POLARIZED SUBSTRUCTURAL SESSION TYPES

BY

DENNIS EDWARD GRIFFITH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Abstract

Concurrent processes can be extremely difficult to reason about, both for programmers and formally. One approach to coping with this difficulty is to study new programming languages and type features such as Session Types. Session types take as their conceptual notion of concurrency as a collection of processes linked together via channels and provide type-level coordination between processes using these channels.

Logically motivated programming languages exploit the idea that providing a proof of a theorem in a logic is similar to proving that a given term has a particular type in a programming language and vice versa. These connections can be interesting for a few different reasons. First, when language and logic are independently discovered and independently useful, the existence of a connection suggests that both are onto some fundamentally important idea. Additionally, a connection provides a basis both for sanity checking our ideas and also can be fruitful grounds for inspiration by seeing how variants of either the logic or the language are reflected through the connection.

This thesis primarily describes an exploration of logically motivated session types, SILL. Polarization, classifying propositions as either positive or negative, provides a natural way to describe a logically based session typing language with asynchronous communication while retaining a semantics that is reasonably implementable. Additionally, polarization gives us a way to smoothly integrate synchronous channels into SILL without needing a semantic extension. When combined with Adjoint Logic, this gives us an ability to incorporate a variety of modalities with relatively little work. From a practical perspective, this gives SILL access to persistent processes and garbage collection.

We additionally explore a trio of loosely related extensions to SILL, and their logical connections, inspired by the above results: bundled message passing to reduce the number of communications performed by processes; racy programs, enabled by a select/epoll-like mechanism; and asynchronous receiving, an almost generalization of the basic asynchronous semantics. We have three different implementations of SILL: a simple but relatively full featured interpreter written in OCaml; a fragment of SILL as an embedded domain specific language in Haskell; and a cleaner version of the same in Idris.

Lastly, we show that Liquid Types and Session Types are compatible. This gives us one notion of a dependently session typed language.

Table of Contents

List of Figures

Chapter 1

Introduction

This thesis describes an exploration in concurrent programming language design, focusing on session types. Our aim is to study logically motivated session typing systems which have received increasing research attention recently [16, 28, 72, 89, 94]. The main result of this investigation is SILL, a language that demonstrates the compatibility of a number of interesting and practically important language features with a logical basis.

1.1 Motivation

Modern computers force programmers to think about concurrency frequently, but modern programmers are often equipped languages that were designed with concurrency as a secondary focus. This leaves an opportunity for new languages with a greater care for concurrency to be developed and understood, hopefully enabling better tools for mainstream programmers. We hope SILL might be another step in developing such languages.

1.1.1 Types

When programming there is a strong need to classify the behaviors of programs to enable thinking about them at a high-level and ensure compatibility of interaction between components of the program. Good type systems assist the programmer in several important ways. First, they give users a vocabulary for thinking about their program behavior, discussing programs and problems with colleagues, and documenting work. Additionally, types help programmers avoid dangerous (either unambiguously wrong or hard to get right) programs. Lastly, types can provide a convenient basis for language implementation design or even enable critical optimizations.

When statically checked, types become even more exciting. One of the most compelling feature of type systems is that their banning of bad behavior means that static checks can rule out entire, hopefully broad and important, classes of bugs before code is executed. In this sense static type systems can be thought of as one of the most important and widely deployed formal verification methods. When made explicit in the program, types can serve as machine checked documentation, more usefully with more expressive type systems. When inferable, static type systems can provide most, or all, of the convenience of their dynamically checked, or unchecked, counterparts with no extra user overhead and some of the previously mentioned benefits. Lastly, these types can enable comparatively expensive type-directed optimizations, of which we will see an example of in this thesis, by amortizing the cost of performing the optimization across all runs of a program.

1.1.2 Concurrency

Concurrent processes can be extremely difficult to reason about, both for programmers and formally. One approach to coping with this difficulty is to study new programming languages and paradigms that try to simplify this task, generally by only allowing for well behaved programs to be constructed, as with type systems. There has been both a large amount of foundational work in this area [3, 59, 60, 83] as well as practical languages resulting from this work [6, 67]. Some of these, particularly when intended for high-performance computing, work by attaching new abstractions to existing languages or paradigms that either make concurrent programs easy to right correctly [78] or completely ban certain classes of dangerous concurrent behavior [52, 79, 86], e.g., raciness for software transactional memory.

One approach to these concurrency problems that has been receiving increasing attention is session types [15,16,28,36,42,46,61,62,64,72,75,90,91,99]. Session types take as their conceptual notion of concurrency a collection of processes linked together via channels. Naively integrated, channels have a number of problems due to the need to coordinate the involved processes' behaviors to enable sensible communication. Session types help us answer the following coordination questions about the users of a channel at any given point of program execution: "Who is sending information?"; "Who is receiving information?"; "What sort of information is being sent?" Each of these questions can have different answers depending on the system. The first question is invariably answered with only one process, but we can either have a constant process for a channel [39] or a potentially different process for every communication along a channel [41]. The second question allows us to say whether the channel is involved in point-to-point communication, if only one process can receive the information, and the most common choice, or broadcast communication [46], a relatively unexplored space. The last question has a variety of answers, from the simplest of "a single fixed type," to elaborate systems that allow for the transmission of channels over channels [41]. Another question some session typing systems can answer is how to coordinate the behavior of multiple processes [42].

1.1.3 Logically Based Languages

Logically motivated programming languages [33, 68, 94, 95] exploit the idea that providing a proof of a theorem in a logic is similar to proving that a given term has a particular type in a programming language and vice versa. These connections can be interesting for a few different reasons. First, when language and logic are independently discovered and independently useful, the existence of a connection suggests that both are onto some fundamentally important idea. Additionally, a connection provide a basis both for sanity checking our ideas and also can be fruitful grounds for inspiration by seeing how variants of either the logic or the language are reflected through the connection. In this thesis we will mostly be concerned with how polarization of substructural logics and related concepts can help us design a logically motivated language for concurrent programming using session types.

1.2 Claims

The main claims of this thesis are the following:

- Polarization provides a natural way to describe a logically based session typing language with asynchronous communication while retaining a semantics that is reasonably implementable. Additionally, polarization gives us a way to smoothly integrate synchronous channels into SILL without needing a semantic extension.

- Polarization and Adjoint Logic combine very cleanly, giving SILL an ability to incorporate a variety of modalities with relatively little work. From a practical perspective, this gives SILL access to persistent processes and garbage collection for processes.

- We explore a trio of loosely related language extensions, and their logical connections, inspired by the above results: bundled message passing to reduce the number of communications performed by processes; racy programs, enabled by a select/epoll-like mechanism; and asynchronous receiving, a generalization of the basic asynchronous semantics.

- We have three different implementations of SILL: a simple but relatively full featured interpreter written in OCaml; a fragment of SILL as an embedded domain specific language in Haskell; and a cleaner version of the same in Idris.

- We show that Liquid Types and Session Types are compatible. This gives us one notion of a dependently session typed language.

1.3 Outline of Thesis

Chapter 2 covers background material needed for the remainder of this thesis. This should be a self-contained reference for those unfamiliar with the material, but contains no novel results itself. Contained are short references on the π-alculus, Session Types, Curry-Howard style connections between logics and type systems, Substructural Logics, Dependent Types, and Bidirectional Typechecking.

Chapter 3 describes an integration of Liquid Typing [82] into a session typed π-calculus. This provides a mechanism for increasing the expressivity of session types by enabling more precise descriptions of data communicated across channels in the system, all while maintaining the ability to perform type inference. These precise descriptions are enabled by adding a very restricted class of dependent types to the language.

Chapter 4 introduces SILL, a logically rooted language for concurrent computation. We first introduce polarized logic and the fragment of SILL based on it, showing a Preservation and Progress result as well as showing how the Curry-Howard interpretation of polarization gives us an ability to intermingle synchronous and asynchronous channels in a typed fashion. Next we study an alternate, message bundling semantics for SILL related to focused logics [9, 51] and show the expected Preservation and Progress results. Additionally, we present one solution to an open question on how to appropriately incorporate racy behavior into session typed programs by exploring the Curry-Howard implications of a result used in proof search, showing the expected progress and preservation results. Then we describe a counterpart of the standard asynchronous semantics that also enables receiving along a channel to be performed asynchronously. Along with the expected Preservation and Progress, we show that this semantics can be reduced to the construct introduced in the section on racy programs. Next we describe a simple integration of polymorphism into SILL along with Curry-Howard interpretations of quantification. Then we explore an integration of polarization and Adjoint Logic [77], which are surprisingly easy to combine, allowing us to integrate more modalities from substructural logic, e.g., the affine modality allows us to easily discard unneeded processes. We then explore a notion of subtyping on branches in SILL and show how it might be applied to some security concerns and limitations exposed while doing so. Lastly, we discuss alternate mechanisms for implementing forwarding and their theoretical and practical properties.

