If $c_i :\not\& \{\ell : A_\ell\}_{\ell \in L}$ then $\Delta_{|\mathsf{fn}(P)} \not\vdash c_i.k; P :: (c_i : \oplus\{\ell : A_\ell\})$. If $k \not\rhd \oplus\{\ell : A_\ell\}_{\ell \in L}$ then $\Delta_{|\mathsf{fn}(P)} \not\vdash c_i.k; P :: (c_i : \oplus\{\ell : A_\ell\})$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(c_i, c_i.k; P)$, it must

be the case that $c_i \in H$.

Subcase $c_i \in H$

Assume $\Delta_{|\mathsf{fn}(P)} \vdash c_i.k; P :: (c_i : \Delta(c_i))$. By the $\oplus$R rule, $\Delta_{|\mathsf{fn}(P)} \vdash c_i.k; P :: (c_i : \oplus\{\ell : A_\ell\})$. However, this contradicts the fact that $c_i :\not\in \oplus\{\ell : A_\ell\}_{\ell \in L}$. By inversion of the $\oplus$R rule, it must also be the case that $k \in L$, which contradicts the fact that $k \not\in \oplus\{\ell : A_\ell\}_{\ell \in L}$. Therefore, $\Delta_{|\mathsf{fn}(P)} \not\vdash c_i.k; P :: (c_i : \oplus\{\ell : A_\ell\})$.

Case plus_r

We have $\mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+), \mathsf{proc}(d, \mathsf{case}\ c_i\{\ell \Rightarrow Q_\ell\}_{\ell \in L}) \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q_k)$. By the configuration typing, $H; \Delta \Vdash \mathsf{msg}(c_i, c_i.k; c_i \leftarrow c_i^+)$ and $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{case}\ c_i\{\ell \Rightarrow Q_\ell\}_{\ell \in L})$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{case}\ c_i\{\ell \Rightarrow Q_\ell\}_{\ell \in L} :: (d : \Delta(d))$ and $\Delta_{|c_i^+} \vdash c_i.k; c_i \leftarrow c_i^+ :: (c_i : \oplus\{\ell : A_\ell\})$. By using the $\oplus$R and the id rule, we can type the message as follows: $c_i^+ : A_k \vdash c_i.k; c_i \leftarrow c_i^+ :: (c_i : \oplus\{\ell : A_\ell\})$. By inversion of the $\oplus$L typing rule, we have that $\Delta_{|\mathsf{fn}(Q)}, c_i : A\ell \vdash Q_\ell :: (d : \Delta(d))$. Let $\Delta' = [c_i^+ : A_k / c_i : A_k]\Delta$. We then have that $\Delta'_{|\mathsf{fn}(Q)}, c_i^+ : A_k \vdash [c_i^+/c_i]Q_k :: (d : \Delta'(d))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q_k)$.

Subcase $d \in H$

Let $\Delta' = \Delta$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q_k)$

A.4 Proof of lemma (one-step) for shared modality

Case $\mathsf{up}_\mathsf{L}^\mathsf{U}$_s

We have $\mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q)$, $!(c_U :\uparrow_L^U A_L^+)$ $\longrightarrow \mathsf{proc}(a, [c_L/x_L]Q)$, $\mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L)$, $!(c_L : A_L^+)$. By the configuration typing, $H; \Phi \Vdash \mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q)$.

Subcase $a \notin H$

By inversion of the configuration typing, $\Phi_{|x_L \cup \mathsf{fn}(Q) \cup c_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q :: (a : \Phi(a))$. By inversion of the $\uparrow$L rule, we have $\Phi_{\mathsf{fn}(Q)}, x_L : A_L^+ \vdash Q :: (a : \Phi(a))$. By using the $\uparrow$L and the id rule, we can type the message as follows: $c_U : \uparrow_L^U A_L^+ \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L :: (c_L : A_L^+)$. Let $\Phi' = [c_L : A_L^+ / x_L : A_L^+]\Phi$. We then have $\Phi'_{\mathsf{fn}(Q) \cup c_L} \vdash [c_L : A_L^+ / x_L : A_L^+]Q :: (a : \Phi'(a))$ and $\Phi'_{|c_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L :: (\Phi'(c_L)$. Therefore, $H; \Phi \Vdash \mathsf{proc}(a, [c_L/x_L]Q)$ and $H; \Phi \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L)$.

Subcase $a \in H$

Using the fact that $c_U : \uparrow_L^U : A_L^+$ and $c_L : \uparrow A_L^+$ we can type the message as follows: $c_U : \uparrow_L^U A_L^+ \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L :: (c_L : A_L^+)$. Let $\Phi' = [c_L : A_L^+ / x_L : A_L^+]\Phi$. We then have $\Phi'_{|c_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L :: (\Phi'(c_L)$. Therefore, $H; \Phi \Vdash$

$\mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L)$. Because $a \in H$, by the configuration typing, we have , $H; \Phi' \Vdash \mathsf{proc}(a, [c_L/x_L]Q)$.

Case $\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s_a}$

We have $\mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q)$, $!(c_u :\not\uparrow^U_L A^+_L)$ $\longrightarrow \mathsf{alarm}(a)$. By the configuration typing, $H; \Phi \Vdash \mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q)$.

Subcase $a \notin H$

By inversion of the configuration typing, $\Phi_{|x_L \cup \mathsf{fn}(Q) \cup c_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q :: (a : \Phi(a))$. By the $\uparrow$L rule, $c_U : \uparrow^U_L A^+_L \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q :: (\Phi(a))$. If $c_U :\not\uparrow^U_L A^+_L$ then $c_U : \uparrow^U_L \not\vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q :: (\Phi(a))$. This is a contradiction, and because $H; \Phi \Vdash \mathsf{proc}(a, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q)$, it must be the case that $a \in H$.

Subcase $a \in H$

Assume $\Phi_{|x_L \cup \mathsf{fn}(Q) \cup c_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q :: (a : \Phi(a))$. By the $\uparrow$L rule, $c_U : \uparrow^U_L \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q :: (\Phi(a))$. However, this contradicts the fact that $c_U :\not\uparrow^U_L A^+_L$. Therefore, $\Phi_{|x_L \cup \mathsf{fn}(Q)} \not\vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; Q :: (a : \Phi(a))$.

Case $\mathsf{up}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{r}$

We have $\mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L), !\mathsf{proc}(c_U, \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P) \longrightarrow \mathsf{proc}(c_L, [c_L/x_L]P)$. By the configuration typing, we have that $H; \Phi \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L)$ and $H; \Phi \Vdash \mathsf{proc}(c_U, \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P)$.

Subcase $c_U \notin H$

By inversion of the configuration typing, $\Phi_{|\mathsf{fn}(P) \cup x_L} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P) :: (c_U : \Phi(c_U)$ and $\Phi_{|c_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L) :: (c_L : \Phi(c_L)$. By using the $\uparrow$L and the id rule, we can type the message as follows: $c_U : \uparrow^U_L A^+_L \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L :: (c_L : A^+_L)$. By inversion of the $\uparrow$L typing rule, we have that $\Phi_{\mathsf{fn}(P)} \vdash P :: (x_L : A^+_L)$. Let $\Phi' = [c_L : A^+_L / x_L : A^+_L]\Phi$. We then have $\Phi'_{\mathsf{fn}(P)} \vdash [c_L/x_L]P :: (c_L : A^+_L)$. Therefore, $H; \Phi' \Vdash \mathsf{proc}(c_L, [c_L/x_L]P)$.

Subcase $c_U \in H$

Let $\Phi' = \Phi$. Because $c_U \in H$, by the configuration typing, we have $H; \Phi' \Vdash \mathsf{proc}(c_L, [c_L/x_L]P)$.

Case $\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s}$

We have $\mathsf{proc}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q)$, $!(c_L :\downarrow^U_L A^-_U)$ $\longrightarrow !\mathsf{proc}(c_U, [c_U/x_U]Q)$, $\mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U)$, $!(c_U : A^-_U)$ $(c_U\ fresh)$. By the configuration typing, we have $H; \Phi \Vdash \mathsf{proc}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q)$.

Subcase $c_L \notin H$

By inversion of the configuration typing, $\Phi_{|x_U \cup \mathsf{fn}(Q)} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q :: (c_L : \Phi(c_L))$. By inversion of the $\downarrow$ R rule, we have $\Phi_{|\mathsf{fn}(Q)} \vdash Q :: (x_U : A^-_U)$. By using the $\downarrow$R and the $\mathsf{id_U}$ rule, we can type the message as follows: $x_U : A^-_U \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L :$

$\downarrow_L^U A_U^-$). Let $\Phi' = [c_U : A_U^-/x_U^-]\Phi$. We then have $\Phi'_{|\mathsf{fn}(Q)} \vdash [c_U/x_U]Q :: (c_U : \Phi'(c_U))$ and $\Phi'_{|x_U} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L : \Phi'(c_L))$. Therefore, $H; \Phi' \Vdash \mathsf{proc}(c_U, [c_U/x_U]Q)$ and $H; \Phi' \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U)$.

Subcase $c_L \in H$

Using the fact that $c_L : \downarrow_L^U A_U^-$ and $c_U : A_U^-$ we can type the message as follows:$x_U : A_U^- \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L : \downarrow_L^U A_U^-)$. Let $\Phi' = [c_U : A_U^-/x_U^-]\Phi$. We then have $\Phi'_{|x_U} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L : \Phi'(c_L))$. Therefore, $H; \Phi' \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U)$. Because $c_L \in H$, by the configuration typing, we have $H; \Phi' \Vdash \mathsf{proc}(c_U, [c_U/x_U]Q)$.

Case $\mathsf{down}_{\mathsf{L}}^{\mathsf{U}}\_\mathsf{s_a}$

We have $\mathsf{proc}(c_L; \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q), !(c_L :\not\downarrow_L^U A_U^-) \longrightarrow \mathsf{alarm}(c_L)$. By the configuration typing, $H; \Phi \Vdash \mathsf{proc}(c_L; \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q)$.

Subcase $c_L \notin H$

By inversion of the configuration typing, $\Phi_{|x_U \cup \mathsf{fn}(Q)} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q :: (c_L : \Phi(c_L))$. By the $\downarrow$R rule, $\Phi_{|x_U \cup \mathsf{fn}(Q)} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q :: (c_L : \downarrow_L^U A_U^-)$. If $c_L :\not\downarrow_L^U A_U^-$ then $\Phi_{|x_U \cup \mathsf{fn}(Q)} \not\vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q :: (c_L : \downarrow_L^U A_U^-)$. This is a contradiction, and because $H; \Phi \Vdash \mathsf{proc}(c_L; \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q)$, it must be the case that $c_L \in H$.

Subcase $c_L \in H$

Assume $\Phi_{|x_U \cup \mathsf{fn}(Q)} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q :: (c_L : \Phi(c_L))$. However, this contradicts the fact that $c_U :\not\uparrow_L^U A_L^+$. Therefore, $\Phi_{|x_U \cup \mathsf{fn}(Q)} \not\vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q :: (c_L : \Phi(c_L))$.

Case $\mathsf{down}_{\mathsf{L}}^{\mathsf{U}}\_\mathsf{r}$

We have $\mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U), \mathsf{proc}(a, \mathsf{shift}\ x_U \leftarrow \mathsf{recv}\ c_L; P) \longrightarrow \mathsf{proc}(a, [c_U/x_U]P)$. By the configuration typing we have that $H; \Phi \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U)$ and $H; \Phi \Vdash \mathsf{proc}(a, \mathsf{shift}\ x_U \leftarrow \mathsf{recv}\ c_L; P)$.

Subcase $a \notin H$

By inversion of the configuration typing, $\Phi_{|\mathsf{fn}(P) \cup c_L \cup x_U} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{recv}\ c_L; P :: (a : \Phi(a))$ and $\Phi \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L : \Phi(c_L))$. By using the $\downarrow$R and the $\mathsf{id_U}$ rule, we can type the message as follows: $x_U : A_U^- \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L : \downarrow_L^U A_U^-)$. By inversion of the $\downarrow$L typing rule we have that $\Phi_{|\mathsf{fn}(Q) \cup x_U} \vdash Q :: (a : \Phi(a))$. Let $\Phi' = [c_U : A_U^-/x_U : A_U^-]\Phi$. We then have that $\Phi_{|\mathsf{fn}(Q) \cup c_U} \vdash [c_U/x_U]Q :: (a : \Phi(a))$. Therefore, $H; \Phi' \Vdash \mathsf{proc}(a, [c_U/x_U]P)$.

Subcase $a \in H$

Let $\Phi' = \Phi$. Because $a \in H$, by the configuration typing, we have $H; \Phi' \Vdash \mathsf{proc}(a, [c_U/x_U]P)$.

Case $\mathsf{id_U}$

We have $\mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L), !\mathsf{proc}(a_U, a_U \leftarrow b_U), !(a_U : A), !(b_U : A)$

$\longrightarrow \mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ b_U; a_L \leftarrow x_L)$. By the configuration typing we have that $H; \Phi \Vdash \mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L)$ and $H; \Phi \Vdash !\mathsf{proc}(a_U, a_U \leftarrow b_U)$.

Subcase $a_U \notin H$

By inversion of the configuration typing, $\Phi_{|a_U \cup x_L} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L :: (a_L : \Phi(a_L))$ and $\Phi_{|b_U} \vdash a_U \leftarrow b_U :: (a_U : \Phi(a_U))$. Using the $\uparrow$L rule and the id rule, we can type the message as follows: $a_U : \uparrow^U_L : A^+_L \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L :: (a_L : A^+_L)$. By the $\mathsf{id_U}$ typing rule, we have that $b_U : \uparrow^U_L : A^+_L \vdash a_U \leftarrow b_U :: (a_U : \uparrow^U_L : A^+_L)$. Let $\Phi' = [b_U : \uparrow^U_L : A^+_L / a_U : \uparrow^U_L : A^+_L]\Phi$. We then have that $\Phi_{|b_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L :: (a_L : A^+_L)$. Therefore, $H; \Phi' \Vdash \mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ b_U; a_L \leftarrow x_L)$.

Subcase $a_U \in H$

By inversion of the configuration typing, $\Phi_{|a_U \cup x_L} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L :: (a_L : \Phi(a_L))$. Using the $\uparrow$L rule and the id rule, we can type the message as follows: $a_U : \uparrow^U_L : A^+_L \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L :: (a_L : A^+_L)$. Because $a_U : \uparrow^U_L : A^+_L$ and $b_U : \uparrow^U_L : A^+_L$, let $\Phi' = [b_U : \uparrow^U_L : A^+_L / a_U : \uparrow^U_L : A^+_L]\Phi$.We then have that $\Phi_{|b_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ a_U; a_L \leftarrow x_L :: (a_L : A^+_L)$. Therefore, $H; \Phi' \Vdash \mathsf{msg}(a_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ b_U; a_L \leftarrow x_L)$.

Case $\mathsf{cut_U}$

We have $\mathsf{proc}(c, x_U : A \leftarrow P; Q)$, $!(\mathsf{typecheck}(P :: x_U : A)) \longrightarrow \mathsf{proc}(c, [a_U/x_U]Q), !\mathsf{proc}(a_U, [a_U/x_U]P)$. By the configuration typing, $H; \Phi \Vdash \mathsf{proc}(c, x_U : A \leftarrow P; Q)$.

Subcase $c \notin H$

By inversion of the configuration typing, $\Phi_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$. By inversion of the $\mathsf{cut_U}$ typing rule, $\Phi_{|CH'} \vdash P :: (x : \Phi(x))$ and $\Phi_{|CH} \vdash Q :: (c : \Phi(c))$. By the typing, we know that $\Phi_{|CH'}$ must be an unrestricted context and that $\mathsf{fn}(P) \subseteq CH'$. Let $CH' = \mathsf{fn}(P) \cup u_1$ where u_1 is unrestricted. Because channel c could be linear or unrestricted, we have $CH = \mathsf{fn}(Q) \cup u_2$ where u_2 is unrestricted. Let $\Phi' = [a_U : A / x_U : A]\Phi$. Let $a_U : A$, so by the substitution lemma we get $\Phi'_{|\mathsf{fn}(P)} \vdash [a_U : A / x_U : A]P :: (a_U : \Phi'(a_0)))$ and $\Phi'_{|\mathsf{fn}(Q)} \vdash [a_U : A / x : A]Q :: (c : \Phi'(c))$. Therefore, $H; \Phi' \Vdash \mathsf{proc}(c, [a_U/x_U]Q)$ and $H; \Phi' \Vdash \mathsf{proc}(a_U, [a_U/x_U]P)$.

Subcase $c \in H$

Using the fact that $P :: (x : A)$ we have $\Phi_{|CH} \vdash P :: (x : \Phi(a))$. Because x_U is unrestricted, we have $CH = \mathsf{fn}(P) \cup u_1$ where u_1 is also unrestricted. Let $\Phi' = [a_U : A / x_U : A]\Phi$. We then have $\Phi'_{|\mathsf{fn}(P)} \vdash [a_U/x_U]P :: (x : \Phi(a))$ which gives us $H; \Phi' \Vdash \mathsf{proc}(a_U, [a_U/x_U]P)$. Because $c \in H$, by the configuration typing, we have that $H; \Phi' \Vdash \mathsf{proc}(c, [a_U/x_U]Q)$.