Chapter 5 covers some results needed for implementing SILL and then describes three different implementations. This first and fullest featured is a OCaml based interpreter that closely follows the development of SILL throughout this thesis. The second shows how recent, and powerful, advances in Haskell's type system allow us to expose much of the functionality of SILL directly in the language, including using Haskell's `do`-notation to provide a pleasant syntax. Lastly, we discuss an effort to implement SILL in Idris, whose richer type system provided a chance to prototype the Haskell version of SILL in a more traditional dependently typed language, and problems encountered during this.

Chapter 2

Background

2.1 π-calculus

The π-calculus [60] is a process algebra for modeling distributed computation. It uses synchronous channels to pass data (including channel names) between processes that execute in parallel. The π-calculus can be viewed as a wrapper providing these distributed communication constructs around some underlying language of data and computation. For the purposes of this thesis we will assume that the underlying language is a simple functional language. We will impose a few other requirements on this underlying language in chapter 3. The syntax of the π-calculus, along with some informal meaning, is presented in the following grammar, where x ranges over a set of data variables, e ranges over expressions in the underlying functional language, k ranges over a distinct set of channel names, τ is a type from the underlying functional language, P_i is a τ-indexed family of processes, and X ranges over a distinct set of definition variables.

$$
\begin{aligned}
P \quad ::= {} & 0 \mid P \| P \mid k!(e).P \mid k!(k).P \\
& \mid k?(x).P \mid k?(k).P \mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ P \mid (\nu k)P \mid k \triangleleft e.P_i \mid \mathsf{case}_\tau\ k \Rightarrow P_i \\
& \mid \mathsf{def}\ X(\vec{x}; \vec{k}) = P\ \mathsf{in}\ P \mid X(\vec{e}, \vec{k})
\end{aligned}
$$

Informally, 0 is the terminated process. The process $P_1 \| P_2$ is the processes P_1 and P_2 executing in parallel. The process $k!(e).P$ sends the result of e along k and then continues as P. The process $k_1!(k_2).P$ sends the channel k_2 over k_1 and then continues as P. The process $k?(x).P$ binds the next data value sent on k to x and then continues as P. The process $k_1?(k_2).P$ receives the next channel sent on k_1 and then continues as P. The process $\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ evaluates e and proceeds as P_1 or P_2 as appropriate. The process $(\nu k)P$ generates a fresh channel and binds it to k. The process $\mathsf{case}_\tau\ k \Rightarrow P_i$ receives a value, v, of type τ along k then proceeds as the corresponding P_v. The declaration $\mathsf{def}\ X(\vec{x}; \vec{k}) = P_1\ \mathsf{in}\ P_2$ defines X as process P_1 that can use the variables in scope along with those supplied by $\vec{x}$ and $\vec{k}$, binds the definition to X, and proceeds as P_2. The process $X(\vec{e}, \vec{k})$ calls the process defined by X and supplies it as arguments the evaluated $\vec{e}$ and $\vec{k}$. π-calculus semantics are traditionally given in terms of a transition semantics that assumes a structural congruence that brings together compatible send/receive instructions so that communication can occur. For more details on semantics see Yoshida's survey [99].

2.2 Session Types

Session Types [41, 99] were introduced to provide a static characterization of the temporal behavior of the π-calculus. They rule out some dangers present in the π-calculus like the nondeterminism possible with channels held by more than two processes and sending and receiving processes disagreeing over the type of data being communicated. The type system disallows these while still allowing for a high degree of expressiveness such as communicating channel names and heterogeneous channel usage. The syntax of session types, S, is given by the following grammar where t ranges over a set of type variable names, τ is a type from the underlying functional language, and S_i is a τ-indexed family of session types.

$$S ::= 1 \mid t \mid \mu t.S \mid \tau \wedge S \mid \tau \supset S \mid S \otimes S \mid S \multimap S \mid \oplus\{\tau_i : S_i\}_\tau \mid \&\{\tau_i : S_i\}_\tau$$

The informal meaning of these are as follows. 1 is the type of channels that will have no further communication. The types $\mu t.S$ and t allow us to construct (possibly infinite) recursive types. We treat types equirecursively (i.e., we identify a recursive type with its unfolding $\mu t.S = S[\mu t.S/t]$). The type $\tau \wedge S$ is that of a channel that sends a data value of type τ and then proceeds as S. The type $\tau \supset S$ is that of a channel that receives a data value of type τ and then proceeds as S. The type $S_1 \otimes S_2$ is that of a channel that sends a channel with type S_1 and then proceeds as S_2. The type $S_1 \multimap S_2$ is that of a channel that receives a channel with type S_1 and then proceeds as S_2. The type $\oplus\{\tau_i : S_i\}_\tau$ is that of a channel that sends a piece of data of type τ and then proceeds as the appropriate τ-indexed S_i. The type $\&\{\tau_i : S_i\}_\tau$ is that of a channel that receives a piece of data of type τ and then proceeds as the appropriate τ-indexed S_i. As with processes, our types have a notion of send/receive pairs. We define the notion of a dual type to encode this correspondence. The dual of a session type S is denoted $\overline{S}$ and defined below.

$$\overline{1} = 1 \qquad \overline{\tau \wedge S} = \tau \supset \overline{S} \qquad \overline{\tau \supset S} = \tau \wedge \overline{S}$$
$$\overline{S_1 \otimes S_2} = S_1 \multimap \overline{S_2} \qquad \overline{S_1 \multimap S_2} = S_1 \otimes \overline{S_2}$$
$$\overline{\oplus\{\tau_i : S_i\}_\tau} = \&\{\tau_i : \overline{S_i}\}_\tau \quad \overline{\&\{\tau_i : S_i\}_\tau} = \oplus\{\tau_i : \overline{S_i}\}_\tau$$

The session type system will use duality to match up compatible channel users. A channel typing is a mapping from channel variables to session types. We use $\Gamma = 1$ to denote that all bindings in Γ are 1. The last notation needed is for marking polarity. Polarity markings are superscripts on channel names that will allow us to distinguish the two conceptual "ends" of a channel so that we can rule out send/receive confusion and more than two processes using a channel

$$\frac{\Theta;\Psi\vdash_S P :: \Gamma,(k{:}S) \quad \Psi\vdash e:\tau}{\Theta;\Psi\vdash_S k!(e).P :: \Gamma,(k\wedge S)}\ \textsc{T.Send} \qquad \frac{\Theta;\Psi,x:\tau\vdash_S P :: \Gamma,(k:S)}{\Theta;\Psi\vdash_S k?(x).P :: \Gamma,(k{:}\,\tau\supset S)}\ \textsc{T.Rec}$$

$$\frac{\Gamma = 1}{\Theta;\Psi\vdash_S 0 :: \Gamma}\ \textsc{T.End} \qquad \frac{\Theta;\Psi\vdash_S P :: \Gamma,(k_1:S_1)}{\Theta;\Psi\vdash_S k_1!(k_2).P :: \Gamma,(k_1:S_2\otimes S_1),(k_2:S_2)}\ \textsc{T.Thr}$$

$$\frac{\Theta;\Psi\vdash_S P :: \Gamma,(k^p:S),(k^{\overline{p}}:\overline{S})}{\Theta;\Psi\vdash_S (\nu\ k)P :: \Gamma}\ \textsc{T.Nu} \qquad \frac{\Theta;\Psi\vdash_S P :: \Gamma,(k_1:S_1),(k_2:S_2)}{\Theta;\Psi\vdash_S k_1?(k_2).P :: \Gamma,(k_1{:}\,S_2\multimap S_1)}\ \textsc{T.Cat}$$

$$\frac{\Theta;\Psi\vdash_S P :: \Gamma,(k^+:G(X))}{\Theta;\Psi\vdash_S \mathsf{accept}\ X(k).P :: \Gamma}\ \textsc{T.Acc} \qquad \frac{\Theta;\Psi\vdash_S P :: \Gamma,(k^-:\overline{G(X)})}{\Theta;\Psi\vdash_S \mathsf{request}\ X(k).P :: \Gamma}\ \textsc{T.Req}$$

$$\frac{\Theta;\Psi\vdash_S P :: \Gamma_1 \quad \Theta;\Psi\vdash_S P :: \Gamma_2}{\Theta;\Psi\vdash_S P\|Q :: \Gamma_1,\Gamma_2}\ \textsc{T.Par}$$

$$\frac{\Psi\vdash e:\texttt{Bool} \quad \Theta;\Psi\vdash_S P :: \Gamma \quad \Theta;\Psi\vdash_S Q :: \Gamma}{\Theta;\Psi\vdash_S \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q :: \Gamma}\ \textsc{T.If}$$

$$\frac{\Psi\vdash e:\tau \quad \tau:\textsc{Enum} \quad \text{for } i\in\tau\text{: } \Theta;\Psi\vdash_S P_i :: \Gamma,(k:S_i)}{\Theta;\Psi\vdash_S k\triangleleft e.P_i :: \Gamma,(k:\oplus\{\tau_i:S_i\}_\tau)}\ \textsc{T.Int}$$

$$\frac{\tau:\textsc{Enum} \quad \text{for } i\in\tau\text{: } \Theta;\Psi\vdash_S P_i :: \Gamma,(k:S_i)}{\Theta;\Psi\vdash_S \mathsf{case}_\tau\ k\Rightarrow P_i :: \Gamma,(k:\&\{\tau_i:S_i\}_\tau)}\ \textsc{T.Ext}$$

$$\frac{\text{for } i\text{: } \tau\vdash e_i:\tau_i \quad \Gamma=1}{\Gamma,(X:(\vec{\tau},\vec{S}));\Psi\vdash_S X(\vec{e},\vec{k}) :: \Gamma,(\vec{k}:\vec{S})}\ \textsc{T.Call}$$

$$\frac{\Theta,(X:(\vec{\tau},\vec{S}));\Psi,(\vec{x}:\vec{\tau})\vdash_S P :: (\vec{k}:\vec{S}) \quad \Theta,(X:(\vec{\tau},\vec{S}));\Psi\vdash_S Q :: \Gamma}{\Theta;\Psi\vdash_S \mathsf{def}\ X(\vec{x};\vec{k}) = P\ \mathsf{in}\ Q :: \Gamma}\ \textsc{T.Def}$$

Figure 2.1: Typing Rules for Simple Session Types

at once. We use k^+ to denote the positive end of channel k, k^- to denote the negative "end", and $k^{\overline{p}}$ to denote swapping the polarity of k^p.

When moving the session typed setting we add two small syntactic extension. First, we add a pair keywords ($\mathsf{request}\ X(k).P$) and ($\mathsf{accept}\ X(k).P$) to initiate sessions, which, informally, match a process that requests a new session X with one that accepts its request. These two processes then communicate along a new channel bound to k within both of their continuation processes. The process $k \triangleleft e.P$ evaluates e and sends it along k then proceeds as P. This will be distinguished from $k!(e).P$ in the type system by allowing the the type of P to depend on the value of e.

Using the notions above we can give the rules for session types. We use Θ to denote a mapping from process variables to tuples of their argument types, Ψ to denote typings for our functional variables, and Γ to denote channel typings. We use Γ_1, Γ_2 to denote the merger of two channel typings that share no common bindings.[1] We use the hypothesis $\tau : \textsc{Enum}$ to denote that the type τ is a finite enumeration. Depending on the details underlying functional language this may have different interpretations. These enumerations could be smoothly generalized to algebraic datatypes, but we present only the simplified view to

[1]Differently polarized channel names are treated as distinct.

avoid unneeded clutter. The judgment $\Theta; \Psi \vdash_S P :: \Gamma$ denotes that, assuming the definitions of Θ and the functional types in Ψ, the free channel variables of process P have the session types in Γ. We use $\Psi \vdash e : \tau$ to denote that the type system for the underlying functional language proves that e has type τ from the assumptions in Ψ. We assume that sessions have some globally visible type and so assume a mapping, G, from session names to session types. Figure 2.1 contains a listing of the typing rules for simple session types. To see how the rules eliminate dangerous behavior consider the rule T.Nu. This rule ensures two things: the fresh channel has two and only two "ends"; the users of each end agree both in the direction of communication at every step and the type of value or channel being communicated.

2.3 Curry-Howard Connections

Curry-Howard connections [94, 95], used throughout this thesis, are connections between logical proof systems and programming languages. These function by noticing that propositions in a logic can be viewed as types in a programming language, and vice versa. These connections are most exciting when the logic and the connected programming language are discovered independently. In this case, it likely means that both the language and the logic are describing something that is in some sense important. However, even when this is not the case, we can still try to force a connection by picking a logic or a type system and asking "What would this look like as a programming language (or logic)?" Another advantage of a connection is the ability to use it as inspiration for explorations into linguistic changes. For example, a logic for which general identity is admissible in the fragment without identity (or only propositional identity), might show us how to simplify our programming language.

A great deal of forcing these connections is fairly mechanical and corresponds to finding a text editor friendly way to represent proof trees in the logical system. The first step for forming such a connection is to name each proposition used throughout the proof so that we can refer to it by name rather than by pattern matching. Next, each rule needs some syntax associated with it, containing at least enough information to ensure we know how the proof rule is used (e.g., by containing all the names of assumptions used by the rule). A bit of care should be applied in selecting suitably suggestive syntax, e.g., since this thesis will focus on concurrent languages communicating across channels, we will use send and recv to denote certain constructs rather than, say, juxtaposition and λ. The last step is to design a semantics that corresponds, often obviously, to proof normalization, generally cut-reduction. At that point we have a logically inspired language, if not a guarantee that we choose something useful at either the syntax or semantics step.

2.4 Substructural Logics

Traditional logics view their assumptions as *persistent*, i.e., they may be used as many or as few times as desired. This will stand in contrast to the substructural logics which will give us the ability to provide much finer grained control over the use (or non-use) of assumptions.

Before describing substructural logics, we need a working baseline of a structural logic with which they can be below. A simple structural logic can be presented by generating propositions from the following grammar:

$$\phi ::= \top \mid \bot \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \to \phi$$

And, then creating a judgment $\Phi \vdash \phi$, where Φ is a list of propositions that may be assumed to be true while proving ϕ. The judgement comes with a variety of *structural* proof rules for manipulating the list of assumptions:

$$\frac{\Phi_1, \phi_3, \phi_2, \Phi_4 \vdash \phi_5}{\Phi_1, \phi_2, \phi_3, \Phi_4 \vdash \phi_5}\ \textsc{Exchange} \qquad \frac{\Phi \vdash \phi'}{\Phi, \phi \vdash \phi'}\ \textsc{Weaken} \qquad \frac{\Phi, \phi, \phi \vdash \phi'}{\Phi, \phi \vdash \phi'}\ \textsc{Contract}$$

Often these structural rules are implicit in a structural logic's description , e.g., by defining Φ to be a set of assumptions rather than a list. Additionally, the logical connectives themselves come with proof rules, mostly one right rule to prove the connective and one left rule to utilize it:

$$\frac{}{\phi \vdash \phi}\ \textsc{Id} \qquad \frac{}{\Phi \vdash \top}\ \top R \qquad \frac{\Phi \vdash \phi}{\Phi, \top \vdash \phi}\ \top L \qquad \frac{}{\Phi, \bot \vdash \phi}\ \bot L$$

$$\frac{\Phi \vdash \phi_1 \quad \Phi \vdash \phi_2}{\Phi \vdash \phi_1 \wedge \phi_2}\ \wedge R \qquad \frac{\Phi, \phi_1, \phi_2 \vdash \phi_3}{\Phi, \phi_1 \wedge \phi_2 \vdash \phi_3}\ \wedge L \qquad \frac{\Phi \vdash \phi_i}{\Phi \vdash \phi_1 \vee \phi_2}\ \vee R_i$$

$$\frac{\Phi, \phi_1 \vdash \phi_3 \quad \Phi, \phi_2 \vdash \phi_3}{\Phi, \phi_1 \vee \phi_2 \vdash \phi_3}\ \vee L \qquad \frac{\Phi, \phi_1 \vdash \phi_2}{\Phi \vdash \phi_1 \to \phi_2}\ \to R \qquad \frac{\Phi \vdash \phi_1 \quad \Phi, \phi_2 \vdash \phi_3}{\Phi, \phi_1 \to \phi_2 \vdash \phi_3}\ \to L$$

Substructural logics omit at least one of the structural rules: Linear Logics [37] permits only Exchange; Affine Logics [70] allow Weaken and Exchange; and Ordered Logics [47] allow only for associativity of environment joining (i.e., environments are lists, not trees). To focus on the linear case, this restriction to only allowing Exchange means that assumptions can be modeled as multisets. Intiuitively, linearity means that we never discard or duplicate assumptions during a proof, in some sense treating them as consumable resources. However, since $\vee L$, $\wedge R$, and $\to L$ duplicate their assumptions, we will need to either decide to allow this or split Φ into a portion for each subproof. In practice, both of these are useful, so, instead, we will split our logical connectives into versions for each. These new connectives are generated by the following grammar:

$$A ::= A \otimes A \mid A \multimap A \mid 1 \mid 0 \mid A \oplus A \mid A \& A \mid \top$$

where: the proposition $A \otimes A$ represent a assumption splitting notion of conjunction; the proposition $A \multimap A$ is (linear) implication; the proposition 1 is a notion of true that is an identity for $\otimes$; the proposition 0 represents false; the proposition $A \oplus A$ represents a non-splitting notion of disjunction; $A \& A$ is a non-splitting conjunction; the proposition $\top$ is a notion of true that is the identity for &; and there is no version of a splitting disjunction.