Case $\mathsf{cut_U a}$

We have $\mathsf{proc}(c, x_U : A \leftarrow P; Q)$, $!(\mathsf{typecheck}(P :: x :\not A)) \longrightarrow \mathsf{alarm}(c)$. By the configuration typing, $H; \Phi \Vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$.

Subcase $c \notin H$.

By inversion of the configuration typing, $\Phi_{|\mathsf{fn}(P)\cup\mathsf{fn}(Q)} \vdash x : A \leftarrow P; Q :: (c : \Phi(c))$. By inversion of the cut typing rule, $\Phi_{|CH'} \vdash P :: (x : \Phi(x)))$ and $\Phi_{|CH} \vdash Q :: (c : \Phi(c))$. By the typing, we know that $\Phi_{|CH'}$ must be an unrestricted context and that $\mathsf{fn}(P) \subseteq CH'$. Let $CH' = \mathsf{fn}(P) \cup u_1$ where u_1 is unrestricted. If $\mathsf{typecheck}(P :: x :/ A)$ then $\Phi_{|\mathsf{fn}(P)} \not\vdash P :: (x : \Phi(x)))$. This means that $\Phi_{|\mathsf{fn}(P)\cup u_1} \not\vdash P :: (x : \Phi(x))$ and $\Phi_{|CH'} \not\vdash P :: (x : \Phi(x))$. Therefore, $\Phi_{|\mathsf{fn}(P)\cup\mathsf{fn}(Q)} \not\vdash x : A \leftarrow P; Q :: (c : \Phi(c))$. This is a contradiction, and because $H; \Phi \Vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$ it must be the case that $c \in H$.

Subcase $c \in H$

We assume $\Phi_{|\mathsf{fn}(P)\cup\mathsf{fn}(Q)} \vdash x : A \leftarrow P; Q :: (c : \Phi(c))$. By inversion of the cut typing rule, $\Phi_{|CH'} \vdash P :: (x : \Phi(x)))$ and $\Phi_{|CH} \vdash Q :: (c : \Phi(c))$. By the typing, we know that $\Phi_{|CH'}$ must be an unrestricted context and that $\mathsf{fn}(P) \subseteq CH'$. Let $CH' = \mathsf{fn}(P) \cup u_1$ where u_1 is unrestricted. If $\mathsf{typecheck}(P :: x :/ A)$ then $\Phi_{|\mathsf{fn}(P)} \not\vdash P :: (x : \Phi(x)))$. This means that $\Phi_{|\mathsf{fn}(P)\cup u_1} \not\vdash P :: (x : \Phi(x))$ and $\Phi_{|CH'} \not\vdash P :: (x : \Phi(x))$. Therefore, $\Phi_{|\mathsf{fn}(P)\cup\mathsf{fn}(Q)} \not\vdash x : A \leftarrow P; Q :: (c : \Phi(c))$. This is a contradiction and $\Phi_{|\mathsf{fn}(P)\cup\mathsf{fn}(Q)} \not\vdash x : A \leftarrow P; Q :: (c : \Phi(c))$.

A.5 Unverified Spawn Configuration Inversion Lemma

Lemma 20 (Configuration-Inversion)**.**

1. *If* $G; H; \Lambda \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$ *where* $\mathcal{C}_2 = \mathsf{proc}(c, P)$ *then* $G; H; \Lambda \Vdash \mathcal{C}_1$ *and* $G; H; \Lambda \Vdash \mathsf{proc}(c, P)$.
2. *If* $G; H; \Lambda \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ *where* $\mathcal{C}_2 = \mathsf{msg}(c, P)$ *and* $\mathcal{C}_3 = \mathsf{proc}(d, Q)$ *then* $G; H; \Lambda \Vdash \mathcal{C}_1$ *and* $G; H; \Lambda \Vdash \mathsf{msg}(c, P)$ *and* $G; H; \Lambda \Vdash \mathsf{proc}(d, Q)$.
3. *If* $G; H; \Lambda \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ *where* $\mathcal{C}_2 = \mathsf{proc}(c, P)$ *and* $\mathcal{C}_3 = \mathsf{msg}(d, Q)$ *then* $G; H; \Lambda \Vdash \mathcal{C}_1$ *and* $G; H; \Lambda \Vdash \mathsf{proc}(c, P)$ *and* $G; H; \Lambda \Vdash \mathsf{msg}(d, Q)$.
4. *If* $G; H; \Lambda \Vdash \mathsf{proc}(c, P)$ *and* $G \not\vdash c : \mathsf{havoc}$*, then* $\Lambda_{|\mathsf{fn}P} \vdash P :: (c : \Lambda(c))$.
5. *If* $G; H; \Lambda \Vdash \mathsf{msg}(c, P)$ *and* $G \not\vdash c : \mathsf{havoc}$*, then* $\Lambda_{|\mathsf{fn}(P)} \vdash P :: (c : \Lambda(c))$.

Proof. By examining the configuration typing rules. □

A.6 Proof of lemma (one-step) for Unverified-Spawn Semantics

Case cut

We have $\mathsf{proc}(c, x : A \leftarrow P; Q) \longrightarrow \mathsf{proc}(c, [a_0/x]Q), \mathsf{proc}(a_0, [a_0/x]P), !(G(c \rightarrow_{sp} a_0))$. By the configuration typing, $G; H; \Delta \Vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$.

Subcase $c \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)\cup\mathsf{fn}(Q)} \vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$. By inversion of the cut typing rule, $\Delta_{|CH'} \vdash P :: (x : \Delta(x))$ and $\Delta_{|CH} \vdash Q :: (c : \Delta(c))$. By the

Freename lemma, we know that $CH' = \mathsf{fn}(P)$ and $CH = \mathsf{fn}(Q)$. Let $\Delta' = [a_0 : A/x : A]\Delta$. Let $a_0 : A$, so by the substitution lemma we get $\Delta'_{|\mathsf{fn}(P)} \vdash [a_0 : A/x : A]P :: (a_0 : \Delta'(a_0)))$ and $\Delta'_{|\mathsf{fn}(Q)} \vdash [a_0 : A/x : A]Q :: (c : \Delta'(c))$. Therefore, $G; H; \Delta' \Vdash \mathsf{proc}(c, [a_0/x]Q)$ and $G; H; \Delta' \Vdash \mathsf{proc}(a_0, [a_0/x]P)$.

Subcase $c \in H$

Because $c \in H$, we have that $G \vdash c : \mathsf{havoc}$. Therefore, by the configuration typing, we have that $G; H; \Delta' \Vdash \mathsf{proc}(c, [a_0/x]Q)$. Because $c \in H$ and $G(c \rightarrow_{sp} a_0)$, we have that $G \vdash a_0 : \mathsf{havoc}$. Therefore, by the configuration typing, we have that $G; H; \Delta' \Vdash \mathsf{proc}(a_0, [a_0/x]P)$.

Case cut_U

We have $\mathsf{proc}(c, x_U : A \leftarrow P; Q) \longrightarrow \mathsf{proc}(c, [a_U/x_U]Q), !\mathsf{proc}(a_U, [a_U/x_U]P)$, $!(G(c \rightarrow_{sp} a_U))$. By the configuration typing, $G; H; \Phi \Vdash \mathsf{proc}(c, x_U : A \leftarrow P; Q)$.

Subcase $c \notin H$

By inversion of the configuration typing, $\Phi_{|\mathsf{fn}(P) \cup \mathsf{fn}(Q)} \vdash \mathsf{proc}(c, x : A \leftarrow P; Q)$. By inversion of the cut_U typing rule, $\Phi_{|CH'} \vdash P :: (x : \Phi(x))$ and $\Phi_{|CH} \vdash Q :: (c : \Phi(c))$. By the typing, we know that $\Phi_{|CH'}$ must be an unrestricted context and that $\mathsf{fn}(P) \subseteq CH'$. Let $CH' = \mathsf{fn}(P) \cup u_1$ where u_1 is unrestricted. Because channel c could be linear or unrestricted, we have $CH = \mathsf{fn}(Q) \cup u_2$ where u_2 is unrestricted. Let $\Phi' = [a_U : A/x_U : A]\Phi$. Let $a_U : A$, so by the substitution lemma we get $\Phi'_{|\mathsf{fn}(P)} \vdash [a_U : A/x_U : A]P :: (a_U : \Phi'(a_0)))$ and $\Phi'_{|\mathsf{fn}(Q)} \vdash [a_U : A/x : A]Q :: (c : \Phi'(c))$. Therefore, $G; H; \Phi' \Vdash \mathsf{proc}(c, [a_U/x_U]Q)$ and $G; H; \Phi' \Vdash \mathsf{proc}(a_U, [a_U/x_U]P)$.

Subcase $c \in H$

Because $c \in H$, we have that $G \vdash c : \mathsf{havoc}$. Therefore, by the configuration typing, we have that $G; H; \Delta' \Vdash \mathsf{proc}(c, [a_0/x_U]Q)$. Because $c \in H$ and $G(c \rightarrow_{sp} a_U)$, we have that $G \vdash a_U : \mathsf{havoc}$. Therefore, by the configuration typing, we have that $G; H; \Delta' \Vdash \mathsf{proc}(a_U, [a_U/x_U]P)$.

Case $\mathsf{up}^\mathsf{U}_\mathsf{L}\_\mathsf{r}$

We have $\mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L), !\mathsf{proc}(c_U, \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P) \longrightarrow \mathsf{proc}(c_L, [c_L/x_L]P), !(G(c_U \rightarrow_{sp} c_L))$. By the configuration typing, we have that $G; H; \Phi \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L)$ and $G; H; \Phi \Vdash \mathsf{proc}(c_U, \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P)$.

Subcase $c_U \notin H$

By inversion of the configuration typing, $\Phi_{|\mathsf{fn}(P) \cup x_L} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{recv}\ c_U; P) :: (c_U : \Phi(c_U)$ and $\Phi_{|c_U} \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L) :: (c_L : \Phi(c_L)$. By using the $\uparrow$L and the id rule, we can type the message as follows: $c_U : \uparrow^U_L A^+_L \vdash \mathsf{shift}\ x_L \leftarrow \mathsf{send}\ c_U; c_L \leftarrow x_L :: (c_L : A^+_L)$. By inversion of the $\uparrow$L typing rule, we have that $\Phi_{\mathsf{fn}(P)} \vdash P :: (x_L : A^+_L)$. Let $\Phi' = [c_L : A^+_L/x_L : A^+_L]\Phi$. We then have $\Phi'_{\mathsf{fn}(P)} \vdash [c_L/x_L]P :: (c_L : A^+_L)$. Therefore, $G; H; \Phi' \Vdash \mathsf{proc}(c_L, [c_L/x_L]P)$.

Subcase $c_U \in H$

Because $c_U \in H$ and $G(c_U \rightarrow_{sp} c_k)$, we have that $G \vdash c_k : \mathsf{havoc}$. Therefore, by the config-

uration typing, we have that $G; H; \Delta' \Vdash \mathsf{proc}(c_k, [c_k/x_k]P)$.

Case $\mathsf{down}^{\mathsf{U}}_{\mathsf{L}}\_\mathsf{s}$

We have $\mathsf{proc}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q)$, $!(c_L : \downarrow^U_L A^-) \longrightarrow !\mathsf{proc}(c_U, [c_U/x_U]Q)$, $\mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U)$, $!(G(c_L \rightarrow_{sp} c_U))$. By the configuration typing, we have
$G; H; \Phi \Vdash \mathsf{proc}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q)$.

Subcase $c_L \notin H$

By inversion of the configuration typing, $\Phi_{|x_U \cup \mathsf{fn}(Q)} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; Q :: (c_L : \Phi(c_L))$. By inversion of the $\downarrow$ R rule, we have $\Phi_{|\mathsf{fn}(Q)} \vdash Q :: (x_U : A^-_U)$. By using the $\downarrow$R and the $\mathsf{id_U}$ rule, we can type the message as follows: $x_U : A^-_U \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L : \downarrow^U_L A^-_U)$. Let $\Phi' = [c_U : A^-_U/x^-_U]\Phi$. We then have $\Phi'_{|\mathsf{fn}(Q)} \vdash [c_U/x_U]Q :: (c_U : \Phi'(c_U))$ and $\Phi'_{|x_U} \vdash \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U :: (c_L : \Phi'(c_L))$. Therefore, $H; \Phi' \Vdash \mathsf{proc}(c_U, [c_U/x_U]Q)$ and $H; \Phi' \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U)$.

Subcase $c_L \in H$

Because $c_L \in H$ and $G(c_L \rightarrow_{sp} c_U)$, we have that $G \vdash c_L : \mathsf{havoc}$. Therefore, by the configuration typing, we have that $G; H; \Delta' \Vdash \mathsf{proc}(c_U, [c_U/x_U]Q)$. Because $c_L \in H$, we have that $G \vdash c_L : \mathsf{havoc}$. Therefore, by the configuration typing, $G; H; \Phi \Vdash \mathsf{msg}(c_L, \mathsf{shift}\ x_U \leftarrow \mathsf{send}\ c_L; c_U \leftarrow x_U)$.

Appendix B

Refinement Proof Cases

B.1 Monitors are Well Typed

Case 1.

$$[\![\langle \mathbf{1} \Leftarrow \mathbf{1} \rangle^{\rho}]\!]_{a,b} = \mathsf{wait}\ b; \mathsf{close}\ a$$

This process is well-typed with respect to both S and T by applying the 1L and 1R rules.

Case $\multimap$.

$$[\![\langle A_1 \multimap A_2 \Leftarrow B_1 \multimap B_2 \rangle^{\rho}]\!]_{a,b} =$$
$$x \leftarrow \mathsf{recv}\ a\ ;\ y \leftarrow [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]_{y,x} \leftarrow x\ ;\ \mathsf{send}\ b\ y;\ ;\ [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]_{a,b}$$

(1) By I.H, we know that $E' = \Psi\ ;\ x : A_1 \vdash_T [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]^{\Psi}_{y,x} :: (y : B_1)$ and $E'' = \Psi\ ;\ b : B_2 \vdash_T [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A_2)$. Apply the $\multimap$L rule to E'' to typecheck the send. Then apply the $\multimap$R rule to typecheck the receive. Therefore, this process is well-typed with respect to T.

(2) We have that $B_1 \multimap B_2 \leq A_1 \multimap A_2$. By subtyping, this means that $A_1 \leq B_1$ and $B_2 \leq A_2$. By I.H., we know that $E' = \Psi\ ;\ x : A_1 \vdash_S [\![\langle B_1 \Leftarrow A_1 \rangle^{\rho}]\!]^{\Psi}_{y,x} :: (y : B_1)$ and $E'' = \Psi\ ;\ b : B_2 \vdash_S [\![\langle A_2 \Leftarrow B_2 \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A_2)$. Apply the $\multimap$L rule to E'' to typecheck the send. Then apply the $\multimap$R rule to typecheck the receive. Therefore, this process is well-typed with respect to S.

Case $\forall$.

$$[\![\langle \forall\{n : \tau \mid e\}.\, A \Leftarrow \forall\{n : \tau' \mid e'\}.\, B \rangle^{\rho}]\!]_{a,b} =$$
$$n \leftarrow \mathsf{recv}\ a; ;\ \mathsf{assert}\ \rho\, e'(x)\ \ ;\ \mathsf{send}\ b\ x; [\![\langle A \Leftarrow B \rangle^{\rho}]\!]_{a,b}$$

(1) By I.H, we know that $E' = \Psi\ ;\ b : B \vdash_T [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A)$. Apply the $\forall$L rule to E' to typecheck the send. Then, apply the assert rule to typecheck the assertion. Finally, apply the $\forall$R rule to typecheck the receive. Therefore, this process is well-typed with respect to T.

(2) Because $\forall\{n : \tau' \mid e'\}.B \leq \forall\{n : \tau' \mid e\}.A$ we know that $B \leq A$. By I.H., we know that $E' = \Psi\ ;\ b : B \vdash_S [\![\langle A \Leftarrow B \rangle^{\rho}]\!]^{\Psi}_{a,b} :: (a : A)$. Apply the $\forall$L rule to E' to typecheck the

send. We now need to show that we can typecheck the assertion using the assert_strong rule. That is, we need to show that $e'(x)$ is true. From the subtyping $refine$ rule, we see that for $x : \tau, [x/y]e \mapsto^*$ true implies $x : \tau, [x/y]e' \mapsto^*$ true. Therefore, $e'(x)$ is true and will not abort. We then apply the assert_strong rule to typecheck the assertion. Finally, apply the $\forall$R rule to typecheck the receive. Therefore, this process is well-typed with respect to S.

Case $\oplus$.

$$[\![\langle \oplus\{\ell : A_\ell\}_{\ell \in I} \Leftarrow \oplus\{\ell : B_\ell\}_{\ell \in J}\rangle^\rho]\!]_{a,b} = \mathsf{case}\ b\ (\ell \Rightarrow Q_\ell)_{\ell \in I}$$

$$\mathsf{where}\ \forall \ell, \ell \in I \cap J,\ a.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]_{a,b} = Q_\ell \mathsf{and}\ \forall \ell, \ell \in J \wedge \ell \notin I,\ Q_\ell = \mathsf{abort}\ \rho$$

(1) By I.H., we know that $E' = \Psi\ ;\ b : B_\ell \vdash_T [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]^\Psi_{a,b} :: (a : A_\ell)$. Apply the $\oplus$R rule to E' to typecheck sending the label $a.\ell$. Then apply the $\oplus$L rule to typecheck the case. Thererefore, this process is well-typed with respect to T.