$$\frac{}{A \vdash A}\ \textsc{Id} \qquad \frac{}{\vdash 1}\ 1R \qquad \frac{\Gamma \vdash A}{\Gamma, 1 \vdash A}\ 1L \qquad \frac{\Gamma_1 \vdash A_1 \quad \Gamma_2 \vdash A_2}{\Gamma_1, \Gamma_2 \vdash A_1 \otimes A_2}\ \otimes R$$

$$\frac{\Gamma, A_1, A_2 \vdash A_3}{\Gamma, A_1 \otimes A_2 \vdash A_3}\ \otimes L \qquad \frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \multimap A_2}\ \multimap R \qquad \frac{\Gamma_1 \vdash A_1 \quad \Gamma_2, A_2 \vdash A_3}{\Gamma_1, \Gamma_2, A_1 \multimap A_2 \vdash A_3}\ \multimap L$$

$$\frac{}{\Gamma \vdash \top}\ \top R \qquad \frac{}{\Gamma, 0 \vdash A}\ 0L \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 \oplus A_2}\ \oplus R_i \qquad \frac{\Gamma, A_1 \vdash A_3 \quad \Gamma, A_2 \vdash A_3}{\Gamma, A_1 \oplus A_2 \vdash A_3}\ \oplus L$$

$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash A_1 \& A_2}\ \& R \qquad \frac{\Gamma, A_i \vdash A_3}{\Gamma, A_1 \& A_2 \vdash A_3}\ \& L_i \qquad \frac{}{\Gamma \vdash \top}\ \top R$$

As we will discuss in more detail in subsection 4.6.1, linear logics often find a need to reintroduce peristent assumptions in a controlled fashion. One mechanism for this is to add a new connective:

$$A ::= \ldots \mid !A$$

and then adding rules for proving that a consequent is persistent (i.e., we can make as much as needed), commit to using it linearly if we want, and contracting or weaking it (here $!\Gamma$ means that all the assumptions in Γ are persistent):

$$\frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ !R \qquad \frac{\Gamma, A_1 \vdash A_2}{\Gamma, !A_1 \vdash A_2}\ !L \qquad \frac{\Gamma, \vdash A_2}{\Gamma, !A_1 \vdash A_2}\ !\textsc{Weak} \qquad \frac{\Gamma, !A_1, !A_1 \vdash A_2}{\Gamma, !A_1 \vdash A_2}\ !\textsc{Contract}$$

2.5 Dependent Types

Dependent Types [53], which we use in chapter 3, section 5.6, and section 5.7, are types that allow the meaning of types to depend on data values. Dependently typed functions explicitly bind their input values at the type level, just as they do at the program level, and generally allow for the usage of function at the type level. For example, a dependently typed programming language might allow us to define lists that explicitly carry their length at the type level, commonly called vectors, with the following code:

```
data Nat = Z | S Nat


data Vect :: a -> Nat -> Type where
  VNil :: (a:Type) -> Vect a Z
  VCons :: (a:Type) -> (n:Nat) -> a -> Vect a n -> Vect a (S n)
```

which says that vectors, `Vect`, define a type that takes a type parameter, `a`, and a natural number length. It has two constructors: the empty vector, `VNil`, which is a list with any element type and a vector with one extra element attached to the front. These two constructors have types that explicitly bind their inputs for later use, i.e., `VNil` takes as an argument the type of empty list it is and `VCons` takes both the type of its elements and the length of the list to prepend an element to. Notice that `VCons` also utilizes a type level function `S n` to ensure it has the correct return type. In practical systems [1, 11, 66], these type level arguments could be marked implicit in some fashion, with the hope that the typechecker will be able to determine them as needed (e.g., via unification).

In chapter 3, we will mostly be interested in are a restricted class of dependent types called refinement types. A refinement type is a basic type (i.e., a non-compound type—`int` but not `int->float`) with a predicate attached to it; e.g., the positive integers are given by $\{v : \texttt{int} | 0 \leq v\}$. Simple types can naturally be viewed as refinement types by using the trivial always-true predicate. From this follows a natural notion of subtyping (with the normal contravariance for functions). In addition to allowing predicates to incorporate constants, we will want them to allow for dependency on previously bound terms, e.g., $\{v : \texttt{int} | v \leq x\}$ for some previously bound x. For compound functional types we assume that refinements are available on the "leaf" types. Rondon [81] shows that for a certain class of predicates in ML we can maintain ML's guarantee of inferability while still allowing for extra expressiveness (e.g., they used their approach to check array bounds accesses through their types).

2.6 Bidirectional Type Checking

Bidirectional typechecking [76] is a scheme for working with type systems that exploits the idea that some expressions are easy to infer, or *synthesize*, the type of and some expressions are only easy to *check* that they have a given type. This provides a few advantages. First, it can incorporate features that may not have complete inference algorithms (e.g., dependent types [2, 7]). Second, because checking is done with the goal type in hand, bidirectional checking can provide easier reporting [74] of high-quality error messages. Additionally, a bidirectional presentation of a type system can, in the incomplete case, show us what kind of expressions require type annotations (i.e., when we are forced to synthesize a type for an expression we only know how to check).

To see this in action, consider the following simple functional language:

$$\begin{array}{lcll} \tau & ::= & \tau \to \tau \mid \texttt{Int} \mid \ \ldots & \text{(Types)} \\ x & ::= & \ldots & \text{(Variables)} \\ e & ::= & x \mid \mathsf{fun}\ x \to e \mid e + e \mid e\ e \mid \ \ldots & \text{(Expressions)} \end{array}$$

and some of its type system, where Ψ maps variables to types:

$$\frac{}{\Psi, x : \tau \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Psi, x : \tau \vdash e : \tau'}{\Psi \vdash \mathsf{fun}\ x \to e : \tau \to \tau'}\ \textsc{Fun} \qquad \frac{\Psi \vdash e_1 : \tau' \to \tau \quad \Psi \vdash e_2 : \tau'}{\Psi \vdash e_1\ e_2 : \tau}\ \textsc{App}$$

A bidirectional system for this language consists of two mutual recursive judgments: the judgment $\Psi \vdash e \Rightarrow \tau$ says, given the mapping Ψ, the expression e synthesizes type τ; the judgment $\Psi \vdash e \Leftarrow \tau$, confirms, given Ψ and τ, that e has the type τ. We add one metarule to mediate between these two judgments and classify our earlier type rules:

$$\frac{\Psi \vdash e \Rightarrow \tau}{\Psi \vdash e \Leftarrow \tau}\ \textsc{Synth} \qquad \frac{}{\Psi, x : \tau \vdash x \Rightarrow \tau}\ \textsc{Var}$$

$$\frac{\Psi, x : \tau \vdash e \Leftarrow \tau'}{\Psi \vdash \mathsf{fun}\ x \to e \Leftarrow \tau \to \tau'}\ \textsc{Fun} \qquad \frac{\Psi \vdash e_1 \Rightarrow \tau' \to \tau \quad \Psi \vdash e_2 \Leftarrow \tau'}{\Psi \vdash e_1\ e_2 \Rightarrow \tau}\ \textsc{App}$$

These directions can be justified by seeing what happens if we try to use each expression with the opposite judgement. Since Synth is a metarule, notice that inverting it requires us to guess τ. A checking version of Var or App is merely the fusion of Synth with Var or App, respectively. For Fun, we again would need to guess the type of x. To get an algorithm out of this, we view the synthesizing judgment as outputting its τ and the checking judgement as only outputting a boolean value. Due to the Synth rule, this is not quite syntax directed, but we can fix this by applying Synth only when no other rule applies.

This altered system is incomplete: there are some terms that have a sensible typing under the non-algorithmic judgment but cannot be checked or synthesized. To see this, consider the following judgment:

$$y : \texttt{Int} \vdash (\mathsf{fun}\ x \to x)\ y : \texttt{Int}$$

This is easily proven, but, due to our inability to synthesize types for anonymous functions, cannot be checked or synthesized. In some sense, the real problem is a lack of polymorphism preventing us from finding most general types, but this example also illustrates how we can easily know where the tricky parts of our type system are. If we want to avoid adding polymorphism, as some bidirectional systems for very powerful languages [30] have explored, we could instead add type annotations to function arguments. Of course, these would occasionally be redundant. An alternative fix would be to add general purpose type annotations to the language, e.g., extending expressions with $\langle e : \tau \rangle$ which denotes that e should have type τ. This gives us a way to write the counterpart of Synth without any guessing:

$$\frac{\Psi \vdash e \Leftarrow \tau}{\Psi \vdash \langle e : \tau \rangle \Rightarrow \tau}\ \textsc{Check}$$

Notice that this still does not quite fix our problem. However, it does enable the bidirectional type checking algorithm to provide a suggestion to the user so that they can change their program to the following provable judgment:

$$y : \texttt{Int} \vdash \langle \textsf{fun}\ x \to x : \texttt{Int} \to \texttt{Int} \rangle\ y \Leftarrow \texttt{Int}$$

Chapter 3

Value-Dependent Session Types

In this chapter, we pursue the integration of the dependent types of section 2.5 with session types.