(2) Because $\oplus\{\ell : B_\ell\}_{\ell \in J} \leq \oplus\{\ell : A_\ell\}_{\ell \in I}$ we know that $B_k \leq A_k$. By I.H., we know that $E' = \Psi\ ;\ b : B_\ell \vdash_S [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]^\Psi_{a,b} :: (a : A_\ell)$. Apply the $\oplus$R rule to E' to typecheck sending the label $a.\ell$. Then apply the $\oplus$L rule to typecheck the case. From the subtyping, we also know that $J \subseteq I$. This means that there does not exist an $\ell \in J \wedge l \notin I$, and no abort branch will be generated. Thererefore, this process is well-typed with respect to S.

B.2 Casts are Transparent

Case 1.

$$[\![\langle \mathbf{1} \Leftarrow \mathbf{1}\rangle^\rho]\!]_{a,b} = \mathsf{wait}\ b; \mathsf{close}\ a$$

Apply the 1R rule and then the 1L rule to get $(b : \mathbf{1}); \Psi; \cdot; \cdot \vdash \mathsf{wait}\ b; \mathsf{close}\ b :: [](a : \mathbf{1})$.

Case $\otimes$.

$$[\![\langle A_1 \otimes A_2 \Leftarrow B_1 \otimes B_2\rangle^\rho]\!]_{a,b} =$$

$$x \leftarrow \mathsf{recv}\ b\ ;\ y \leftarrow [\![\langle A_1 \Leftarrow B_1\rangle^\rho]\!]_{y,x} \leftarrow x\ ;\ \mathsf{send}\ a\ y\ ;\ [\![\langle A_2 \Leftarrow B_2\rangle^\rho]\!]_{a,b}$$

We first apply the $\otimes$L rule. We then apply the cut_{3++} rule which produces two premises. The first premise is $P = [\![\langle A_1 \Leftarrow B_1\rangle^\rho]\!]_{y,x}$. By induction, we have a derivation for P. The second premise is $\mathsf{send}\ a\ y\ ;\ [\![\langle A_2 \Leftarrow B_2\rangle^\rho]\!]_{a,b}$. We then apply the $\otimes$R rule. Let $Q = [\![\langle A_2 \Leftarrow B_2\rangle^\rho]\!]_{a,b}$. By induction, we have a derivation for Q and we are done.

Case $\exists$.

$$[\![\langle \exists\{n : \tau \mid e\}.\ A \Leftarrow \exists\{n : \tau' \mid e'\}.\ B\rangle^\rho]\!]_{a,b} = n \leftarrow \mathsf{recv}\ b\ ;\ \mathsf{assert}\ \rho\, e(x)\ \ ;\ \mathsf{send}\ a\ x; [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}$$

We first apply the $\exists$L rule. We then apply the $\exists$R rule, which produces a premise $P = [\![\langle A \Leftarrow B\rangle^\rho]\!]_{a,b}$. By induction, we have a derivation for P and we are done.

Case $\&$.

$$[\![\langle \&\{\ell : A_\ell\}_{\ell\in I} \Leftarrow \&\{\ell : B_\ell\}_{\ell\in J}\rangle^\rho]\!]_{a,b} = \mathsf{case}\ a\ (\ell \Rightarrow Q_\ell)_{\ell\in I}$$

$$\mathsf{where}\ \forall \ell, \ell \in I \cap J,\ b.\ell\ ;\ [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]_{a,b} = Q_\ell\ \mathsf{and}\ \forall \ell, \ell \in I \wedge \ell \notin J,\ Q_\ell = \mathsf{abort}\ \rho$$

We first apply the $\&$L rule. We then apply the $\&$R rule, which produces a premise $P = [\![\langle A_\ell \Leftarrow B_\ell\rangle^\rho]\!]_{a,b}$. By induction, we have a derivation for P and we are done.

B.3 Subtype Inversion

Lemma 21 (Subtype-Inversion)**.**

1. *If* $\tau \leq 1$ *then* $\tau = 1$.
2. *If* $1 \leq \tau$ *then* $\tau = 1$.
3. *If* $\tau \leq A \otimes B$*, then* $\tau = C \otimes D$ *for some types* C *and* D.
4. *If* $A \otimes B \leq \tau$*, then* $\tau = C \otimes D$ *for some types* C *and* D.
5. *If* $A \otimes B \leq A' \otimes B'$ *then* $A \leq A'$ *and* $B \leq B'$.
6. *If* $\tau \leq A \multimap B$*, then* $\tau = C \multimap D$ *for some types* C *and* D.
7. *If* $A \multimap B \leq \tau$*, then* $\tau = C \multimap D$ *for some types* C *and* D.
8. *If* $A \multimap B \leq A' \multimap B'$ *then* $A' \leq A$ *and* $B \leq B'$.
9. *If* $\tau \leq \oplus\{lab_k : A_k\}_{k\in J}$ *then* $\tau = \oplus\{lab_m : A_m\}_{m\in I}$ *for some* m *and* I *and type* A_m.
10. *If* $\oplus\{lab_k : A_k\}_{k\in J} \leq \tau$ *then* $\tau = \oplus\{lab_m : A_m\}_{m\in I}$ *for some* m *and* I *and type* A_m.
11. *If* $\oplus\{lab_k : A_k\}_{k\in J} \leq \oplus\{lab_k : A'_k\}_{k\in I}$ *then* $A_k \leq A'_k$ *for* $k \in J$ *and* $J \subseteq I$.
12. *If* $\tau \leq \&\{lab_k : A_k\}_{k\in J}$ *then* $\tau = \&\{lab_m : A_m\}_{m\in I}$ *for some* m *and* I *and type* A_m.
13. *If* $\&\{lab_k : A_k\}_{k\in J} \leq \tau$ *then* $\tau = \&\{lab_m : A_m\}_{m\in I}$ *for some* m *and* I *and type* A_m.
14. *If* $\&\{lab_k : A_k\}_{k\in J} \leq \&\{lab_k : A'_k\}_{k\in I}$ *then* $A_k \leq A'_k$ *for* $k \in J$ *and* $I \subseteq J$.
15. *If* $\tau \leq \downarrow A$ *then* $\tau = \downarrow B$ *for some type* B.
16. *If* $\downarrow A \leq \tau$ *then* $\tau = \downarrow B$ *for some type* B.
17. *If* $\downarrow A \leq \downarrow B$ *then* $A \leq B$.
18. *If* $\tau \leq \uparrow A$ *then* $\tau = \uparrow B$ *for some type* B.
19. *If* $\uparrow A \leq \tau$ *then* $\tau = \uparrow B$ *for some type* B.
20. *If* $\uparrow A \leq \uparrow B$ *then* $A \leq B$.
21. *If* $\tau \leq \exists n : \tau_1.A$ *then* $\tau = \exists n : \tau_2.B$ *for some types* τ_2 *and* B.
22. *If* $\exists n : \tau_1.A \leq \tau$ *then* $\tau = \exists n : \tau_2.B$ *for some types* τ_2 *and* B.
23. *If* $\exists n : \tau_1.A \leq \exists n : \tau_2.B$ *then* $A \leq B$ *and* $\tau_1 \leq \tau_2$.
24. *If* $\tau \leq \forall n : \tau_1.A$ *then* $\tau = \forall n : \tau_2.B$ *for some types* τ_2 *and* B.
25. *If* $\forall n : \tau_1.A \leq \tau$ *then* $\tau = \forall n : \tau_2.B$ *for some types* τ_2 *and* B.
26. *If* $\forall n : \tau_1.A \leq \forall n : \tau_2.B$ *then* $A \leq B$ *and* $\tau_2 \leq \tau_1$.

Proof. Cases 1,2: by examining the subtyping rule for 1.
Cases 3,4,5: by examining the subtyping rule for $\otimes$.
Cases 6,7,8: by examining the subtyping rule for $\multimap$.
Cases 9,10,11: by examining the subtyping rule for $\oplus$.
Cases 12,13,14: by examining the subtyping rule for $\&$.
Cases 15,16,17: by examining the subtyping rule for $\downarrow$.
Cases 18,19,20: by examining the subtyping rule for $\uparrow$.
Cases 21,22,23: by examining the subtyping rule for $\exists$.
Cases 24,25,26: by examining the subtyping rule for $\forall$.

□

B.4 Subtype Substitution

Case id_cast.

$$\frac{A \sim A'}{\Psi\ ;\ b : A' \vdash a \leftarrow \langle A \Leftarrow A' \rangle^{\rho}\ b :: (a : A)}\ \mathsf{id\_cast}$$

Subcase left. We have $A'' \leq A'$. Let $g : A''$. When we perform the substitution $[g : A''/b : A']$ we update the cast $\langle A \Leftarrow A' \rangle^{\rho}$ to $\langle A \Leftarrow A'' \rangle^{\rho}$. We then get $\Psi; g : A'' \vdash a \leftarrow \langle A \Leftarrow A'' \rangle^{\rho} g :: (a : A)$ which matches our substitution rules.

Subcase right. We have $A \leq A''$. Let $g : A''$. When we perform the substitution $[g : A''/b : A']$ we update the cast $\langle A \Leftarrow A' \rangle^{\rho}$ to $\langle A'' \Leftarrow A' \rangle^{\rho}$. We then get $\Psi; b : A' \vdash g \leftarrow \langle A'' \Leftarrow A' \rangle^{\rho} b :: (g : A'')$ which matches our substitution rules.

Case val_cast.

$$\frac{\Psi \vdash v : \tau' \quad E' = \Psi, x : \tau; \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'}{\Psi; \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^{\rho}\ v; Q :: (c : C)}\ \mathsf{val\_cast}$$

We have $C \leq C'$. Apply I.H. to E' to get E''. For some fresh $g : C'$ we have $E'' = \Psi, x : \tau; \Delta \vdash [g : C'/c : C]Q :: (g : C')$. Now apply the val_cast rule to E'' to get $\Psi; \Delta \vdash x \leftarrow \langle \tau \Leftarrow \tau' \rangle^{\rho} v; [g : C'/c : C]Q :: (g : C')$ which matches our substitution rules.

Case $\oplus$L.

$$E = \frac{E' = \Psi; \Delta, c : A_\ell \vdash Q_\ell :: (d : D) \quad \text{for every } \ell \in L}{\Psi; \Delta, c : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \mathsf{case}\ c\ (\ell \Rightarrow Q_\ell)_{\ell \in L} :: (d : D)}\ \oplus L$$

We have $\oplus\{\ell : A'_\ell\}_{\ell \in J} \leq \oplus\{\ell : A_\ell\}_{\ell \in L}$. By subtyping, we have $A'_\ell \leq A_\ell$ for $\ell \in J, J \subseteq L$. Apply I.H. to E' to get E''. For any fresh $f : A'_\ell$ we have $E'' = \Psi, \Delta, f : A'_\ell \vdash [f : A'_\ell/c : A_\ell]Q :: (d : D)$ for $\ell \in J$. We then apply $\oplus L$ to E'' to get $\Psi; \Delta, f : \oplus\{\ell : A'_l\}_{\ell \in J} \vdash \mathsf{case}\ f\ l \Rightarrow [f : A'_\ell/c : A_\ell]Q :: (d : D)$. By our substitution rules, this is equivalent to

$\Psi; \Delta, f : \oplus\{\ell : A'_l\}_{\ell \in J} \vdash [f : \oplus\{\ell : A'_\ell\}/c : \oplus\{\ell : A_\ell\}]\mathsf{case}\ c\ l \Rightarrow Q :: (d : D)$.

Case $\oplus$R.

$$E = \frac{k \in L \quad E' = \Psi; \Delta \vdash P :: (c : A_k)}{\Psi; \Delta \vdash c.k\ ;\ P :: (c : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus R$$

We have $\oplus\{\ell : A_\ell\}_{\ell \in J} \leq \oplus\{\ell : A'_\ell\}_{\ell \in L}$. By subtyping, we have $A_\ell \leq A'_\ell$ for $\ell \in J, J \subseteq L$. Apply I.H. to E' to get E''. For any fresh $f : A'_k$ we have $E'' = \Psi, \Delta \vdash [f : A'_k/c : A_k]P :: (f : A'_k)$. We then apply $\oplus R$ to E'' to get $\Psi; \Delta \vdash f.k; [f : A'_k/c : A_k]P :: (f : \oplus\{\ell : A'_\ell\}_{\ell \in L})$. By our substitution rules, this is equivalent to $\Psi; \Delta \vdash [f : \oplus\{\ell : A'_\ell\}/c : \oplus\{\ell : A_\ell\}]c.k; P :: (f : \oplus\{\ell : A_\ell\}_{\ell \in L})$.

Case $\exists L$.

$$E = \frac{E' = \Psi, n{:}\tau\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \exists n{:}\tau.\ A \vdash n \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \exists L$$

We have $\exists n : \tau'.A' \leq \exists n : \tau.A$. By subtyping, we get that $\tau' \leq \tau$ and $A' \leq A$. Apply I.H. to E' to get E''. For any fresh $f : A'$ we have $E'' = \Psi, n : \tau; \Delta, f : A' \vdash [f : A'/c : A]Q :: (d : D)$. By construction, $|E'| = |E''|$, so we can apply I.H. to E'' to get E'''. For any fresh $m : \tau'$ we have $E''' = \Psi, m : \tau'; \Delta, f' : A' \vdash [m : \tau'/n : \tau][f : A'/c : A]Q :: (d : D)$. We now apply $\exists L$ to E''' to get $\Psi; \Delta, f : \exists m : \tau'.A' \vdash m \leftarrow \mathsf{recv}\ f; [m : \tau'/n : \tau][f : A'/c : A]Q :: (d : D)$. By our substitution rules, this is equivalent to $\Psi; \Delta, f : \exists m : \tau'.A' \vdash [f : \exists n : \tau'.A'/c : \exists n : \tau.A]n \leftarrow \mathsf{recv}\ c; Q :: (d : D)$.

Case $\exists R$.

$$E = \frac{\Psi \vdash v : \tau' \quad E' = \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A) \quad \tau \sim \tau'}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^\rho\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R$$

We have $\exists n : \tau.A \leq \exists n : \tau''.A'$. By subtyping, we have $\tau \leq \tau''$ and $A \leq A'$. Apply I.H. to E' to get E''. For any fresh $f : A'$ we have $E'' = \Psi; \Delta \vdash [f : [v/n]A'/c : [v/n]A]P :: (f : [v/n]A')$. Now apply $\exists R$ to E'' to get $\Psi, \Delta \vdash \mathsf{send}\ f\ \langle \tau \Leftarrow \tau' \rangle^\rho v; [f : A'/c : A]P :: (f : \exists n : \tau.A')$. We update the cast as follows: $\Psi, \Delta \vdash \mathsf{send}\ f\ \langle \tau'' \Leftarrow \tau' \rangle^\rho v; [f : A'/c : A]P :: (f : \exists n : \tau''.A')$. By our substitution rules, this is equivalent to $\Psi, \Delta \vdash [f : \exists n : \tau''.A'/c : \exists n : \tau.A]\mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^\rho v; P :: (f : \exists n : \tau.A)$.

Case $\multimap$L.

$$E = \frac{E' = \Psi\ ;\ \Delta, c : B \vdash Q :: (d : D) \quad A \sim A'}{\Psi\ ;\ \Delta, a : A', c : A \multimap B \vdash \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^\rho\ a\ ;\ Q :: (d : D)}\ \multimap L$$

Subcase principal. We have $A'' \multimap B \leq A \multimap B'$. By subtyping, we get $A \leq A''$ and $B' \leq B$. Apply I.H. to E' to get E''. For any fresh $f : B'$ we have $E'' = \Psi; \Delta, f : B' \vdash [f : B'/c : B]Q :: (d : D)$. Now apply $\multimap$L to E'' to get $\Psi; \Delta, a : A', f : A \multimap B' \vdash$

$\mathsf{send}\ f\ \langle A \Leftarrow A'\rangle^{\rho}\ a; [f : B'/c : B]Q :: (d : D)$. We then update the cast as follows: $\Psi; \Delta, a : A', f : A \multimap B' \vdash \mathsf{send}\ f\ \langle A'' \Leftarrow A'\rangle^{\rho}\ a; [f : B'/c : B]Q :: (d : D)$. By our substitution rules, this is equivalent to $\Psi; \Delta, a : A', f : A'' \multimap B' \vdash [f : A'' \multimap B'/c : A \multimap B]\mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ a\ ;\ Q :: (d : D)$.

Subcase side. Let $A'' \leq A'$. We can update the cast as follows: $\Psi\ ;\ \Delta, a : A'', c : A \multimap B \vdash \mathsf{send}\ c\ \langle A \Leftarrow A''\rangle^{\rho}\ a\ ;\ Q :: (d : D)$. By our substitution rules, this is equivalent to $[a : A''/a : A']\mathsf{send}\ c\ \langle A \Leftarrow A'\rangle^{\rho}\ a\ ;\ Q :: (d : D)$.