3.1 Basics

Refined session types, Υ, are generated by the following grammar, where ρ denotes a refined simple type and Υ_i denotes a τ-indexed family of refined session types.

$$\Upsilon \quad ::= 1 \mid t \mid \mu t.\Upsilon \mid \rho \wedge \Upsilon \mid (x : \rho) \supset \Upsilon \mid \Upsilon \otimes \Upsilon \mid \Upsilon \multimap \Upsilon \mid \oplus_\tau \Upsilon_i \mid \&_\tau \Upsilon_i$$

These types are nearly the same as their simple counterparts but utilizing refined functional types instead of simple functional types. A construct that does change is $(x : \rho) \supset \Upsilon$. This construct allows refined session types to bind the data value that was sent across the channel and refer to this in later refined types. In particular, this allows for session types like $(\mathtt{x} : \{v : \mathtt{int} | \textsc{True}\}) \supset \{v : \mathtt{int} | v \geq \mathtt{x}\} \wedge 1$, which would be a refined session type for describing a process that receives an integer and then returns the absolute value of that integer. Why not provide more binders? For the sending of data there is no new value introduced, e could always be reconstructed in our refinement as needed, so there is nothing to bind. An additional practical consideration is that it is not obvious what variable to use to bind the result of e. For sending and receiving channels, we assume that the logic that section 3.2 uses cannot analyze channels and so has no need to refer to a received channel in our predicates. For the two choice constructs, there is no need to provide an explicit binding for the enumeration value chosen, the τ-indexed family of types can already implicitly use this knowledge.

We will need a few more definitions before introducing the typing rules for refined session types. First, $\rho\downharpoonleft$ is a refined type with all the refinement information striped out (e.g., $\{v : \mathtt{int} | 0 \leq v\}\downharpoonleft = \mathtt{int}$). This has a natural generalization to environments and typings. The notion of the dual of a session type is essentially unchanged except for the need to handle bindings during the reception of data, so we say that for any x, $\overline{\rho \wedge \Upsilon} = (x : \rho) \supset \overline{\Upsilon}$ and $\overline{(x : \rho) \supset \Upsilon} = \rho \wedge \overline{\Upsilon}$. Additionally, refinements introduce a notion of subtyping. We use $\Psi \vdash \rho_1 \sqsubseteq \rho_2$ to denote that ρ_1 is a subtype of ρ_2 under the assumptions in Ψ (defined by Rondon [81]) and

$$\frac{\Theta;\Psi \vdash_{SL} P :: \Gamma, k:\Upsilon \quad \Psi \vdash_L e:\rho \quad \Psi \vdash \rho \sqsubseteq \rho'}{\Theta;\Psi \vdash_{SL} k!(e).P :: \Gamma, (k:\rho' \wedge \Upsilon)}\ \textsc{R.Send}$$

$$\frac{\Theta;\Psi, x:\rho \vdash_{SL} P :: \Gamma, (k:\Upsilon) \quad \Psi \vdash \rho' \sqsubseteq \rho}{\Theta;\Psi \vdash_{SL} k?(x).P :: \Gamma, (k:(x:\rho') \supset \Upsilon)}\ \textsc{R.Rec}$$

$$\frac{\Theta;\Psi \vdash_{SL} P :: \Gamma, k_1:\Upsilon_1}{\Theta;\Psi \vdash_{SL} k_1!(k_2).P :: \Gamma, (k_1:\Upsilon_2 \otimes \Upsilon_1), (k_2:\Upsilon_2)}\ \textsc{R.Thr}$$

$$\frac{\Theta;\Psi \vdash_{SL} P :: \Gamma, (k_1:\Upsilon_1), (k_2:\Upsilon_2)}{\Theta;\Psi \vdash_{SL} k_1?(k_2).P :: \Gamma, (k_1:\Upsilon_2 \multimap \Upsilon_1)}\ \textsc{R.Cat}$$

$$\frac{\Theta;\Psi \vdash_{SL} P :: \Gamma, (k^p:\Upsilon), (k^{\overline{p}}:\overline{\Upsilon})}{\Theta;\Psi \vdash_{SL} (\nu\ k)P :: \Gamma}\ \textsc{R.Nu}$$

$$\frac{\Theta;\Psi \vdash_{SL} P :: \Gamma_1 \quad \Theta;\Psi \vdash_{SL} P :: \Gamma_2}{\Theta;\Psi \vdash_{SL} P \| Q :: \Gamma_1, \Gamma_2}\ \textsc{R.Par} \qquad \frac{\text{for } (k:\Upsilon) \in \Gamma\text{: } \Upsilon = 1}{\Theta;\Psi \vdash_{SL} 0 :: \Gamma}\ \textsc{R.End}$$

$$\frac{\Theta;\Psi \vdash_{SL} P :: \Gamma, (k^+ : G_L(X))}{\Theta;\Psi \vdash_{SL} \mathsf{accept}\ X(k).P :: \Gamma}\ \textsc{R.Acc} \qquad \frac{\Theta;\Psi \vdash_{SL} P :: \Gamma, (k^- : \overline{G_L(X)})}{\Theta;\Psi \vdash_{SL} \mathsf{request}\ X(k).P :: \Gamma}\ \textsc{R.Req}$$

$$\frac{\begin{array}{lll}\Psi \vdash_L e:\rho & \rho\rfloor = \mathsf{Bool} & \Theta;\Psi, e \vdash_{SL} P :: \Gamma_1 \\ \Theta;\Psi, \neg e \vdash_{SL} Q :: \Gamma_2 & \Psi, e \vdash \Gamma_1 \sqsubseteq \Gamma & \Psi, \neg e \vdash \Gamma_2 \sqsubseteq \Gamma\end{array}}{\Theta;\Psi \vdash_{SL} \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q :: \Gamma}\ \textsc{R.If}$$

$$\frac{\Psi \vdash_L e:\rho \quad \rho\rfloor : \textsc{Enum} \quad \text{for } i \in \rho\rfloor\text{: } \Theta;\Psi \vdash_{SL} P_i :: \Gamma, (k:S_i)}{\Theta;\Psi \vdash_{SL} k \triangleleft e.P_i :: \Gamma, k:\oplus\{\tau_i : S_i\}_\tau}\ \textsc{R.Int}$$

$$\frac{\tau : \textsc{Enum} \quad \text{for } i \in \tau\text{: } \Theta;\Psi \vdash_{SL} P_i :: \Gamma, (k:S_i)}{\Theta;\Psi \vdash_{SL} \mathsf{case}_\tau\ k \Rightarrow P_i :: \Gamma, k:\&\{\tau_i : S_i\}_\tau}\ \textsc{R.Ext}$$

$$\frac{\text{for i: } \Psi \vdash_L e_i : \rho_i' \quad \text{for i: } \Psi \vdash \rho_i' \sqsubseteq \rho_i \quad \text{for } (k:\Upsilon) \in \Gamma\text{: } \Upsilon = 1}{\Theta, X : (\vec{\rho}, \vec{\Upsilon});\Psi \vdash_{SL} X(\vec{e}, \vec{k}) :: \Gamma, \vec{k} : \vec{\Upsilon}}\ \textsc{R.Call}$$

$$\frac{\Theta, X : (\vec{\rho}, \vec{\Upsilon});\Psi, \vec{x} : \vec{\rho} \vdash_{SL} P :: (\vec{k} : \vec{\Upsilon}) \quad \Theta, X : (\vec{\rho}, \vec{\Upsilon});\Psi \vdash_{SL} Q :: \Gamma}{\Theta;\Psi \vdash_{SL} \mathsf{def}\ X(\vec{x};\vec{k}) = P\ \mathsf{in}\ Q :: \Gamma}\ \textsc{R.Def}$$

Figure 3.1: Type Rules for Refined Session Types

$\Psi \vdash \Upsilon_1 \sqsubseteq \Upsilon_2$ for subtyping of refined session types, defined below:

$$\frac{}{\Psi \vdash 1 \sqsubseteq 1} \qquad \frac{\Psi \vdash \rho_1 \sqsubseteq \rho_2 \quad \Psi \vdash \Upsilon_1 \sqsubseteq \Upsilon_2}{\Psi \vdash \rho_1 \wedge \Upsilon_1 \sqsubseteq \rho_2 \wedge \Upsilon_2}$$

$$\frac{\Psi \vdash \rho_1 \sqsubseteq \rho_2 \quad \Psi \vdash \Upsilon_1 \sqsubseteq \Upsilon_2}{\Psi \vdash (x:\rho_1) \supset \Upsilon_1 \sqsubseteq (x:\rho_2) \supset \Upsilon_2} \qquad \frac{\Psi \vdash \Upsilon_1 \sqsubseteq \Upsilon_3 \quad \Psi \vdash \Upsilon_2 \sqsubseteq \Upsilon_4}{\Psi \vdash \Upsilon_1 \otimes \Upsilon_2 \sqsubseteq \Upsilon_2 \otimes \Upsilon_3}$$

$$\frac{\Psi \vdash \Upsilon_3 \sqsubseteq \Upsilon_1 \quad \Psi \vdash \Upsilon_2 \sqsubseteq \Upsilon_4}{\Psi \vdash \Upsilon_1 \multimap \Upsilon_2 \sqsubseteq \Upsilon_3 \multimap \Upsilon_4} \qquad \frac{\Psi \vdash \text{for } i\text{: } \Upsilon_i \sqsubseteq \Upsilon_i'}{\Psi \vdash \oplus\{\tau_i : \Upsilon_i\}_\tau \sqsubseteq \oplus\{\tau_i : \Upsilon_i'\}_\tau}$$

$$\frac{\text{for all } i\text{: } \Psi \vdash \Upsilon_i \sqsubseteq \Upsilon_i'}{\Psi \vdash \&\{\tau_i : \Upsilon_i\}_\tau \sqsubseteq \&\{\tau_i : \Upsilon_i'\}_\tau}$$