Case $\multimap$R.

$$\frac{E' = \Psi; \Delta, x : A \vdash P :: (c : B)}{E = \Psi; \Delta \vdash x \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : A \multimap B)}\ \multimap R$$

We have $A \multimap B \leq A' \multimap B'$. By subtyping, we have $A' \leq A$ and $B \leq B'$. Apply I.H. to E' to get E''. For any fresh $f : C'$ we have $E'' = \Psi; \Delta, x : A \vdash [f : B'/c : B]P :: (f : B')$. By construction, we know that $|E'| = |E''|$. So we now apply I.H. to E'' to get E'''. For any fresh $h : A'$ we have $E''' = \Psi; \Delta, h : A' \vdash [h : A'/x : A][f : B'/c : B]P :: (f : B')$. Now apply $\multimap$R to E''' to get $\Psi; \Delta \vdash h \leftarrow \mathsf{recv}\ f; [h : A'/x : A][f : B'/c : B]P :: (f : A' \multimap B')$. By our substitution rules, this is equivalent to $\Psi; \Delta \vdash [h : A' \multimap B'/c : A \multimap B](x \leftarrow \mathsf{recv}\ c; P) :: (f : A \multimap B)$.

B.5 Configuration Inversion

Lemma 22 (Configuration-Inversion)**.**

1. *If* $\Delta \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$ *where* $\mathcal{C}_2 = \mathsf{proc}(c, P)$ *then* $\Delta \Vdash \mathcal{C}_1$ *and* $\Delta_{|\mathsf{fn}(P)} \Vdash \mathsf{proc}(c, P)$.
2. *If* $\Delta \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ *where* $\mathcal{C}_2 = \mathsf{msg}(c, P)$ *and* $\mathcal{C}_3 = \mathsf{proc}(d, Q)$ *then* $\Delta \Vdash \mathcal{C}_1$ *and* $\Delta_{|\mathsf{fn}(P)} \Vdash \mathsf{msg}(c, P)$ *and* $\Delta_{|\mathsf{fn}(P)} \Vdash \mathsf{proc}(d, Q)$.
3. *If* $\Delta \Vdash \mathcal{C}$ *and* $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3$ *where* $\mathcal{C}_2 = \mathsf{proc}(c, P)$ *and* $\mathcal{C}_3 = \mathsf{msg}(d, Q)$ *then* $\Delta \Vdash \mathcal{C}_1$ *and* $\Delta_{|\mathsf{fn}(P)} \Vdash \mathsf{proc}(c, P)$ *and* $\Delta_{|\mathsf{fn}(Q)} \Vdash \mathsf{msg}(d, Q)$.
4. *If* $\Delta \Vdash \mathsf{proc}(c, P)$ *then* $\Delta_{|\mathsf{fn}(P)} \vdash P :: (c : A)$.
5. *If* $\Vdash \mathsf{msg}(c, P)$ *then there* $\Delta_{\mathsf{fn}(Q)} \vdash P :: (c : A)$.

Proof. By examining the configuration typing rules. □

B.6 Preservation

Case id_cast.

We have $\mathsf{proc}(a, a \leftarrow \langle A \Leftarrow A'\rangle^{\rho} b), \mathcal{C} \longrightarrow [b : A'/a : A]\mathcal{C}$ where $A' \leq A$. We know that $\mathsf{proc}(a, a \leftarrow b)$ is a client of exactly one process, which we can call $\mathsf{proc}(q, Q) \in \mathcal{C}$. This process must be well typed. Let $E' = \Delta, x : A \vdash Q :: (q : C)$. Given $A' \leq A$, we then apply the subtype substitution lemma to E'. For a fresh $b : A'$ we then have $\Delta, b : A' \vdash [b : A'/x : A]Q :: (q : C)$.

Case val_cast.

We have $\mathsf{proc}(a, x \leftarrow \langle \tau' \Leftarrow \tau \rangle^{\rho}\ v; Q) \longrightarrow \mathsf{proc}(a, Q[v : \tau'/x : \tau])$. We need to show that $\mathsf{proc}(a, [v : \tau'/x : \tau]Q)$ is well-typed. By inversion of val_cast, we know that $E' = x : \tau; \Delta \vdash Q :: (c : C) \quad \tau \sim \tau'$ is well-typed. Given that $\tau' \leq \tau$, we can apply the subtype-substitution lemma to E'. For a fresh $v : \tau'$, we get that $v : \tau'; \Delta \vdash [v : \tau'/x : \tau]Q :: (c : C)$.

Case cut.

We have $\mathsf{proc}(c, x{:}A \leftarrow \langle A \Leftarrow A' \rangle^{\rho} P\ ;\ Q) \longrightarrow \mathsf{proc}(a, [a : A'/x : A']P), \mathsf{proc}(c, [a : A'/x : A]Q) \quad$ *(a fresh)*. We need to show that $\mathsf{proc}(a, [a : A'/x : A']P)$ is well-typed. By inversion on cut, we have that $E' =; \Delta \vdash P :: (x : A')$ is well-typed. Given that $A' \leq A'$, we can apply subtype-substitution to E'. For a fresh $a : A'$, we have $\Delta \vdash [a : A'/x : A']P :: (a : A')$. We also need to show that $\mathsf{proc}(c, [a : A'/x : A]Q)$ is well-typed. By inversion on cut, we have that $E'' = x : A, \Delta' \vdash Q :: (c : C)$ is well-typed. Given $A' \leq A$, we can apply the subtype-substitution lemma to E''. For a fresh $a : A'$, we get $a : A', \Delta' \vdash [a : A'/x : A]Q :: (c : C)$.

Case $\otimes$send.

We have $\mathsf{proc}(c, \mathsf{send}\ c\ a\ ;\ P) \longrightarrow \mathsf{proc}(c', [c' : B/c : B]P), \mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^{\rho}\ a\ ;\ c \leftarrow c') \quad$ *(c′ fresh)*. We need to show that $\mathsf{proc}(c, [c' : B/c : B]P)$ is well typed. By inversion of $\otimes$R, we know that $E' = \Delta \vdash P :: (c : B)$ is well typed. Given that $B \leq B$, we can apply the subtype-substitution lemma to E'. For a fresh $c' : B$, we have $\Delta \vdash [c' : B/c : B]P :: (c : B)$. The message is well-typed because of the cast.

Case $\otimes$recv.

We have $\mathsf{msg}(c, \mathsf{send}\ c\ \langle A \Leftarrow A' \rangle^{\rho}\ a\ ;\ c \leftarrow c'), \mathsf{proc}(d, x \leftarrow \mathsf{recv}\ c\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c' : B/c : B][a : A'/x : A]Q)$. We need to show that $\mathsf{proc}(d, [c' : B/c : B][a : A'/x : A]Q)$ is well-typed given that $A' \leq A$. By inversion of $\otimes L$ we know that $E' = \Delta, x : A, c : B \vdash Q :: (d : D)$ is well-typed. We apply the subtype-substitution lemma to E' to get E''. For any fresh $y : A'$ we have $E'' = \Delta, y : A', c : B \vdash [y : A'/x : A]Q :: (d : D)$. Given that $B \leq B$, we can apply subtype-substitution to E''. For a fresh $c' : B$, we have $\Delta, y : A', c' : B \vdash [c' : B/c : B][y : A'/x : A]Q :: (d : D)$.

Case $\&$send.

We have $\mathsf{proc}(d, c.k\ ;\ Q) \longrightarrow \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c' : A_k/c : A_k]Q) \quad (c'$ fresh). We need to show that $\mathsf{proc}(d, [c' : A_k/c : A_k]Q)$ are well-typed. By inversion on $\&L$, we have that $E' = \Delta, c : A_k \vdash Q :: (d : D)$ is well-typed. Given $A_k \leq A_k$, we can apply subtype-substitution to E'. For a fresh $c' : A_k$, we get $\Delta, c' : A_k \vdash [c' : A_k/c : A_k]Q :: (d : D)$. We also have that $c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\ ;\ c' \leftarrow c :: (c' : A_k)$ which types the message.

Case $\&$recv.

We have $\mathsf{proc}(c, \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L}), \mathsf{msg}(c', c.k\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c' : A_\ell/c : A_\ell]P_k)$. We need to show that $\mathsf{proc}(c', [c' : A_\ell/c : A_\ell]P_k)$ is well-typed. By inversion of $\&R$, we know that $E' = \Delta \vdash P_\ell :: (c : A_\ell)$ is well-typed. Because $A_\ell \leq A_\ell$, we can apply subtype-substitution to E'. For a fresh $c' : A_\ell$, we have $\Delta \vdash [c' : A_\ell/c : A_\ell]]P_k :: (c' : A_\ell)$.

Case $\forall$send.

We have $\mathsf{proc}(d, \mathsf{send}\ c\ v\ ;\ Q) \longrightarrow \mathsf{msg}(c', \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^{\rho} v\ ;\ c' \leftarrow c), \mathsf{proc}(d, [c' : [v/n]A/c : [v/n]A]Q)$. We need to show that $\mathsf{proc}(d, [c' : [v/n]A/c : [v/n]A]Q)$ is well-typed. By inversion of $\forall L$, we have that $E' =; \Delta, c : [v/n]A \vdash Q :: (d : D)$ is well-typed. Given that $[v/n]A \leq [v/n]A$, we can apply the subtype-substitution lemma to E'. For a fresh $c' : [v/n]A$, we get $\Delta, c' : [v/n]A \vdash [c : [v/n]A/c' : [v/n]A]Q :: (d : D)$. The message is well-typed because of the cast.

Case $\forall$recv.

We have $\mathsf{proc}(c, x \leftarrow \mathsf{recv}\ c\ ;\ P), \mathsf{msg}(c', \mathsf{send}\ c\ \langle \tau \Leftarrow \tau' \rangle^{\rho} v\ ;\ c' \leftarrow c) \longrightarrow \mathsf{proc}(c', [c' : A/c : A][v : \tau'/n : \tau]P)$. We need to show that $\mathsf{proc}(c', [c' : A/c : A][v : \tau'/n : \tau]P)$ is well-typed. By inversion of $\forall R$, we have that $E' = n{:}\tau\ ;\ \Delta \vdash P :: (c : A)$ is well-typed. Because we have $\tau' \leq \tau$, we can apply the subtype-substitution lemma to E' to get E''. For a fresh $v : \tau'$ we have $E'' = v{:}\tau'\ ;\ \Delta \vdash [v : \tau'/n : \tau]P :: (c : A)$. Given $A \leq A$ we can apply the subtype-substitution lemma to E''. For fresh $c' : A$, we have $v{:}\tau'\ ;\ \Delta \vdash [c' : A/c : A][v : \tau'/n : \tau]P :: (c : A)$.

Appendix C

Dependent Proof Cases

C.1 Irrelevant Substitution

Case id

$$\frac{}{\Psi, c \div \tau; b : A \vdash a \leftarrow b; (a : A)}\ \mathsf{id}$$

We have $\Psi^{\oplus} \vdash v : \tau$. We want to show that $\Psi; [v/c](b : A) \vdash [v/c](a \leftarrow b) :: (a : [v/c]A)$. This simplifies to: $\Psi; b : [v/c]A \vdash a \leftarrow b :: (a : [v/c]A)$. Let $D = [v/c]A$ which gives us $\Psi; b : D \vdash a \leftarrow b :: (a : D)$.

Case cut

$$\frac{E' = \Psi, b \div \tau; \Delta \vdash P :: (x : A) \quad E'' = \Psi, b \div \tau; x : A, \Delta' \vdash Q :: (c : C)}{\Psi, b \div \tau; \Delta, \Delta' \vdash x : A \leftarrow P; Q :: (c : C)}\ \mathsf{cut}$$

We have that $\Psi^{\oplus} \vdash w : \tau$. We want to show that $\Psi; [w/b]\Delta, [w/b]\Delta' \vdash [w/b](x : A \leftarrow P; Q) :: (c : [w/b]C)$. We apply I.H. to E' to get: $\Psi; [w/b]\Delta \vdash [w/b]P :: (x : [w/b]A)$. We apply induction to E'' to get: $\Psi; [w/b](x : A, \Delta') \vdash [w/b]Q :: (c : [w/b]C)$. We then apply the *cut* rule to get: $\Psi; [w/b]\Delta, [w/b]\Delta' \vdash x : [w/b]A \leftarrow [w/b]P; [w/b]Q :: (c : [w/b]C)$.

Case $\exists$R

$$\frac{\Psi, b \div \tau' \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta \vdash P :: (c : [v/n]A)}{\Psi, b \div \tau'; \Delta \vdash \mathsf{send}\ c\ v; P :: (c : \exists n : \tau.A}\ \exists R$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $\Psi; [w/b]\Delta \vdash [w/b](\mathsf{send}\ c\ v; P) :: (c : [w/b]\exists n : \tau.A)$. We apply I.H. to E' to get $\Psi; [w/b]\Delta \vdash [w/b]P :: (c : [v/b][v/n]A)$. If $\Psi, b \div \tau' \vdash v : \tau$, then by irrelevance, it must be the case that $\Psi \vdash v : \tau$. We also note that b cannot appear in v or τ. We then apply the $\exists$R rule to get: $\Psi; [w/b]\Delta \vdash \mathsf{send}\ c\ v; [v/b]P) :: (c : \exists n : \tau.[v/b]A)$.

Case $\exists$L

$$\frac{E' = \Psi, b \div \tau', n : \tau; \Delta, c : A \vdash Q :: (d : D)}{\Psi, b \div \tau'; \Delta, c : \exists n : \tau.A \vdash n \leftarrow \mathsf{recv}\ c; Q :: (d : D)}\ \exists L$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $\Psi; [w/b](\Delta, c : \exists n : \tau.A) \vdash [w/b](n \leftarrow \mathsf{recv}\ c; Q) :: (d : [w/b]D)$. By weakening, we have that $\Psi^{\oplus}, n : \tau \vdash w : \tau'$. This is equivalent to $(\Psi, n : \tau)^{\oplus} \vdash w : \tau'$. We apply I.H. to E' to get $\Psi, n : \tau; [w/b]\Delta, c : [w/b]A \vdash [w/b]Q :: (d : [w/b]D)$. We then apply the $\exists$L rule to get $\Psi; [w/b]\Delta, c : \exists n : \tau.[w/b]A \vdash n \leftarrow \mathsf{recv}\ c; [w/b]Q :: (d : [w/b]D)$.

Case $\forall$L

$$\frac{\Psi, b \div \tau' \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi, b \div \tau'; \Delta, c : \forall n : \tau.A \vdash \mathsf{send}\ c\ v; Q :: (d : D)}\ \forall L$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that
$\Psi; [w/b](\Delta, c : \forall n : \tau.A) \vdash [w/b](\mathsf{send}\ c\ v; Q) :: (d : [w/b]D))$. We apply I.H. to E' to get $\Psi; [w/b]\Delta, c : [w/b][v/n]A \vdash [w/b]Q :: (d : [w/b]D)$. If $\Psi, b \div \tau' \vdash v : \tau$, then by irrelevance, it must be the case that $\Psi \vdash v : \tau$. We also note that b cannot appear in v or τ. We then apply the $\forall$L rule to get $\Psi; [v/b]\Delta, c : \forall n : \tau.[v/b]A \vdash \mathsf{send}\ c\ v; [v/b]Q :: (d : [v/b]D)$.

Case $\forall$R

$$\frac{E' = \Psi, b \div \tau', n : \tau; \Delta \vdash P :: (c : A)}{\Psi, b \div \tau'; \Delta \vdash n \leftarrow \mathsf{recv}\ c; P :: (c : \forall n : \tau.A)}\ \forall R$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $\Psi; [w/b]\Delta \vdash [w/b](n \leftarrow \mathsf{recv}\ c; P) :: (c : [w/b]\forall n : \tau.A)$. By weakening, we have that $\Psi^{\oplus}, n : \tau \vdash w : \tau'$. This is equivalent to $(\Psi, n : \tau)^{\oplus} \vdash w : \tau'$. We apply I.H. to E' to get $\Psi, n : \tau; [w/b]\Delta \vdash [w/b]P :: (c : [w/b]A)$. We then apply the $\forall$R rule to get $\Psi, [w/b]\Delta \vdash n \leftarrow \mathsf{recv}\ c; [w/b]P :: (c : \forall n : \tau.[w/b]A)$.

Case $\exists R_{[]}$

$$\frac{(\Psi, b \div \tau')^{\oplus} \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta \vdash P :: (c : [v/n]A)}{\Psi, b \div \tau'; \Delta \vdash \mathsf{send}\ c\ [v]; P :: (c : \exists n \div \tau.A)}\ \exists R_{[]}$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $\Psi; [w/b]\Delta \vdash [w/b](\mathsf{send}\ c\ [v]; P) :: (c : [w/b]\exists n \div \tau.A)$.. We apply I.H. to E' to get $\Psi; [w/b]\Delta \vdash [w/b]P :: (c : [w/b][v/n]A)$. We have that $(\Psi, b \div \tau')^{\oplus} \vdash v : \tau$ which is equivalent to $\Psi^{\oplus}, b : \tau' \vdash v : \tau$. We now apply standard substitution to get $\Psi^{\oplus} \vdash [w/b]v : [w/b]\tau$. We then apply the $\exists R_{[]}$ rule to get $\Psi; [w/b]\Delta \vdash \mathsf{send}\ c\ [[w/b]v]; [w/b]P :: (c : \exists n \div [w/b]\tau.[w/b]A)$.

Case $\exists L_{[]}$

$$\frac{E' = \Psi, b \div \tau', n \div \tau; \Delta, c : A \vdash Q :: (d : D)}{\Psi, b \div \tau; \Delta, c : \exists n \div \tau.A \vdash [n] \leftarrow \mathsf{recv}\ c; Q :: (d : D)}\ \exists L_{[]}$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $\Psi; [w/b](\Delta, c : \exists n \div \tau.A) \vdash [w/b]([n] \leftarrow \mathsf{recv}\ c; Q) :: (d : [w/b]D)$. By weakening, we have that $\Psi^{\oplus}, n : \tau \vdash w : \tau'$. This is equivalent to $(\Psi, n \div \tau)^{\oplus} \vdash w : \tau'$. We apply I.H. to E' to get $\Psi, n \div \tau; [w/b]\Delta, c : [w/b]A \vdash [w/b]Q :: (d : [w/b]D)$. We then apply the $\exists L_{[]}$ rule to get $\Psi; [w/b]\Delta, c : \exists n \div \tau.[w/b]A \vdash [n] \leftarrow \mathsf{recv}\ c; [w/b]Q :: (d : [w/b]D)$.

Case $\forall L_{[]}$

$$\frac{(\Psi, b \div \tau')^{\oplus} \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi, b \div \tau'; \Delta, c : \forall n \div \tau.A \vdash \mathsf{send}\ c\ [v]; Q :: (d : D)}\ \forall L_{[]}$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $\Psi; [w/b](\Delta, c : \forall n \div \tau.A) \vdash [w/b](\mathsf{send}\ c\ [v]; Q) :: (d : [w/b]D)$. We apply I.H. to E' to get $\Psi; [w/b]\Delta, c : [w/b][v/n]A \vdash [w/b]Q :: (d : [w/b]D)$. We have that $(\Psi, b \div \tau')^{\oplus} \vdash v : \tau$ which is equivalent to $\Psi^{\oplus}, b : \tau' \vdash v : \tau$. We now apply standard substitution to get $\Psi^{\oplus} \vdash [w/b]v : [w/b]\tau$. We then apply the $\forall L_{[]}$ rule to get $\Psi; [w/b]\Delta, c : \forall n \div [w/b]\tau.[w/b]A \vdash \mathsf{send}\ c\ [[w/b]v]; [w/b]Q :: (d : [w/b]D)$.

Case $\forall R_{[]}$

$$\frac{E' = \Psi, b \div \tau', n \div \tau; \Delta \vdash P :: (c : A)}{\Psi, b \div \tau'; \Delta \vdash [n] \leftarrow \mathsf{recv}\ c; P :: (c : \forall n \div \tau.A)}\ \forall R_{[]}$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $\Psi; [w/b]\Delta \vdash [w/b]([n] \leftarrow \mathsf{recv}\ c; P) :: (c : [w/b]\forall n \div \tau.A)$. By weakening, we have that $\Psi^{\oplus}, n : \tau \vdash w : \tau'$. This is equivalent to $(\Psi, n \div \tau)^{\oplus} \vdash w : \tau'$. We apply I.H. to E' to get $\Psi, n \div \tau; [w/b]\Delta \vdash [w/b]P :: (c : [w/b]A)$. We then apply the $\forall R_{[]}$ rule to get $\Psi; [w/b]\Delta \vdash [n] \leftarrow \mathsf{recv}\ c; [w/b]P :: (c : \forall n \div \tau.[w/b]A)$.

C.2 Proof of lemma (one-step)

Case exists_s$_{[]}$

We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P)$, $!(\exists \phi.\exists p.c_i : \exists n \div \phi.A), p : \phi)$ $\longrightarrow$ $\mathsf{proc}(c_i^+, [c_i^+/c_i]P)$, $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+)$, $!(c_i^+ : A)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ; P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ []\ ;\ P :: (c_i : \Delta(c_i))$. By inversion of the $\exists_{[]}$R rule, we have $\Delta_{|\mathsf{fn}(P)} \vdash P :: (c_i : \Delta(c_i))$. By using the $\exists_{[]}$R and the id rule, we can type the message as follows: $c_i^+ : A \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n \div \phi.A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+/c_i]P :: (c_i^+ : \Delta(c_i^+))$ and $\Delta'_{|c_i^+} \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n \div \phi.A)$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$ and $H; \Delta'_{|c_i^+} \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+)$.

Subcase $c_i \in H$

Using the fact that $c_i^+ : A$ and $c_i : \exists n \div \phi.A$ we can type the message as follows: $c_i^+ : A \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n \div \phi.A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+} \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n \div \phi.A)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+)$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, CH, [c_i^+/c_i]P)$.

Case $\mathsf{exists\_s\_a}_{[]}$

We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P), !\neg(\exists\phi.\exists p.c_i : \exists n \div \phi.A), p : \phi) \longrightarrow \mathsf{alarm}(c_i)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ [v]\ ;\ P :: (c_i : \Delta(c_i))$. By the $\exists_{[]}\mathrm{R}$ rule, $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ [v]; P :: (c_i : \exists n \div \phi.A)$. If $\neg(\exists\phi.\exists p.c_i : \exists n \div \phi.A), p : \phi)$, then $\Delta_{|\mathsf{fn}(P)} \nVdash \mathsf{send}\ c_i\ [v]; P :: (c_i : \exists n \div \phi.A)$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P)$, it must be the case that $c_i \in H$.

Subcase $c_i \in H$

Assume $\Delta_{|\mathsf{fn}(P)} \vdash \mathsf{send}\ c_i\ [v]\ ;\ P :: (c_i : \Delta(c_i))$. By the $\exists\mathrm{R}$ rule, it must be the case that $c_i : \exists n \div \phi.A$ for some ϕ. However, this contradicts the fact that there does not exist ϕ and p such that $c_i : \exists n \div \phi.A$ and $p : \phi$. Therefore, $\Delta_{|\mathsf{fn}(P)} \nVdash \mathsf{send}\ c_i\ [v]\ ;\ P :: (c_i : \Delta(c_i))$.

Case $\mathsf{exists\_r}_{[]}$

We have $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [] \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q)$. By the configuration typing, we have that $H; \Delta \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+)$ and $H; \Delta \Vdash \mathsf{proc}(d, [] \leftarrow \mathsf{recv}\ c_i\ ;\ Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash [v] \leftarrow \mathsf{recv}\ c_i\ ;\ Q :: (d : \Delta(d))$ and $\Delta_{c_i^+} \vdash \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+ :: (c_i : \Delta(c_i)$. By using the $\exists_{[]}\mathrm{R}$ and the id rule, we can type the message as follows: $c_i^+ : A \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+ :: (c_i : \exists n \div \tau.A)$. By inversion of the $\exists_{[]}\mathrm{L}$ typing rule we have that $n \div \tau; \Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. By inversion of the $\exists_{[]}\mathrm{R}$ typing rule, we have $\cdot \vdash v : \tau$. We now apply Irrelevant Substitution to get: $\cdot; [v/n]\Delta_{|\mathsf{fn}(Q)} \vdash [v/n]Q :: (d : [v/n]\Delta(d))$. Let $\Delta' = [c_i^+ : A^-/c_i : A^-]\Delta$ and $\Delta'_{|\mathsf{fn}(Q)} = [c_i^+ : A/c_i : A]\Delta_{|\mathsf{fn}(Q)}$. We then have that $\cdot; \Delta'_{|\mathsf{fn}(Q)} \vdash [c_i^+/c_i]Q :: (d : \Delta(d))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Subcase $d \in H$

Let $\Delta' = \Delta$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case $\mathsf{forall\_s}_{[]}$

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q), !(\exists\phi.\exists p.c_i : \forall n \div \phi.A), p : \phi) \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;$

$c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q), !(c_i^+ : A)$. By the configuration typing, $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ [v]\ ;\ Q :: (d : \Delta(d))$. By inversion of the $\forall_{[]}$ L rule, we have $\Delta_{|\mathsf{fn}(Q)} \vdash Q :: (d : \Delta(d))$. By using the $\forall_{[]}$L and the id rule, we can type the message as follows: $c_i : \forall n \div \phi.A \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i^+ \cup \mathsf{fn}(Q)} \vdash [c_i^+/c_i]Q :: (d : \Delta'(d))$ and $\Delta'_{|c_i} \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : \Delta'(c_i^+))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$ and $H; \Delta' \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i)$.

Subcase $d \in H$

Using the fact that $c_i^+ : A$ and $c_i : \forall n \div \phi.A$ we can type the message as follows: $c_i : \forall n \div \phi.A \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : A)$. Let $\Delta' = \Delta \cup c_i^+$. We then have $\Delta'_{|c_i} \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : A)$. Therefore, $H; \Delta' \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i)$. Because $d \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(d, [c_i^+/c_i]Q)$.

Case forall_s_a$_{[]}$

We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q), !\neg(\exists\phi.\exists p.c_i : \forall n \div \phi.A), p : \phi) \longrightarrow \mathsf{alarm}(d)$. By the configuration typing, $H; \Psi; \Delta \Vdash \mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P)$.

Subcase $d \notin H$

By inversion of the configuration typing, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ [v]\ ;\ Q) :: (d : \Delta(d))$. By the $\forall_{[]}$L rule, $\Delta_{|\mathsf{fn}(Q)}, c_i : \forall n \div \phi.A \vdash \mathsf{send}\ c_i\ [v]\ ;\ Q :: (d : \Delta(d))$. If $\neg(\exists\phi.\exists p.c_i : \forall n \div \phi.A), p : \phi)$, then $\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ [v]\ ;\ Q :: (d : \Delta(d))$. This is a contradiction, and because $H; \Delta \Vdash \mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q)$, it must be the case $d \in H$.

Subcase $d \in H$

Assume $\Psi; \Delta_{|c_i \cup \mathsf{fn}(Q)} \vdash \mathsf{send}\ c_i\ [v]\ ;\ Q :: (d : \Delta(d))$. By the $\forall$L rule, it must be the case that $c_i : \exists n \div \phi.A$ for some ϕ. However, this contradicts the fact that there does not exist ϕ and p such that $c_i : \exists n \div \phi$ and $p : \phi$. Therefore, $\Delta_{|c_i \cup \mathsf{fn}(Q)} \not\vdash \mathsf{send}\ c_i\ [v]\ ;\ Q) :: (d : \Delta(d))$.

Case forall_r$_{[]}$

We have $\mathsf{proc}(c_i, [] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$. By the configuration typing, we have that $H; \Delta \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i)$ and $H; \Delta \Vdash \mathsf{proc}(c_i, [] \leftarrow \mathsf{recv}\ c_i\ ;\ P)$.

Subcase $c_i \notin H$

By inversion of the configuration typing, $\Delta_{|\mathsf{fn}(P)} \vdash [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P :: (c_i : \Delta(c_i))$ and $\Delta_{|c_i} \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i :: (c_i : \Delta(c_i))$. By the $\forall$L rule and the id rule, we can type the message as follows: $c_i : \forall n \div \tau.A \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : A)$. By inversion of the $\forall_{[]}$R typing rule we have that $n \div \tau; \Delta_{|\mathsf{fn}(P)} \vdash P :: (c_i : A)$. By inversion of the $\forall_{[]}$L typing rule, we have that $\cdot \vdash v : \tau$. We now apply Irrelevant Substitution to get: $\cdot; [v/n]\Delta_{\mathsf{fn}(P)} \vdash [v/n]P :: (c_i : [v/n]A))$. Let $\Delta' = [c_i^+ : A^-/c_i : A^-]\Delta$ and $\Delta'_{|\mathsf{fn}(P)} = [c_i^+ : A/c_i : A]\Delta_{|\mathsf{fn}(P)}$. We then have that

$\cdot; \Delta'_{|\mathsf{fn}(P)} \vdash [c_i^+/c_i]P :: (c_i^+ : \Delta'(c_i^+))$. Therefore, $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$.

Subcase $c_i \in H$

Let $\Delta' = \Delta$. Because $c_i \in H$, by the configuration typing, we have $H; \Delta' \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$.

C.3 Erasure Correctness

Case id

$$\frac{}{\Psi; b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id}$$

We want to show that $\Psi^\dagger; (b : A)^\dagger \vdash (a \leftarrow b)^\dagger :: (a : A^\dagger)$. This is equivalent to $\Psi^\dagger; b : A^\dagger \vdash a \leftarrow b :: (a : A^\dagger)$. Let $\Psi' = \Psi^\dagger$ and $A' = A^\dagger$. We then have $\Psi; b : A' \vdash a \leftarrow b :: (a : A')$.

Case 1R

$$\frac{}{\cdot; \cdot \vdash \mathsf{close}\ c\ :: (c : 1)}\ 1R$$

We want to show that $\cdot^\dagger; \cdot^\dagger \vdash (\mathsf{close}\ c)^\dagger :: (c : 1^\dagger)$. This is equivalent to $\cdot; \cdot \vdash \mathsf{close}\ c :: (c : 1)$.

Case 1L

$$\frac{E' = \Psi; \Delta \vdash Q :: (d : D)}{\Psi; \Delta, c : 1 \vdash \mathsf{wait}\ c; Q :: (d : D)}\ 1L$$

We want to show that $\Psi^\dagger; (\Delta, c : 1)^\dagger \vdash (\mathsf{wait}\ c; Q)^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger; \Delta^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We then apply the 1L rule to get $\Psi^\dagger; \Delta^\dagger, c : 1 \vdash \mathsf{wait}\ c; Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger; \Delta^\dagger, (c : 1)^\dagger \vdash (\mathsf{wait}\ c; Q)^\dagger :: (d : D^\dagger)$.

Case cut

$$\frac{E' = \Psi; \Delta \vdash P :: (x : A) \quad E'' = \Psi; x : A, \Delta' \vdash Q :: (c : C)}{E' = \Psi; \Delta, \Delta' \vdash x : A \leftarrow P; Q :: (c : C)}\ \mathsf{cut}$$

We want to show that $\Psi^\dagger; (\Delta, \Delta')^\dagger \vdash (x : A \leftarrow P; Q)^\dagger :: (c : C^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger; \Delta^\dagger \vdash P^\dagger :: (x : A^\dagger)$. We apply I.H. to E'' to get $\Psi^\dagger; x : A^\dagger, \Delta'^\dagger \vdash Q^\dagger :: (c : C^\dagger)$. We then apply the cut rule to get $\Psi^\dagger; \Delta^\dagger, \Delta'^\dagger \vdash x : A^\dagger \leftarrow P^\dagger; Q^\dagger :: (c : C^\dagger)$.

Case $\&$L

$$\frac{k \in L \quad E' = \Psi; \Delta, c : A_k \vdash Q :: (d : D)}{\Psi; \Delta, c : \&\{\ell : A_\ell\}_{\ell \in L} \vdash c.k\ ; Q :: (d : D)}\ \&L$$

We want to show that $\Psi^\dagger; (\Delta, c : \&\{\ell : A_\ell\}_{\ell \in L})^\dagger \vdash (c.k\ ;\ Q)^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger; \Delta^\dagger, c : A_k^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We then apply the $\&$L rule to get $\Psi^\dagger; \Delta^\dagger, c : \&\{\ell : A_\ell^\dagger\}_{\ell \in L} \vdash c.k\ ;\ Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger; \Delta^\dagger, c : \&\{\ell : A_\ell\}_{\ell \in L}^\dagger \vdash (c.k\ ;\ Q)^\dagger :: (d : D^\dagger)$.

Case $\&$R

$$\frac{E' = \Psi; \Delta \vdash P_\ell :: (c : A_\ell) \quad \text{for every } \ell \in L}{\Psi; \Delta \vdash \mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L} :: (c : \&\{\ell : A_\ell\}_{\ell \in L})}\ \&R$$

We want to show that $\Psi^\dagger; \Delta^\dagger \vdash (\mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L})^\dagger :: (c : \&\{\ell : A_\ell\}_{\ell \in L}^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger; \Delta^\dagger \vdash P_\ell^\dagger :: (c : A_\ell^\dagger)$. We then apply the $\&$R rule to get $\Psi^\dagger; \Delta^\dagger \vdash \mathsf{case}\ c\ (\ell \Rightarrow P_\ell^\dagger)_{\ell \in L} :: (c : \&\{\ell : A_\ell^\dagger\}_{\ell \in L})$ which is equivalent to $\Psi^\dagger; \Delta^\dagger \vdash (\mathsf{case}\ c\ (\ell \Rightarrow P_\ell)_{\ell \in L})^\dagger :: (c : \&\{\ell : A_\ell\}_{\ell \in L}^\dagger)$.

We note that the $\oplus$L and $\oplus$R cases are similar.