Figure 3.1 introduces the typing rules for Refined Session Types. The judgment $\Theta;\Psi \vdash_{SL} P :: \Gamma$ denotes that, using the definitions of Θ and assumptions of

Ψ (many of which are just functional typing assignments), the free process channels of process P have the refined session types in Γ. The judgment $\Psi \vdash_L e : \rho$ denotes that, under the assumptions of Ψ, e has refined type ρ, the details of which depend on the underlying functional language. The rules are similar to the rules presented for unrefined session types but with the addition of subtyping information where appropriate. R.Send uses the idea that a process may transmit a subtype of its declared type and still maintain correct behavior. Conversely, R.Rec encodes that a process may use a looser approximation of its received data than required while still maintaining correctness. R.Nu remains "unchanged" for two reasons. First, the notion of duality has changed a bit, so an implicit change to handle refinements occurs. Second, while this would be a reasonable place to include subtyping information, the rules R.Send and R.Rec already account for this. Similarly, R.Call and not R.Def encapsulates the idea that definitions usage can accepted more tightly constrained types for a particular instance than they accept in general. Perhaps the most interesting rule is R.If. This rule makes refined session types path sensitive [4] by allowing for both branches to have different types and slightly different assumptions (e vs. $\neg e$) and then combining to have one unified typing for the whole process.

The type system for refined session types has a close connection with the simple session types as exhibited by the following lemma.

Lemma 1 (Judgement Correspondence)**.** *For refined definition environment Θ, refined functional assumptions Γ, process P and refined channel environment Γ, $\Theta; \Psi \vdash_{SL} P :: \Gamma$ implies $\Theta\lfloor; \Psi\lfloor \vdash_S P :: \Gamma\lfloor$. For simple definition environment Θ_1, simple functional environment Ψ_1, and simple channel typing Γ_1, $\Theta_1; \Psi_1 \vdash_S P :: \Delta_1$ implies there exists Θ_2, Ψ_2, and Δ_2 s.t. $\Theta_2; \Psi_2 \vdash_{SL} P :: \Gamma_2$ and $\Theta_2\lfloor = \Theta_1$, $\Psi_2\lfloor = \Psi$, and $\Gamma_2\lfloor = \Gamma_1$.*

Proof. Both proofs proceed by induction on the size of proof trees. For the first result, notice that by dropping all the refinement information (and subtyping) each of the refined session type rules becomes a simple session typing rule. For the second result, use the trivial always-true predicate to (not) constrain the types. $\square$

3.2 Inferencing

Inferring arbitrary refinement predicates is undecidable in general (consider trying to infer the type of a function that generates random primes) so we will restrict our attention significantly. In particular, we will fix some set of basic predicates and then infer predicates that are finite conjunctions drawn from this set. For example, if wishing to infer simple interval properties, we might have a set of predicates like $\{v \leq 5, v \leq \mathsf{x}, \mathsf{y} \leq v, \ldots\}$. Following [81], we will assume that this set is generated by a finite set of templates instantiated by program variables. We then look for conjunctions of ground substitutions for these templates that are suitable solutions to our constraints.

Inferring refined session types proceeds in three major steps:

1. Infer simple types and record some information from doing so
2. Add predicate variables to types and gather constraints on them
3. Solve these constraints

3.2.1 Simple Types

Inferring simple types is done by utilizing prior work [4, 26, 41]. In particular, we assume that for our functional language we can infer simple types. During this inferencing we will need to record a bit of extra information. Specifically, we will assume that the simple session type inferencing algorithm annotates channel generation with the channel's session type. Because of polarity considerations there is not a single type for a channel but two dual types, one for each end. For presentational compactness, we will assume that $(\nu k)P$ is annotated to become $(\nu\ k : S)P$ were S was the type of k^+, found during inferencing. Additionally, we will assume that parallel compositions are annotated with how to split the combined channel typing environment for the process into one typing for each of the two subprocesses. We will denote this split by converting $P_1 \| P_2$ into $P_1{}_{K_1}\|_{K_2} P_2$ with the names of K_i being those for P_i. Last, we assume that definitions are annotated with their argument types. That is $\mathsf{def}\ X(\vec{x}; \vec{k}) = P_1\ \mathsf{in}\ P_2$ becomes $\mathsf{def}\ X(\vec{x}; \vec{k}) : (\vec{\tau}; \vec{S}) = P_1\ \mathsf{in}\ P_2$. With these annotations we will be able to calculate at any point the simple channel typing of a subprocess of the process that we are trying to infer types. A more complicated implementation might be able to cache information closer to its use location, but we think these annotations provide a good trade-off between clarity and completeness.

3.2.2 Constraints

We utilize constraints of the following forms during constraint generation. $\Psi \vdash_{\text{wf}} \Upsilon$ indicates that Υ is well-formed w.r.t. Ψ, i.e., that the free variables in Υ are bound in Γ, ($\textsc{FreeNames}(\Upsilon) \subseteq \text{dom}(\Psi)$). Additionally, we use subtyping requirements of the form $\Psi \vdash \Upsilon_1 \sqsubseteq \Upsilon_2$ and $\Psi \vdash \rho_1 \sqsubseteq \rho_2$. The constraint $\Upsilon_1 = \overline{\Upsilon_2}$ is used to enforce duality. We also lift our constraints to work on (equal length) vectors of types pointwise (e.g., $\Psi \vdash_{\text{wf}} \vec{\rho}$ is equivalent to $\bigcup\{\Psi \vdash_{\text{wf}} \rho_i\}$).

We assume that we have some constraint generation algorithm that will produce correct constraints for our underlying functional language [81]. Armed with this we can read our typing rules as generating constraints by inserting subtyping constraints as appropriate (and in the case of T.Nu a duality constraint). Throughout the process of constraint gathering we will occasionally need to generate new refined session types with predicate variables, we denote this by $\textsc{Fresh}(\tau)$ for basic types and $\textsc{Fresh}(S)$ for session types. Whenever we perform this generation, we will provide some well-formedness constraint in addition to any subtyping constraints generated by the typing rules.

As an example consider the rule R.Send. Suppose that we know $(\Theta\lfloor; \Psi\lfloor \vdash_{SL} k!(e).P :: \Gamma, (k : \tau \wedge S))$ from our simple inference step. When we generate constraints for this, we will make one call to our functional constraint generation algorithm $(\Psi \vdash_L e : \rho)$, one recursive call to our session type constraint generation algorithm $(\Theta; \Psi \vdash_{SL} P :: \Gamma, (k : \Upsilon))$, generate one refined type $\textsc{Fresh}(\tau)$, and add the constraints $\Psi \vdash \rho \sqsubseteq \textsc{Fresh}(\tau)$ and $\Psi \vdash_{\text{wf}} \textsc{Fresh}(\tau)$, which corresponds to the constraints imposed by the typing rule.

Figure 3.2 provides a listing of the constraint generation algorithm for refined session types. To avoid confusion between tuple construction and map combining we denote the later with $\cdot$ in the algorithm listing. $\textsc{Constr}_{\text{SL}}(\Theta, \Psi, P, \Gamma_S)$ returns (Γ, C) a pair of refined channel typing (with predicate variables) and a set of constraints. We use $\textsc{Constr}_{\text{L}}(\Psi, e)$ to denote the assumed constraint gatherer of our underlying functional language. The algorithm assumes that, for functional assumptions Ψ, $\textsc{Constr}_{\text{L}}(\Psi, e)$ returns (ρ, C) a pair of a refined functional type and a set of constraints (both well-formedness and subtyping). A small abuse of notation occurs in the case for terminated processes and in process variable definition. Specifically, we use Γ_S as both a simple session typing and as a refined one. From our typing rules we know in both cases it must be entirely composed of mappings of the form $(k : 1)$ and so can be reasonably used in both contexts. While most of the cases used to define $\textsc{Constr}_{\text{SL}}$ are relatively straightforward, we highlight a few rules here.