Case $\multimap$L

$$\frac{E' = \Psi\ ;\ \Delta, c : B \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, a : A, c : A \multimap B \vdash \mathsf{send}\ c\ a\ ;\ Q :: (d : D)}\ \multimap L$$

We want to show that $\Psi^\dagger\ ;\ (\Delta, a : A, c : A \multimap B)^\dagger \vdash (\mathsf{send}\ c\ a\ ;\ Q)^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger\ ;\ (\Delta^\dagger, c : B^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We then apply the $\multimap$L rule to get $\Psi^\dagger\ ;\ \Delta^\dagger, a : A, c : A \multimap B^\dagger \vdash \mathsf{send}\ c\ a\ ;\ Q^\dagger :: (d : D^\dagger)$. Let $A = A^\dagger$. We then have $\Psi^\dagger\ ;\ \Delta^\dagger, a : A^\dagger, c : A^\dagger \multimap B^\dagger \vdash \mathsf{send}\ c\ a\ ;\ Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger\ ;\ \Delta^\dagger, a : A^\dagger, (c : A \multimap B)^\dagger \vdash (\mathsf{send}\ c\ a\ ;\ Q)^\dagger :: (d : D^\dagger)$.

Case $\multimap$R

$$\frac{E' = \Psi; \Delta, x : A \vdash P :: (c : B)}{\Psi; \Delta \vdash x \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : A \multimap B)}\ \multimap R$$

We want to show that $\Psi^\dagger; \Delta^\dagger \vdash (x \leftarrow \mathsf{recv}\ c\ ;\ P)^\dagger :: (c : (A \multimap B)^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger; \Delta^\dagger, x : A^\dagger \vdash P^\dagger :: (c : B^\dagger)$. We then apply the $\multimap$R rule to get $\Psi^\dagger; \Delta^\dagger \vdash x \leftarrow \mathsf{recv}\ c\ ;\ P^\dagger :: (c : A^\dagger \multimap B^\dagger)$ which is equivalent to $\Psi^\dagger; \Delta^\dagger \vdash (x \leftarrow \mathsf{recv}\ c\ ;\ P)^\dagger :: (c : (A \multimap B)^\dagger)$.

Case $\otimes$L

$$\frac{E' = \Psi; \Delta, x : A, c : B \vdash Q :: (d : D)}{\Psi; \Delta, c : A \otimes B \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q :: (d : D)}\ \otimes L$$

We want to show that $\Psi^\dagger; (\Delta, c : A \otimes B)^\dagger \vdash (x \leftarrow \mathsf{recv}\ c\ ;\ Q)^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger; \Delta^\dagger, x : A^\dagger, c : B^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We then apply the $\otimes$L rule to get $\Psi^\dagger; \Delta^\dagger, c : A^\dagger \otimes B^\dagger \vdash x \leftarrow \mathsf{recv}\ c\ ;\ Q^\dagger :: (d : D^\dagger)$ which is equivalent to

$\Psi^\dagger; \Delta^\dagger, c : (A \otimes B)^\dagger \vdash (x \leftarrow \mathsf{recv}\ c\ ;\ Q)^\dagger :: (d : D^\dagger)$.

Case $\otimes$R

$$\frac{E' = \Psi\ ;\ \Delta \vdash P :: (c : B)}{\Psi\ ;\ \Delta, a : A \vdash \mathsf{send}\ c\ \ a\ ;\ P :: (c : A \otimes B)}\ \otimes R$$

We want to show that $\Psi^\dagger\ ;\ (\Delta, a : A)^\dagger \vdash (\mathsf{send}\ c\ \ a\ ;\ P0^\dagger :: (c : (A \otimes B)^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger; \Delta^\dagger \vdash P^\dagger :: (c : B^\dagger)$. We then apply the $\otimes$R rule to get $\Psi^\dagger\ ;\ \Delta^\dagger, a : A \vdash \mathsf{send}\ c\ \ a\ ;\ P^\dagger :: (c : A \otimes B^\dagger)$. Let $A = A^\dagger$. We then have $\Psi^\dagger\ ;\ \Delta^\dagger, a : A^\dagger \vdash \mathsf{send}\ c\ \ a\ ;\ P^\dagger :: (c : A^\dagger \otimes B^\dagger)$ which is equivalent to $\Psi^\dagger\ ;\ \Delta^\dagger, a : A^\dagger \vdash (\mathsf{send}\ c\ \ a\ ;\ P)^\dagger :: ((c : A \otimes B)^\dagger)$.

Case $\uparrow$R

$$\frac{E' = \Psi\ ;\ \Delta \vdash P :: (c : A)}{\Psi\ ;\ \Delta \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P :: (c : \uparrow A)}\ \uparrow R$$

We want to show that $\Psi^\dagger\ ;\ \Delta^\dagger \vdash (\mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P^\dagger) :: (c : (\uparrow A)^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger\ ;\ \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger)$. We then apply the $\uparrow$R rule to get $\Psi^\dagger\ ;\ \Delta^\dagger \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P^\dagger :: (c : \uparrow A^\dagger)$ which is equivalent to $\Psi^\dagger\ ;\ \Delta^\dagger \vdash (\mathsf{shift} \leftarrow \mathsf{recv}\ c\ ;\ P)^\dagger :: (c : (\uparrow A)^\dagger)$.

Case $\uparrow$L

$$\frac{E' = \Psi\ ;\ \Delta, c : A \vdash Q :: (d : D)}{\Psi\ ;\ \Delta, c : \uparrow A \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q :: (d : D)}\ \uparrow L$$

We want to show that $\Psi^\dagger\ ;\ (\Delta^\dagger, c\ :\ \uparrow A)^\dagger\ \vdash\ (\mathsf{send}\ \ c\ \ \mathsf{shift}\ \ ;\ \ Q)^\dagger\ \ ::\ \ (d\ :\ D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger\ ;\ \Delta^\dagger, c\ :\ A^\dagger\ \vdash\ Q^\dagger\ ::\ (d\ :\ D^\dagger)$. We then apply the $\uparrow$L rule to get $\Psi^\dagger\ ;\ \Delta^\dagger, c : \uparrow A^\dagger \vdash \mathsf{send}\ c\ \mathsf{shift}\ ;\ Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger\ ;\ \Delta^\dagger, c : (\uparrow A)^\dagger \vdash (\mathsf{send}\ c\ \mathsf{shift}\ ;\ Q)^\dagger :: (d : D^\dagger)$.

We note that the $\downarrow$R and $\downarrow$L cases are similar.

Case $\exists$R

$$\frac{\Psi \vdash v : \tau \quad E' = \Psi\ ;\ \Delta \vdash P :: (c : [v/n]A)}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ c\ v\ ;\ P :: (c : \exists n{:}\tau.\ A)}\ \exists R$$

We want to show that $\Psi^\dagger\ ;\ \Delta^\dagger \vdash (\mathsf{send}\ c\ v\ ;\ P)^\dagger :: (c : (\exists n{:}\tau.\ A)^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger\ ;\ \Delta^\dagger \vdash P^\dagger :: (c : ([v/n]A)^\dagger)$ which is equivalent to $\Psi^\dagger\ ;\ \Delta^\dagger \vdash P^\dagger :: (c : [v/n]A^\dagger)$. Let $\Psi = \Psi_1, \Psi_2$ where Ψ_1 and Ψ_2 consists of the irrelevant and relevant expressions respectively. We then have $\Psi_1, \Psi_2 \vdash v : \tau$. By irrelevance, $\Psi_2 \vdash v : \tau$. By definition of erasure, $\Psi^\dagger = (\Psi_1, \Psi_2)^\dagger = \Psi_1^\dagger, \Psi_2^\dagger = \Psi_2^\dagger$. Therefore, we have $\Psi^\dagger \vdash v : \tau$. We then apply the $\exists$R rule to get $\Psi^\dagger\ ;\ \Delta^\dagger \vdash \mathsf{send}\ \ c\ \ v\ \ ;\ \ P^\dagger\ \ ::\ \ (c\ :\ \exists n{:}\tau.\ A^\dagger)$ which is equivalent to $\Psi^\dagger\ ;\ \Delta^\dagger \vdash (\mathsf{send}\ c\ v\ ;\ P)^\dagger :: (c : \exists n{:}\tau.\ A)^\dagger$.

Case $\exists$L

$$\frac{E' = \Psi, n{:}\tau \ ; \ \Delta, c : A \vdash Q :: (d : D)}{\Psi \ ; \ \Delta, c : \exists n{:}\tau.\, A \vdash n \leftarrow \mathsf{recv}\ c \ ; \ Q :: (d : D)}\ \exists L$$

We want to show that $\Psi^\dagger \ ; \ (\Delta, c : \exists n{:}\tau.\, A)^\dagger \vdash (n \leftarrow \mathsf{recv}\ c \ ; \ Q)^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger, n{:}\tau \ ; \ \Delta^\dagger, c : A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We then apply the $\exists$L rule to get $\Psi^\dagger \ ; \ \Delta^\dagger, c : \exists n{:}\tau.\, A^\dagger \vdash n \leftarrow \mathsf{recv}\ c \ ; \ Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger, c : (\exists n{:}\tau.\, A)^\dagger \vdash (n \leftarrow \mathsf{recv}\ c \ ; \ Q)^\dagger :: (d : D^\dagger)$.

Case $\forall$R

$$\frac{E' = \Psi, n{:}\tau \ ; \ \Delta \vdash P :: (c : A)}{\Psi \ ; \ \Delta \vdash n \leftarrow \mathsf{recv}\ c \ ; \ P :: (c : \forall n{:}\tau.\, A)}\ \forall R$$

We want to show that $\Psi^\dagger \ ; \ \Delta^\dagger \vdash (n \leftarrow \mathsf{recv}\ c \ ; \ P)^\dagger :: (c : (\forall n{:}\tau.\, A)^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger, n{:}\tau \ ; \ \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger)$. We then apply the $\forall$R rule to get $\Psi^\dagger \ ; \ \Delta^\dagger \vdash n \leftarrow \mathsf{recv}\ c \ ; \ P^\dagger :: (c : \forall n{:}\tau.\, A^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger \vdash (n \leftarrow \mathsf{recv}\ c \ ; \ P)^\dagger :: (c : (\forall n{:}\tau.\, A)^\dagger)$.

Case $\forall$L

$$\frac{\Psi \vdash v : \tau \quad E' = \Psi \ ; \ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi \ ; \ \Delta, c : \forall n{:}\tau.\, A \vdash \mathsf{send}\ c\ v \ ; \ Q :: (d : D)}\ \forall L$$

We want to show that $\Psi^\dagger \ ; \ (\Delta, c : \forall n{:}\tau.\, A)^\dagger \vdash (\mathsf{send}\ c\ v \ ; \ Q^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger \ ; \ \Delta^\dagger, c : ([v/n]A^\dagger) \vdash Q^\dagger :: (d : D)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger, c : [v/n]A^\dagger \vdash Q^\dagger :: (d : D)$. Let $\Psi = \Psi_1, \Psi_2$ where Ψ_1 and Ψ_2 consists of the irrelevant and relevant expressions respectively. We then have $\Psi_1, \Psi_2 \vdash v : \tau$. By irrelevance, $\Psi_2 \vdash v : \tau$. By definition of erasure, $\Psi^\dagger = (\Psi_1, \Psi_2)^\dagger = \Psi_1^\dagger, \Psi_2^\dagger = \Psi_2^\dagger$. Therefore, we have $\Psi^\dagger \vdash v : \tau$. We then apply the $\forall$L rule to get $\Psi^\dagger \ ; \ \Delta^\dagger, c : \forall n{:}\tau.\, A^\dagger \vdash \mathsf{send}\ c\ v \ ; \ Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger, c : (\forall n{:}\tau.\, A)^\dagger \vdash (\mathsf{send}\ c\ v \ ; \ Q)^\dagger :: (d : D^\dagger)$.

Case $\exists R_{[]}$

$$\frac{\Psi^\oplus \vdash v : \tau \quad E' = \Psi \ ; \ \Delta \vdash P :: (c : [v/n]A)}{\Psi \ ; \ \Delta \vdash \mathsf{send}\ c\ [v] \ ; \ P :: (c : \exists n \div \tau.\, A)}\ \exists R_{[]}$$

We want to show that $\Psi^\dagger \ ; \ \Delta^\dagger \vdash (\mathsf{send}\ c\ [v] \ ; \ P)^\dagger :: (c : (\exists n \div \tau.\, A)^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger \ ; \ \Delta^\dagger \vdash P^\dagger :: (c : ([v/n]A)^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger \vdash P^\dagger :: (c : [v/n]A^\dagger)$. By definition of erasure, because the promoted context $\Psi^\oplus$ cannot contain irrelevant expressions, we have that $(\Psi^\oplus)^\dagger = \Psi^\oplus$. We then apply the $\exists$R rule to get $\Psi^\dagger \ ; \ \Delta^\dagger \vdash \mathsf{send}\ c\ [v] \ ; \ P^\dagger :: (c : \exists n \div \tau.A^\dagger)$. Let $\tau = 1$, which gives us $\Psi^\dagger \ ; \ \Delta^\dagger \vdash \mathsf{send}\ c\ [] \ ; \ P^\dagger :: (c : \exists n \div 1.A^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger \vdash (\mathsf{send}\ c\ [v] \ ; \ P)^\dagger :: (c : (\exists n \div \tau.\, A)^\dagger)$.

Case $\exists L_{[]}$

$$\frac{E' = \Psi, n \div \tau \ ; \ \Delta, c : A \vdash Q :: (d : D)}{\Psi \ ; \ \Delta, c : \exists n \div \tau.\, A \vdash [n] \leftarrow \mathsf{recv}\ c \ ; \ Q :: (d : D)}\ \exists L_{[]}$$

We want to show that $\Psi^\dagger \ ; \ (\Delta, c : \exists n \div \tau.\, A)^\dagger \vdash ([n] \leftarrow \mathsf{recv}\ c \ ; \ Q)^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger, (n \div \tau)^\dagger \ ; \ \Delta^\dagger, c : A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger, c : A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We then apply the $\exists L_{[]}$ rule to get $\Psi^\dagger \ ; \ \Delta^\dagger, c : \exists n \div \tau . A^\dagger \vdash [n] \leftarrow \mathsf{recv}\ c \ ; \ Q^\dagger :: (d : D^\dagger)$. Let $\tau = 1$, which gives us $\Psi^\dagger \ ; \ \Delta^\dagger, c : \exists n \div 1 . A^\dagger \vdash [] \leftarrow \mathsf{recv}\ c \ ; \ Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger, c : (\exists n \div \tau.\, A)^\dagger \vdash ([n] \leftarrow \mathsf{recv}\ c \ ; \ Q)^\dagger :: (d : D^\dagger)$.

Case $\forall R_{[]}$

$$\frac{E' = \Psi, n \div \tau \ ; \ \Delta \vdash P :: (c : A)}{\Psi \ ; \ \Delta \vdash [n] \leftarrow \mathsf{recv}\ c \ ; \ P :: (c : \forall n \div \tau.\, A)}\ \forall R_{[]}$$

We want to show that $\Psi^\dagger \ ; \ \Delta^\dagger \vdash ([n] \leftarrow \mathsf{recv}\ c \ ; \ P)^\dagger :: (c : (\forall n \div \tau.\, A)^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger, (n \div \tau)^\dagger \ ; \ \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger)$. We then apply the $\forall R_{[]}$ rule to get $\Psi^\dagger \ ; \ \Delta^\dagger \vdash [n] \leftarrow \mathsf{recv}\ c \ ; \ P^\dagger :: (c : \exists n \div \tau . A^\dagger)$. Let $\tau = 1$, which gives us $\Psi^\dagger \ ; \ \Delta^\dagger \vdash [] \leftarrow \mathsf{recv}\ c \ ; \ P^\dagger :: (c : \exists n \div 1 . A^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger \vdash ([n] \leftarrow \mathsf{recv}\ c \ ; \ P)^\dagger :: (c : (\forall n \div \tau.\, A)^\dagger)$.

Case $\forall L_{[]}$

$$\frac{\Psi^\oplus \vdash v : \tau \quad E' = \Psi \ ; \ \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi \ ; \ \Delta, c : \forall n \div \tau.\, A \vdash \mathsf{send}\ c\ [v] \ ; \ Q :: (d : D)}\ \forall L_{[]}$$

We want to show that $\Psi^\dagger \ ; \ (\Delta, c : \forall n \div \tau.\, A)^\dagger \vdash (\mathsf{send}\ c\ [v] \ ; \ Q)^\dagger :: (d : D^\dagger)$. We apply I.H. to E' to get $\Psi^\dagger \ ; \ \Delta^\dagger, c : ([v/n]A)^\dagger \vdash Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger, c : [v/n]A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. By definition of erasure, because the promoted context $\Psi^\oplus$ cannot contain irrelevant expressions, we have that $(\Psi^\oplus)^\dagger = \Psi^\oplus$. We then apply the $\forall L_{[]}$ rule to get $\Psi^\dagger \ ; \ \Delta^\dagger, c : \forall n \div \tau . A^\dagger \vdash \mathsf{send}\ c\ [v] \ ; \ Q^\dagger :: (d : D^\dagger)$. Let $\tau = 1$, which gives us $\Psi^\dagger \ ; \ \Delta^\dagger, c : \forall n \div 1 . A^\dagger \vdash \mathsf{send}\ c\ [] \ ; \ Q^\dagger :: (d : D^\dagger)$ which is equivalent to $\Psi^\dagger \ ; \ \Delta^\dagger, c : (\forall n \div \tau.\, A)^\dagger \vdash (\mathsf{send}\ c\ [v] \ ; \ Q)^\dagger :: (d : D^\dagger)$.