Consider the case for conditional branching, perhaps the most complicated case. First we make recursive calls with the altered assumptions, allowing for sensitivity to the value of e. From R.If we know that both of the typings returned by these must be subtypes of our overall typing. Since we do not have a preexisting typing use for this subtyping we have to generate one $(\textsc{Fresh}(\Gamma_S(k)))$. We then return this typing along with our recursively generated constraints and three new constraints for each channel in our typing. The first new constraint ensures that our freshly generated types are well-formed. The other encodes the subtyping present in the rule. One might worry that if both k^p and $k^{\overline{p}}$ appear in our typing that this might cause them to become delinked. Since we will only use our constraint generation on closed processes after simple session type inferencing we know that these paired channel ends will eventually be generated by some $((\nu\ k)P)$ and thus duality will be ensured there.

The following lemma gives us the correctness of our constraint generation algorithm.

Lemma 2 (Constraint Correctness)**.** *For a closed annotated process P, empty definition and functional environments and simple typing,* $\textsc{Constr}_{SL}(\emptyset, \emptyset, P, \emptyset)$ *returns* (Γ, C)*, s.t.* $\emptyset; \emptyset \vdash_{SL} P :: \Gamma$ *if and only if C has a solution.*

Proof. Induction on the proof of $\Theta; \Psi \vdash_{SL} P :: \Gamma$. The result is a corollary of this induction. $\square$

3.2.3 Solving

Once all constraints have been generated, we will have many predicate variables left. A solution to a system of constraints is a ground substitution for predicate variables such that all constraints are satisfied. Assuming that our constraints allow all legal solutions (Lemma 2), we know that there is at least one possible solution, the trivial always-true solution. The important question is then that of finding a maximally specific solution. We search for a maximal solution using the normal implication ordering lifted to maps (i.e., $\sigma_1 \geq \sigma_2 \iff \forall x.\sigma_1(x) \implies \sigma_2(x)$).

A first pass removes all duality constraints by performing the substitutions implied by the equations. Since all Ψ in our constraints are finite and every predicate variable has at least one well-formedness constraint, we know that for any given predicate variable, there can be at most a finite number of ground substitutions admissible by its well-formedness constraints. This, together with the observation that only the predicate variables mentioned in our constraints matter for a substitution's admissibility, ensures we have only a finite number of "interesting" substitutions that might be solutions. Assuming that we can decide admissibility and solution ordering (e.g., via an SMT solver) then we can just try all solutions and select a maximal one. This requirement for being able to decide ordering is perhaps the biggest constraint on what we can choose as our templates, since we need to stay away from choosing those that are incompatible with our choice of SMT solver.

This proposed solution process is unsatisfyingly slow, so we instead suggest using Iterative Weakening [81]. Iterative Weakening is a technique that starts from the strongest admissible ground substitution (for each predicate variable a conjunction of all predicates admissible by its well-formedness constraints) and iteratively removes an offending conjunct that prevents us from satisfying all constraints. Since we deal with conjunctions of instantiated templates we know that removing a conjunct can at most preserve a substitution's strength and the always-true substitution is a solution, we know that iterative weakening will find a maximally specific solution. From the above arguments we have the following lemma.

Lemma 3 (Solver Correctness)**.** *For a given set of constraints, s.t. every predicate variable has at least one (finite) well-formedness constraint, iterative weakening produces a maximally specific solution.*

Sketch. Outlined above, this is proven by a generalization of Rondon [81]. $\square$

```
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, 0, \Gamma_S) = (\Gamma_S, \emptyset)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, \mathsf{accept}\ X(k).P, \Gamma_S) =$
    $(\Gamma \cdot (k^+ : \Upsilon), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P, \Gamma \cdot (k^+ : G(X)))$
    $\mathsf{return}\ (\Gamma, C \cup \{\Psi \vdash \Upsilon \sqsubseteq G_L(X)\}$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, \mathsf{request}\ X(k).P, \Gamma_S) =$
    $(\Gamma \cdot (k^- : \Upsilon), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P, \Gamma \cdot (k^- : \overline{G(X)}))$
    $\mathsf{return}\ (\Gamma, C \cup \{\Psi \vdash \Upsilon \sqsubseteq \overline{G_L(X)}\}$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, k!(e).P, \Gamma_S \cdot (k : \tau \wedge S)) =$
    $(\Gamma \cdot k : \Upsilon, C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P, \Gamma_S \cdot (k : S))$
    $(\rho, C') \leftarrow \textsc{Constr}_{\mathrm{L}}(\Psi, e)$
    $\rho' \leftarrow \textsc{Fresh}\tau$
    $\mathsf{return}\ (\Gamma \cdot k : \rho' \wedge \Upsilon, C \cup C' \cup \{\Psi \vdash_{\mathrm{wf}} \rho'; \Psi \vdash \rho \sqsubseteq \rho'\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, k?(x).P, \Gamma_S \cdot (k : \tau \supset S)) =$
    $\rho \leftarrow \textsc{Fresh}\tau$
    $\rho' \leftarrow \textsc{Fresh}\tau$
    $(\Gamma \cdot k : \Upsilon, C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi \cdot x : \rho, P, \Gamma_S \cdot (k : S))$
    $\mathsf{return}\ (\Gamma \cdot k : (x : \rho') \supset \Upsilon, C \cup \{\Psi \vdash_{\mathrm{wf}} \rho; \Psi \vdash_{\mathrm{wf}} \rho'; \Psi \vdash \rho \sqsubseteq \rho'\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, k_1!(k_2).P, \Gamma_S \cdot (k_1 : S_2 \otimes S_1) \cdot (k_2 : S_2)) =$
    $(\Gamma \cdot k_1 : \Upsilon_1, C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P, \Gamma_S \cdot (k_1 : S_1))$
    $\Upsilon_2 \leftarrow \textsc{Fresh}S_2$
    $\mathsf{return}\ (\Gamma \cdot k_1 : \Upsilon_2 \otimes \Upsilon_1, C \cup \{\Psi \vdash_{\mathrm{wf}} \Upsilon_2\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, k_1?(k_2).P, \Gamma_S \cdot (k_1 : S_2 \multimap S_1)) =$
    $(\Gamma \cdot (k_1 : \Upsilon_1) \cdot (k_2 : \Upsilon_2), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P, \Gamma_S \cdot (k_1 : S_1) \cdot (k_2 : S_2))$
    $\mathsf{return}\ (\Gamma \cdot (k_1 : \Upsilon_2 \multimap \Upsilon_1), C)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P_1{}_{K_1}\|_{K_2}P_2, \Gamma_S) =$
    $(\Gamma_1, C_1) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P_1, \Gamma_S \restriction_{K_1})$
    $(\Gamma_2, C_2) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P_2, \Gamma_S \restriction_{K_2})$
    $\mathsf{return}\ (\Gamma_1 \cdot \Gamma_2, C_1 \cup C_2)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, k \triangleleft e.P_i, \Gamma_S \cdot (k : \oplus\{\tau_i : S_i\}_\tau)) =$
    for $i$: $(\Gamma \cdot (k : \Upsilon_i), C_i) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi \cdot (e = i), P_i, \Gamma_S \cdot (k : S_i))$
    $\mathsf{return}\ (\Gamma \cdot (k : \oplus\{\tau_i : \Upsilon_i\}_\tau), \bigcup C_i)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, \mathsf{case}_k\ e \Rightarrow P_i, \Gamma_S \cdot (k : \&\{\tau_i : S_i\}_\tau)) =$
    for $i$: $(\Gamma \cdot (k : \Upsilon_i), C_i) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi \cdot (e = i), P_i, \Gamma_S \cdot (k : S_i))$
    $\mathsf{return}\ (\Gamma \cdot (k : \oplus\{\tau_i : \Upsilon_i\}_\tau), \bigcup C_i)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, \Gamma_S) =$
    $(\Gamma_1, C_1) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi \cdot e, P_1, \Gamma_S)$
    $(\Gamma_2, C_2) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi \cdot (\neg e), P_2, \Gamma_S)$
    for $k \in \mathrm{dom}(\Gamma_S)$: $\Upsilon_k \leftarrow \textsc{Fresh}\Gamma_S(k)$
    $\mathsf{return}\ \left(\vec{k} : \vec{\Upsilon}, C_1 \cup C_2 \cup \bigcup_{k \in \mathrm{dom}(\Gamma)} \left\{\begin{array}{l} \Psi \vdash_{\mathrm{wf}} \Upsilon_k; \\ \Psi \cdot e \vdash \Gamma_1(k) \sqsubseteq \Upsilon_k; \\ \Psi \cdot (\neg e) \vdash \Gamma_2(k) \sqsubseteq \Upsilon_k \end{array}\right\}\right)$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, (\nu\ k : S)P, \Gamma_S) =$
    $(\Gamma \cdot (k^+ : \Upsilon_1) \cdot (k^- : \Upsilon_2), C) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, P, \Gamma \cdot (k^+ : S) \cdot (k^- : \overline{S}))$
    $\mathsf{return}\ (\Gamma, C \cup \{\Upsilon_1 = \overline{\Upsilon_2}\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta \cdot (X : (\vec{\rho}, \vec{\Upsilon})), \Psi, X(\vec{e}, \vec{k}), \Gamma_S \cdot (\vec{k} : \vec{S})) =$
    for $i$: $(\rho'_i, C_i) \leftarrow \textsc{Constr}_{\mathrm{L}}(\Psi, e_i)$
    $\mathsf{return}\ (\Gamma_S \cdot (\vec{k} : \vec{\Upsilon}), \bigcup C_i \cup \{\Psi \vdash \vec{\rho'} \sqsubseteq \vec{\rho}\})$
$\textsc{Constr}_{\mathrm{SL}}(\Theta, \Psi, \mathsf{def}\ X(\vec{x}; \vec{k}) : (\vec{\tau}; \vec{S}) = P_1\ \mathsf{in}\ P_2, \Gamma_S \cdot (\vec{k} : \vec{S})) =$
    $(\vec{\rho}; \vec{\Upsilon}) \leftarrow (\textsc{Fresh}\vec{\tau}; \textsc{Fresh}\vec{S})$
    $(\Gamma_1 \cdot (\vec{k} : \vec{\Upsilon}'), C_1) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta \cdot (X : (\vec{\rho}; \vec{\Upsilon})), \Psi \cdot (\vec{x} : \vec{\rho}), P_1, (\vec{k} : \vec{S}))$
    $(\Gamma_2, C_2) \leftarrow \textsc{Constr}_{\mathrm{SL}}(\Theta \cdot (X : (\vec{\rho}; \vec{\Upsilon}')), \Psi, P_2, \Gamma_S)$
    $\mathsf{return}\ (\Gamma_2, C_1 \cup C_2 \cup \{\Psi \vdash_{\mathrm{wf}} \vec{\rho}; \Psi \vdash_{\mathrm{wf}} \vec{\Upsilon}; \Psi \vdash \vec{\Upsilon}' \sqsubseteq \vec{\Upsilon}\})$
```