C.4 Irrelevant Erasure

Case id

$$\frac{}{\Psi, c \div \tau; b : A \vdash a \leftarrow b :: (a : A)}\ \mathsf{id}$$

We want to show that $(\Psi, c \div \tau)^\dagger; (b : A)^\dagger \vdash (a \leftarrow b)^\dagger :: (a : A^\dagger)$. This is equivalent to $\Psi^\dagger; b : A^\dagger \vdash a \leftarrow b :: (a : A^\dagger)$. Let $\Psi' = \Psi^\dagger$ and $A' = A^\dagger$. We then have $\Psi', b : A' \vdash a \leftarrow b :: (a : A')$.

Case cut

$$\frac{E_1 = \Psi, b \div \tau; \Delta \vdash P :: (x : A) \quad E_2 = \Psi, b \div \tau; x : A, \Delta' \vdash Q :: (c : C)}{\Psi, b \div \tau; \Delta, \Delta' \vdash x : A \leftarrow P; Q :: (c : C)}\ \mathsf{cut}$$

We have that $\Psi^{\oplus} \vdash v : \tau$. We want to show that $(\Psi, b \div \tau)^{\dagger}; (\Delta, \Delta')^{\dagger} \vdash (x : A \leftarrow P; Q)^{\dagger} :: (c : C^{\dagger})$. This is equivalent to $\Psi^{\dagger}; \Delta^{\dagger}, \Delta'^{\dagger} \vdash x : A^{\dagger} \leftarrow P^{\dagger}; Q^{\dagger} :: (c : C^{\dagger})$. We first apply I.H. to E_1 to get $E_3 = (\Psi, b \div \tau)^{\dagger}; \Delta^{\dagger} \vdash P^{\dagger} :: (x : A^{\dagger}) = \Psi^{\dagger}; \Delta^{\dagger} \vdash P^{\dagger} :: (x : A^{\dagger})$. We then apply I.H. to E_2 to get $E_4 = (\Psi, b \div \tau)^{\dagger}; (x : A, \Delta')^{\dagger} \vdash Q^{\dagger} :: (c : C^{\dagger}) = \Psi^{\dagger}; x : A^{\dagger}, \Delta'^{\dagger} \vdash Q^{\dagger} :: (c : C^{\dagger})$. We then apply the cut rule to E_3 and E_4 to get $\Psi^{\dagger}; \Delta^{\dagger}, \Delta'^{\dagger} \vdash x : A^{\dagger} \leftarrow P^{\dagger}; Q^{\dagger} :: (c : C^{\dagger})$.

Case $\exists$R

$$\frac{\Psi, b \div \tau' \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta \vdash P :: (c : [v/n]A)}{\Psi, b \div \tau'; \Delta \vdash \mathsf{send}\ c\ v; P :: (c : \exists n : \tau.A)}\ \exists R$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau')^{\dagger}; \Delta^{\dagger} \vdash \mathsf{send}\ c\ v; P^{\dagger} :: (c : (\exists n : \tau.A)^{\dagger})$. This is equivalent to $\Psi^{\dagger}; \Delta^{\dagger} \vdash \mathsf{send}\ c\ v; P^{\dagger} :: (c : \exists n : \tau.A^{\dagger})$. We first apply I.H. to E' to get $E'' = (\Psi, b \div \tau')^{\dagger}; \Delta^{\dagger} \vdash P^{\dagger} :: (c : [v/n]A^{\dagger}) = \Psi^{\dagger}; \Delta^{\dagger} \vdash P^{\dagger} :: (c : [v/n]A^{\dagger})$. We know that $\Psi, b \div \tau' \vdash v : \tau$. Let $\Psi = \Psi_1, \Psi_2$ where Ψ_1 contains assumptions of the form $a : \tau$ and Ψ_2 contains assumptions of the form $b \div \tau'$. By definition of erasure, $\Psi^{\dagger} = (\Psi_1, \Psi_2)^{\dagger} = \Psi_1^{\dagger}, \Psi_2^{\dagger} = \Psi_1^{\dagger}$. Because Ψ_1 only contains relevant assumptions, we have that $\Psi_1 = \Psi_1^{\dagger}$. By irrelevance, $\Psi_1 \vdash v : \tau$, which gives us $\Psi^{\dagger} \vdash v : \tau$. We now apply the $\exists R$ rule to get $\Psi^{\dagger}; \Delta^{\dagger} \vdash \mathsf{send}\ c\ v; P^{\dagger} :: (c : \exists n : \tau.A^{\dagger})$.

Case $\exists$L

$$\frac{E' = \Psi, b \div \tau', n : \tau; \Delta, c : A \vdash Q :: (d : D)}{\Psi, b \div \tau'; \Delta, c : \exists n : \tau.A \vdash n \leftarrow \mathsf{recv}\ c; Q :: (d : D)}\ \exists L$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau')^{\dagger}; (\Delta, c : \exists n : \tau.A)^{\dagger} \vdash n \leftarrow \mathsf{recv}\ c; Q^{\dagger} :: (d : D^{\dagger})$. This is equivalent to $\Psi^{\dagger}; \Delta^{\dagger}, c : \exists n : \tau.A^{\dagger})^{\dagger} \vdash n \leftarrow \mathsf{recv}\ c; Q^{\dagger} :: (d : D^{\dagger})$. We apply I.H. to E' to get $E'' = (\Psi, b \div \tau', n : \tau)^{\dagger}; (\Delta, c : A)^{\dagger} \vdash Q^{\dagger} :: (d : D^{\dagger}) = \Psi^{\dagger}, n : \tau; \Delta^{\dagger}, c : A^{\dagger} \vdash Q^{\dagger} :: (d : D^{\dagger})$. We now apply the $\exists L$ rule to get $\Psi^{\dagger}; \Delta^{\dagger}, c : \exists n : \tau.A^{\dagger} \vdash n \leftarrow \mathsf{recv}\ c; Q^{\dagger} :: (d : D)$.

Case $\forall$L

$$\frac{\Psi, b \div \tau' \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi, b \div \tau'; \Delta, c : \forall n : \tau.A \vdash \mathsf{send}\ c\ v; Q :: (d : D)}\ \forall L$$

We have that $\Psi^{\oplus} \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau')^{\dagger}; \Delta^{\dagger}, c : \forall n : \tau.A^{\dagger} \vdash \mathsf{send}\ c\ v; Q^{\dagger} :: (d : D^{\dagger})$. This is equivalent to $\Psi^{\dagger}; \Delta^{\dagger}, c : \forall n : \tau.A^{\dagger} \vdash \mathsf{send}\ c\ v; Q^{\dagger} :: (d : D^{\dagger})$. We first apply I.H. to E' to get $E'' = (\Psi, b \div \tau')^{\dagger}; (\Delta, c : [v/n]A)^{\dagger} \vdash Q^{\dagger} :: (d : D^{\dagger}) = \Psi^{\dagger}; \Delta^{\dagger}, c : [v/n]A^{\dagger} \vdash Q^{\dagger} :: (d : D^{\dagger})$. We know that $\Psi, b \div \tau' \vdash v : \tau$. Let $\Psi = \Psi_1, \Psi_2$ where Ψ_1 contains assumptions of the form $a : \tau$ and Ψ_2 contains assumptions of the form $b \div \tau'$. By definition of

erasure, $\Psi^\dagger = (\Psi_1, \Psi_2)^\dagger = \Psi_1^\dagger, \Psi_2^\dagger = \Psi_1^\dagger$. Because Ψ_1 only contains relevant assumptions, we have that $\Psi_1 = \Psi_1^\dagger$. By irrelevance, $\Psi_1 \vdash v : \tau$, which gives us $\Psi^\dagger \vdash v : \tau$. We now apply the $\forall L$ rule to get $\Psi^\dagger; \Delta^\dagger, c : \forall n : \tau.A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$.

Case $\forall$R

$$\frac{E' = \Psi, b \div \tau', n : \tau; \Delta \vdash P :: (c : A)}{\Psi, b \div \tau'; \Delta \vdash n \leftarrow \mathsf{recv}\ c; P :: (c : \forall n : \tau.A)}\ \forall R$$

We have that $\Psi^\oplus \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau')^\dagger; \Delta^\dagger \vdash n \leftarrow \mathsf{recv}\ c; P^\dagger :: (c : (\forall n : \tau.A)^\dagger)$. This is equivalent to $\Psi^\dagger; \Delta^\dagger \vdash n \leftarrow \mathsf{recv}\ c; P^\dagger :: (c : \forall n : \tau.A^\dagger)$. We apply I.H. to E' to get $E'' = (\Psi, b \div \tau', n : \tau)^\dagger; \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger) = \Psi^\dagger, n : \tau; \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger)$. We now apply the $\forall R$ rule to get $\Psi^\dagger; \Delta^\dagger \vdash n \leftarrow \mathsf{recv}\ c; P^\dagger :: (c : \forall n : \tau.A^\dagger)$.

Case $\exists R_{[]}$

$$\frac{(\Psi, b \div \tau')^\oplus \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta \vdash P :: (c : [v/n]A)}{\Psi, b \div \tau'; \Delta \vdash \mathsf{send}\ c\ [v]; P :: (c : \exists n \div \tau.A)}\ \exists R_{[]}$$

We have that $\Psi^\oplus \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau')^\dagger; \Delta^\dagger \vdash \mathsf{send}\ c\ [v]; P^\dagger :: (c : (\exists n \div \tau.A)^\dagger)$. This is equivalent to $\Psi^\dagger; \Delta^\dagger \vdash \mathsf{send}\ c\ [v]; P^\dagger :: (c : \exists n \div \tau.A^\dagger)$. We first apply I.H. to E' to get $E'' = (\Psi, b \div \tau')^\dagger; \Delta^\dagger \vdash P^\dagger :: (c : [v/n]A^\dagger) = \Psi^\dagger; \Delta^\dagger \vdash P^\dagger :: (c : [v/n]A^\dagger)$. We have $(\Psi, b \div \tau')^\oplus \vdash v : \tau$. This is equivalent to $\Psi^\oplus, b : \tau' \vdash v : \tau$. We apply standard substitution to get $\Psi^\oplus \vdash [w/b]v : [w/b]\tau$. By definiton, erasing a promoted context leaves the promoted context unchanged, so $(\Psi^\oplus)^\dagger = \Psi^\oplus$. We then have $(\Psi^\oplus)^\dagger \vdash [w/b]v : [w/b]\tau$. We now apply the $\exists R_{[]}$ rule to get $\Psi^\dagger; \Delta^\dagger \vdash \mathsf{send}\ c\ [[w/b]v]; P^\dagger :: (c : \exists n \div [w/b]\tau.A^\dagger)$. Let $v' = [w/b]v$ and $\tau'' = [w/b]\tau$. We then have $\Psi^\dagger; \Delta^\dagger \vdash \mathsf{send}\ c\ [v']; P^\dagger :: (c : \exists n \div \tau''.A^\dagger)$.

Case $\exists L_{[]}$

$$\frac{E' = \Psi, b \div \tau', n \div \tau; \Delta, c : A \vdash Q :: (d : D)}{\Psi, b \div \tau; \Delta, c : \exists n \div \tau.A \vdash [n] \leftarrow \mathsf{recv}\ c; Q :: (d : D)}\ \exists L_{[]}$$

We have that $\Psi^\oplus \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau)^\dagger; (\Delta, c : \exists n \div \tau.A)^\dagger \vdash [n] \leftarrow \mathsf{recv}\ c; Q^\dagger :: (d : D^\dagger)$. This is equivalent to $\Psi^\dagger; \Delta^\dagger, c : \exists n \div \tau.A^\dagger \vdash [n] \leftarrow \mathsf{recv}\ c; Q^\dagger :: (d : D^\dagger)$. We apply I.H. to E" to get $E" = (\Psi, b \div \tau', n \div \tau)^\dagger; (\Delta, c : A)^\dagger \vdash Q^\dagger :: (d : D^\dagger) = \Psi^\dagger, n \div \tau; \Delta^\dagger, c : A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We now apply the $\exists L_{[]}$ rule to get $\Psi^\dagger; \Delta^\dagger, c : \exists n \div \tau.A \vdash [n] \leftarrow \mathsf{recv}\ c; Q^\dagger :: (d : D^\dagger)$.

Case $\forall L_{[]}$

$$\frac{(\Psi, b \div \tau')^\oplus \vdash v : \tau \quad E' = \Psi, b \div \tau'; \Delta, c : [v/n]A \vdash Q :: (d : D)}{\Psi, b \div \tau'; \Delta, c : \forall n \div \tau.A \vdash \mathsf{send}\ c\ [v]; Q :: (d : D)}\ \forall L_{[]}$$

We have that $\Psi^\oplus \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau')^\dagger; (\Delta, c : \forall n \div \tau.A)^\dagger \vdash \mathsf{send}\ c\ [v]; Q^\dagger :: (d : D^\dagger)$. This is equivalent to $\Psi^\dagger; \Delta^\dagger, c : \forall n \div \tau.A^\dagger \vdash \mathsf{send}\ c\ [v]; Q^\dagger :: (d : D^\dagger)$.

We first apply I.H. to E' to get $E'' = (\Psi, b \div \tau')^\dagger; (\Delta, c : [v/n]A)^\dagger \vdash Q^\dagger :: (d : D^\dagger) = \Psi^\dagger; \Delta^\dagger, c : [v/n]A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. We have $(\Psi, b \div \tau')^\oplus \vdash v : \tau$. This is equivalent to $\Psi^\oplus, b : \tau' \vdash v : \tau$. We apply standard substitution to get $\Psi^\oplus \vdash [w/b]v : [w/b]\tau$. By definiton, erasing a promoted context leaves the promoted context unchanged, so $(\Psi^\oplus)^\dagger = \Psi^\oplus$. We then have $(\Psi^\oplus)^\dagger \vdash [w/b]v : [w/b]\tau$.

We now apply the $\forall L_{[]}$ rule to get $\Psi^\dagger; \Delta^\dagger, c : \forall n \div [w/b]\tau.A^\dagger \vdash \mathsf{send}\ c\ [[w/b]v]; Q^\dagger :: (d : D^\dagger)$. Let $v' = [w/b]v$ and $\tau'' = [w/b]\tau$. We then have $\Psi^\dagger; \Delta^\dagger, c : \forall n \div \tau''.A^\dagger \vdash \mathsf{send}\ c\ [v]; Q'^\dagger :: (d : D^\dagger)$.

Case $\forall R_{[]}$

$$\frac{E' = \Psi, b \div \tau', n \div \tau; \Delta \vdash P :: (c : A)}{\Psi, b \div \tau'; \Delta \vdash [n] \leftarrow \mathsf{recv}\ c; P :: (c : \forall n \div \tau.A)}\ \forall R_{[]}$$

We have that $\Psi^\oplus \vdash w : \tau'$. We want to show that $(\Psi, b \div \tau')^\dagger; \Delta^\dagger \vdash [n] \leftarrow \mathsf{recv}\ c; P^\dagger :: (c : (\forall n \div \tau.A)^\dagger)$. This is equivalent to $\Psi^\dagger; \Delta^\dagger \vdash [n] \leftarrow \mathsf{recv}\ c; P^\dagger :: (c : \forall n \div \tau.A^\dagger)$. We apply I.H. to E' to get $E'' = (\Psi, b \div \tau', n \div \tau)^\dagger; \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger) = \Psi^\dagger, n \div \tau; \Delta^\dagger \vdash P^\dagger :: (c : A^\dagger)$. We then apply the $\forall R_{[]}$ to get $\Psi^\dagger; \Delta^\dagger \vdash [n] \leftarrow \mathsf{recv}\ c; P^\dagger :: (c : \forall n \div \tau.A^\dagger)$.

C.5 Erasure Preservation

Case $\mathsf{forall\_s}_{[]}$

1. We have $\mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q) \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q)$. Let $C = \mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q)$ and $C' = \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q)$. Let $C'' = \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q^\dagger)$. We need to show that $C^\dagger \rightarrow C''$ and $(C')^\dagger = C''$. We have that $C^\dagger = \mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q)^\dagger = \mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q^\dagger)$. By the $\mathsf{forall\_s}_{[]}$ rule, $\mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q^\dagger) \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q^\dagger)$, which gives us $C^\dagger \rightarrow C''$. We have that $(C')^\dagger = (\mathsf{msg}(c_i^+, \mathsf{send}\ c_i; [v]; c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q))^\dagger = (\mathsf{msg}(c_i^+, \mathsf{send}\ c_i; [v]; c_i^+ \leftarrow c_i))^\dagger, (\mathsf{proc}(d, [c_i^+/c_i]Q))^\dagger = \mathsf{msg}(c_i^+, (\mathsf{send}\ c_i\ [v]; c_i^+ \leftarrow c_i)^\dagger), \mathsf{proc}(d, [c_i^+/c_i]Q^\dagger) = \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q^\dagger) = C''$.

2. We have $(\mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q))^\dagger \longrightarrow (\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q))^\dagger$. Let $C^\dagger = (\mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q))^\dagger = \mathsf{proc}(d, \mathsf{send}\ c_i\ []\ ;\ Q^\dagger)$ and $(C')^\dagger = (\mathsf{msg}(c_i^+, \mathsf{send} c_i; [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q))^\dagger = (\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i))^\dagger, (\mathsf{proc}(d, [c_i^+/c_i]Q))^\dagger = \mathsf{msg}(c_i^+, (\mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i)^\dagger), \mathsf{proc}(d, [c_i^+/c_i]Q^\dagger) = \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q^\dagger)$. Let $C'' = \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q)$. We need to show that $C \rightarrow C''$ and $C' = C''$. We have that $C = \mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q)$. By the $\mathsf{forall\_s}_{[]}$ rule, $\mathsf{proc}(d, \mathsf{send}\ c_i\ [v]\ ;\ Q) \longrightarrow \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q)$, which gives us $C \rightarrow C''$. We have that $C' = \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i), \mathsf{proc}(d, [c_i^+/c_i]Q) = C''$.

Case forall_r$_{[]}$

1. We have $\mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P)$. Let $C = \mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i)$ and $C' = \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P)$. Let $C'' = \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger)$. We need to show that $C^\dagger \rightarrow C''$ and $(C')^\dagger = C''$. We have that $C^\dagger = (\mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i))^\dagger = (\mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P))^\dagger, (\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i))^\dagger = \mathsf{proc}(c_i, [] \leftarrow \mathsf{recv}\ c_i\ ;\ P^\dagger), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i)$. By the forall_r$_{[]}$ rule, $\mathsf{proc}(c_i, [] \leftarrow \mathsf{recv}\ c_i\ ;\ P^\dagger), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$, which gives us $C^\dagger \rightarrow C''$. We have that $(C')^\dagger = (\mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P))^\dagger = \mathsf{proc}(c_i^+, [c_i^+/c_i]([v/n]P)^\dagger)$. We have $\Psi^\dagger; [c_i^+/c_i]([v/n]\Delta)^\dagger \vdash [c_i^+/c_i]([v/n]P)^\dagger :: (c_i^+ : ([v/n]A)^\dagger)$. Because $\cdot; \Psi; \Omega \Vdash C$, we have $\cdot; \Psi; \Omega \Vdash \mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P)$ and $\cdot; \Psi; \Omega \Vdash \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i)$. Therefore, $\Psi; \Delta \vdash [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P :: (c_i : \forall n \div \tau.A)$ and $\Psi; \Delta \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i :: (c_i^+ : [v/n]A)$. By inversion of the $\forall R_{[]}$ and $\forall L_{[]}$ typing rules, we have $\Psi, n \div \tau; \Delta \vdash P :: (c_i : A)$ and $\Psi^\oplus \vdash v : \tau$. By the Irrelevant Erasure lemma, $\Psi^\dagger; \Delta^\dagger \vdash P^\dagger :: (c_i : A^\dagger)$. Substituting c_i^+ for c_i, we get $\Psi^\dagger; [c_i^+/c_i]\Delta^\dagger \vdash [c_i^+/c_i]P^\dagger :: (c_i^+ : A^\dagger)$. We then have $\Psi^\dagger; [c_i^+/c_i]\Delta \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger)$ which is equivalent to C''.

2. We have $(\mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i))^\dagger \longrightarrow (\mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P))^\dagger$. Let $C^\dagger = (\mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i))^\dagger = (\mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P))^\dagger, (\mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i))^\dagger = \mathsf{proc}(c_i, [] \leftarrow \mathsf{recv}\ c_i\ ;\ P^\dagger), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ []\ ;\ c_i^+ \leftarrow c_i)$ and $(C')^\dagger = (\mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P))^\dagger$. Let $C'' = \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P)$. We need to show that $C \rightarrow C''$ and $C' = C''$. We have that $C = \mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i)$. By the forall_r$_{[]}$ rule, $\mathsf{proc}(c_i, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ P), \mathsf{msg}(c_i^+, \mathsf{send}\ c_i\ [v]\ ;\ c_i^+ \leftarrow c_i) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P)$, which gives us $C \rightarrow C''$. We have that $C' = \mathsf{proc}(c_i^+, [c_i^+/c_i][v/n]P) = C''$.

Case exists_s$_{[]}$

1. We have $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+)$. Let $C = \mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P)$ and $C' = \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+)$. Let $C = \mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P)$. Let $C'' = \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+)$. We need to show that $C^\dagger \rightarrow C''$ and $(C')^\dagger = C''$. We have that $C^\dagger = (\mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P))^\dagger = \mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P^\dagger)$. By the exists_s$_{[]}$ rule, $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ []\ ;\ P^\dagger) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+)$, which gives us $C^\dagger \rightarrow C''$. We have that $(C')^\dagger = (\mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+))^\dagger = (\mathsf{proc}(c_i^+, [c_i^+/c_i]P))^\dagger$, $(\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+))^\dagger = \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger), \mathsf{msg}(c_i, (\mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+)^\dagger) = \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+) = C''$.

2. We have $(\mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P))^\dagger \longrightarrow (\mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+))^\dagger$. Let $C^\dagger = (\mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P))^\dagger = \mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P^\dagger)$ and $(C')^\dagger = (\mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+))^\dagger = (\mathsf{proc}(c_i^+, [c_i^+/c_i]P))^\dagger$, $(\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+))^\dagger = \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger), \mathsf{msg}(c_i, (\mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow$

$c_i^+)^\dagger) = \mathsf{proc}(c_i^+, [c_i^+/c_i]P^\dagger), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+)$. Let $C'' = \mathsf{proc}(c_i^+, [c_i^+/c_i]P)$, $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+)$. We need to show that $C \rightarrow C''$ and $C' = C''$. We have that $C = \mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P)$. By the exists_s$_{[]}$ rule, $\mathsf{proc}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ P) \longrightarrow \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+)$, which gives us $C \rightarrow C''$. We have that $C' = \mathsf{proc}(c_i^+, [c_i^+/c_i]P), \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+) = C''$.

Case exists_r$_{[]}$

1. We have $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \longrightarrow$ $\mathsf{proc}(d, [c_i^+/c_i][v/n]Q)$. Let $C = \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q)$ and $C' = \mathsf{proc}(d, [c_i^+/c_i][v/n]Q)$. Let $C'' = \mathsf{proc}(d, [c_i^+/c_i]Q^\dagger)$. We need to show that $C^\dagger \rightarrow C''$ and $(C')^\dagger = C''$. We have that $C^\dagger = (\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q))^\dagger = (\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+))^\dagger, (\mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q))^\dagger = \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [] \leftarrow \mathsf{recv}\ c_i\ ;\ Q^\dagger)$. By the exists_r$_{[]}$ rule, $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [] \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c_i^+/c_i]Q)$, which gives us $C^\dagger \rightarrow C''$. We have that $(C')^\dagger = (\mathsf{proc}(d, [c_i^+/c_i][v/n]Q))^\dagger =$ $\mathsf{proc}(d, [c_i^+/c_i]([v/n]Q)^\dagger)$. We have $\Psi^\dagger; [c_i^+/c_i]([v/n]\Delta)^\dagger \vdash [c_i^+/c_i]([v/n]Q)^\dagger :: (d : ([v/n]D)^\dagger)$. Because $\cdot; \Psi; \Omega \Vdash C$, we have $\cdot; \Psi; \Omega \Vdash \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q)$ and $\cdot; \Psi; \Omega \Vdash \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+)$. Therefore, $\Psi; \Delta, c : \exists n \div \tau.A \vdash [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q :: (d : D)$ and $\Psi; \Delta \vdash \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+ :: (c_i : [v/n]A)$. By inversion of the $\exists L_{[]}$ and $\exists R_{[]}$ typing rules, we have $\Psi, n \div \tau; \Delta, c_i : A \vdash Q :: (d : D)$ and $\Psi^\oplus \vdash v : \tau$. By the Irrelevant Erasure lemma, $\Psi^\dagger; \Delta^\dagger, c_i : A^\dagger \vdash Q^\dagger :: (d : D^\dagger)$. Substituting c_i^+ for c_i, we get $\Psi^\dagger; [c_i^+/c_i]\Delta^\dagger, c_i^+ : A^\dagger \vdash [c_i^+/c_i]Q^\dagger :: (d : D^\dagger)$. We then have $\Psi^\dagger; [c_i^+/c_i]\Delta, c_i^+ : A^\dagger \Vdash \mathsf{proc}(c_i^+, [c_i^+/c_i]Q^\dagger)$ which is equivalent to C''.

2. We have $(\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q))^\dagger$ $\longrightarrow (\mathsf{proc}(d, [c_i^+/c_i][v/n]Q))^\dagger$. Let $C^\dagger = (\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q))^\dagger = (\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+))^\dagger, (\mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q))^\dagger = \mathsf{msg}(c_i, \mathsf{send}\ c_i\ []\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [] \leftarrow \mathsf{recv}\ c_i\ ;\ Q^\dagger)$ and $(C')^\dagger =$ $(\mathsf{proc}(d, [c_i^+/c_i][v/n]Q))^\dagger$. Let $C'' = \mathsf{proc}(d, [c_i^+/c_i][v/n]Q)$. We need to show that $C \rightarrow C''$ and $C' = C''$. We have that $C = \mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q)$. By the exists_r$_{[]}$ rule, $\mathsf{msg}(c_i, \mathsf{send}\ c_i\ [v]\ ;\ c_i \leftarrow c_i^+), \mathsf{proc}(d, [n] \leftarrow \mathsf{recv}\ c_i\ ;\ Q) \longrightarrow \mathsf{proc}(d, [c_i^+/c_i][v/n]Q)$, which gives us $C \rightarrow C''$. We have that $C' = \mathsf{proc}(d, [c_i^+/c_i][v/n]Q) = C''$.

Bibliography

[1] Umut A Acar and Guy E Blelloch. *Algorithms: Parallel and Sequential*. URL `http://www.parallel-algorithms-book.com/`. 6.2

[2] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*, 2011. doi: 10.1145/1570506.1570507. URL `https://doi.acm.org/10.1145/1570506.1570507`. 2.2, 5.4

[3] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proc. ACM Program. Lang.*, 1(ICFP):37:1–37:29, August 2017. ISSN 2475-1421. doi: 10.1145/3110281. URL `https://doi.acm.org/10.1145/3110281`. 2.1, 7.3

[4] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In Luís Caires, editor, *Programming Languages and Systems*, pages 611–639, Cham, 2019. Springer International Publishing. 7.3

[5] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. In *Formal Techniques for Distributed Systems (FMOODS 2013)*, 2013. 3.5

[6] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21st International Conference on Concurrency Theory (CONCUR 2010)*, 2010. doi: 10.1007/978-3-642-15375-4_16. URL `https://dx.doi.org/10.1007/978-3-642-15375-4_16`. 2.1

[7] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types. 2.1

[8] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, 2009. doi: 10.1016/j.ic.2008.11.006. URL `https://dx.doi.org/10.1016/j.ic.2008.11.006`. 2.1

[9] Tzu-Chen Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *6th International Symposium on Trustworthy Global Computing (TGC 2011)*, pages 25–45, Aachen, Germany, June 2012. Springer LNCS 7173. 3.5

[10] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, pages 215–226, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926410. URL https://doi.acm.org/10.1145/1926385.1926410. 2.2, 5.4

[11] Christos Dimoulas, Sam T. Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *21st European Conference on Programming Languages and Systems (ESOP 2012)*, 2012. doi: 10.1007/978-3-642-28869-2_11. URL https://dx.doi.org/10.1007/978-3-642-28869-2_11. 2.2, 5.4

[12] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, 2011. doi: 10.1145/2034773.2034800. URL https://doi.acm.org/10.1145/2034773.2034800. 4.4

[13] Luminous Fennell and Peter Thiemann. The blame theorem for a linear lambda calculus with type dynamic. In *13th International Symposium on Trends in Functional Programming (TFP 2012)*, 2012. 2.2

[14] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581484. URL https://doi.acm.org/10.1145/581478.581484. 1, 2.2, 5.4

[15] Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005. doi: 10.1007/s00236-005-0177-z. URL https://dx.doi.org/10.1007/s00236-005-0177-z. 2.1, 5.3

[16] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. Session-typed concurrent contracts. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 771–798, 2018. doi: 10.1007/978-3-319-89884-1_27. URL https://doi.org/10.1007/978-3-319-89884-1_27. 1, 4.3

[17] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 353–364, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706341. URL http://doi.acm.org/10.1145/1706299.1706341. 6.4

[18] Dennis Griffith. *Polarized Substructural Session Types*. PhD thesis, University of Illinois at Urbana-Champaign, April 2016. 2.1

[19] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR 1993)*, 1993. 2.1

[20] Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proc. ACM Program. Lang.*, 1(ICFP):38:1–38:28, August 2017. ISSN 2475-1421. doi: 10.1145/3110282. URL https:/doi.acm.org/10.1145/3110282.

5.4

[21] Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and blame assignment for higher-order session types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 582–594, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837662. URL https://doi.acm.org/10.1145/2837614.2837662. 1, 3.4

[22] Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*, 2015. doi: 10.1145/2784731.2784737. URL https://doi.acm.org/10.1145/2784731.2784737. 2.2, 5.4

[23] Hernán Melgratti and Luca Padovani. Chaperone contracts for higher-order sessions. *Proc. ACM Program. Lang.*, 1(ICFP):35:1–35:29, August 2017. ISSN 2475-1421. doi: 10.1145/3110279. URL https://doi.acm.org/10.1145/3110279. 4.4

[24] Don P. Mitchell and Michael J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 282–284, New York, NY, USA, 1984. ACM. ISBN 0-89791-143-1. doi: 10.1145/800222.806755. URL http://doi.acm.org/10.1145/800222.806755. 7.3

[25] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: 10.1145/263699.263712. URL http://doi.acm.org/10.1145/263699.263712. 1, 6

[26] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: Compile-time api generation of distributed protocols with refinements in f#. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 128–138, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179495. URL http://doi.acm.org/10.1145/3178372.3179495. 6.4

[27] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US. ISBN 978-1-4020-8141-5. 6.4

[28] Frank Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 221–, Washington, DC, USA, 2001. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=871816.871845. 1, 6

[29] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In *18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2015)*, 2015. doi: 10.1007/978-3-662-46678-0_1. URL https://doi.acm.org/10.1007/978-3-662-46678-0_1. Invited talk. 2.1, 4.1

[30] Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-carrying code in a session-typed

process calculus. In *1st International Conference on Certified Programs and Proofs (CPP 2011)*, 2011. 6

[31] Jeremy Siek, Peter Thiemann, and Philip Wadler. Blame and Coercion: Together Again for the First Time. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*, 2015. 2.2

[32] Peter Thiemann. Session Types with Gradual Typing. In *9th International Symposium on Trustworthy Global Computing (TGC 2014)*, 2014. doi: 10.1007/978-3-662-45917-1_10. URL https://dx.doi.org/10.1007/978-3-662-45917-1_10. 5.4

[33] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 462–475, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.4230/LIPIcs.ECOOP.2016.9. URL https:/acm.doi.org/10.4230/LIPIcs.ECOOP.2016.9. 4.2

[34] Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and New University of Lisbon, 2015. 2.1

[35] Bernardo Toninho and Nobuko Yoshida. Depending on session-typed processes. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 128–145, 2018. doi: 10.1007/978-3-319-89366-2_7. URL https://doi.org/10.1007/978-3-319-89366-2_7. 6.4

[36] Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International Conference on Principles and Practice of Declarative Programming*, PPDP'11, pages 161–172, Odense, Denmark, July 2011. ACM. 6

[37] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *22nd European Symposium on Programming (ESOP 2013)*, 2013. doi: 10.1007/978-3-642-37036-6_20. URL https://dx.doi.org/10.1007/978-3-642-37036-6_20. 1, 2.1

[38] Philip Wadler. A Complement to Blame. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.309. URL https://doi.acm.org/10.4230/LIPIcs.SNAPL.2015.309. 2.2, 5.4

[39] Philip Wadler and Robert B. Findler. Well-Typed Programs Can't Be Blamed. In *18th European Symposium on Programming Languages and Systems (ESOP 2009)*, 2009. doi: 10.1007/978-3-642-00590-9_1. URL https://dx.doi.org/10.1007/978-3-642-00590-9_1. 2.2, 3.5, 5, 5.4

[40] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, November 1997. ISSN 0004-5411. doi: 10.1145/268999.269003. URL https://doi.acm.org/10.1145/268999.269003. 4.2

[41] Lucas Waye, Stephen Chong, and Christos Dimoulas. Whip: Higher-order contracts for

modern services. *Proc. ACM Program. Lang.*, 1(ICFP):36:1–36:28, August 2017. ISSN 2475-1421. URL https://doi.acm.org/10.1145/3110280. 3.5, 4.4