Figure 3.2: Constraint Generation Algorithm

Chapter 4

Curry-Howard Session Types

4.1 Polarization

A way to design concurrent and distributed programs is as a collection of processes communicating over channels connecting them. To be sensibly behaved these processes need to have some notion of how communication across these channels will proceed. For example, a channel between two processes, A and B, might want to ensure that A sends `Ints` only when B expects to receive them. One mechanism that has been successful for describing channels in such systems is Session Types [41]. Session types ensure that all channels are used by exactly two processes and that each of these processes agrees both on what will be sent over the channel and which of the two processes will send and which will receive for every exchange over the lifetime of the program.

Session Types can be connected to Intuistionistic Linear Logic [72, 90] via a Curry-Howard style connection that associates propositions with types, proofs with programs, and computation with proof normalization. As mentioned in section 2.3, such connections both provide sanity checking and suggest new directions for language or logic design by transporting results and other innovations from one domain to the other, as is done in this thesis. With these in mind SILL will exploit various logical features to describe and inspire its own features.

4.1.1 Polarized Intuistionistic Linear Logic

To start with, we introduce the multiplicative and additive fragment of Intuitionistic Linear Logic [48, 58]. Like other linear logics, this is a substructural logic (section 2.4), i.e., it gives up on some of the normal structural rules on hypothesises. Propositions are split into two *polarities*, positive and negative propositions, and are generated by the following grammar, where τ is some underlying set of basic propositions, indented to be basic data types after a Curry-Howard connection (e.g., τ might contain `Int` or `Float`):

$$\begin{aligned} A^+, B^+, C^+ \quad &::= 1 \mid \tau \wedge A^+ \mid A^+ \otimes A^+ \mid A^+ \oplus A^+ \mid {\downarrow}A^- \\ A^-, B^-, C^- \quad &::= \tau \supset A^- \mid A^+ \multimap A^- \mid A^- \& A^- \mid {\uparrow}A^+ \end{aligned}$$

When the polarity of a proposition is unimportant we will write unannotated A, B, or C.

Propositions are connected with (session) types. These types describe the behavior of each channel (i.e., the provider of the channel sending or receiving a particular kind of message along it). Our system, following the tradition of session types, will force channels to connect exactly two processes, i.e., we do not allow for broadcast communications (with some extensions along those lines in section 4.6). In principle, we could give a type to channel from the client of the channel's dual perspective (perhaps erasing the notion of provider and client entirely), and some systems do [41], but we will not pursue that route futher.

The session types have the following informal meanings: the type 1 represents a channel that will terminate; the type $A^+ \otimes B^+$ is the channel that sends a channel of type A^+ and then continues as B^+; the type $\tau \wedge A^+$ is a channel that sends a value of type τ and continues as A^+; the type $\tau \supset A^-$ expects to receive a value of type τ and then behaves as A^-; the type $A^+ \multimap B^-$ is the type of channel that expects to receive a channel of type A and then continue as B^-; while the type $A^+ \oplus B^+$ is the channel that sends its choice between behaving as A^+ and B^+ in the future; and $A^- \& B^-$ is the type of a channel that expects to receive a choice to behave as either A^- or B^- in the future.

Since we will wish to model potentially infinite interactions along channels (e.g., a long running webserver), we need to include some ability to create infinite propositions. Perhaps the simplest would be to just allow infinite types (i.e., assume our grammar only specifies local acceptability of finite terms), but this leaves us with a problem of how to finitarily represent our propositions. For concreteness, we will add a least fixed point operator to our propositions, augmenting our grammar as follows, where x is drawn from a countably infinite set of variables:

$$A, B, C ::= \ldots \mid \mu x.A \mid x$$

We assume that propositions formed with this fixed point operator are contractive [35] (e.g., not of the form $\mu x.x$) and are closed. Additionally, we take an equirecursive view of these propositions, i.e., $\mu x.A = A[\mu x.A]$ without the need to explicitly unfold the μ. Occasionally, we will want to be more explicit about our treatment of μ and will utilize an explicit UNFOLD operator to perform any required substitutions in those situations. Additionally, we will sometimes specify propositions, particularly when viewed as types, by giving a recursive equation. As an example, the type of a channel that sends an unending stream of `Int`s might be defined by this equation:

$$\texttt{Stream} = \mu x.\texttt{Int} \wedge x$$

and a stream that provides an ability to either generate the next element of the

$$\frac{}{\Psi; A \vdash A}\ \textsc{Id} \qquad \frac{\Psi; \Gamma \vdash B \quad \Psi; \Gamma', B \vdash A}{\Psi; \Gamma, \Gamma' \vdash A}\ \textsc{Cut} \qquad \frac{}{\Psi; \emptyset \vdash 1}\ 1R$$

$$\frac{\Psi; \Gamma \vdash A}{\Psi; \Gamma, 1 \vdash A}\ 1L \qquad \frac{\Psi; \Gamma \vdash A^+ \quad \Psi; \Gamma' \vdash B^+}{\Psi; \Gamma, \Gamma' \vdash A^+ \otimes B^+}\ \otimes R \qquad \frac{\Psi; \Gamma, A^+, B^+ \vdash C}{\Psi; \Gamma, A^+ \otimes B^+ \vdash C}\ \otimes L$$

$$\frac{\Psi; \Gamma, A^+ \vdash B^-}{\Psi; \Gamma \vdash A^+ \multimap B^-}\ \multimap R \qquad \frac{\Psi; \Gamma \vdash A^+ \quad \Psi; \Gamma', B^- \vdash C}{\Psi; \Gamma, \Gamma', A^+ \multimap B^- \vdash C}\ \multimap L$$

$$\frac{\Psi; \Gamma \vdash A^- \quad \Psi; \Gamma \vdash B^-}{\Psi; \Gamma \vdash A^- \& B^-}\ \& R \qquad \frac{\Psi; \Gamma, A^- \vdash C}{\Psi; \Gamma, A^- \& B^- \vdash C}\ \& L_1 \qquad \frac{\Psi; \Gamma, B^- \vdash C}{\Psi; \Gamma, A^- \& B^- \vdash C}\ \& L_2$$

$$\frac{\Psi; \Gamma \vdash A^+}{\Psi; \Gamma \vdash A^+ \oplus B^+}\ \oplus R_1 \qquad \frac{\Psi; \Gamma \vdash B^+}{\Psi; \Gamma \vdash A^+ \oplus B^+}\ \oplus R_2 \qquad \frac{\Psi; \Gamma, A^+ \vdash C \quad \Gamma, B^+ \vdash C}{\Psi; \Gamma, A^+ \oplus B^+ \vdash C}\ \oplus L$$

$$\frac{\Psi \vdash \tau \quad \Psi, \Gamma \vdash A^+}{\Psi, \Gamma \vdash \tau \wedge A^+}\ \wedge R \qquad \frac{\Psi, \tau; \Gamma, A^+ \vdash B}{\Psi; \Gamma, \tau \wedge A^+ \vdash B}\ \wedge L$$

$$\frac{\Psi, \tau; \Gamma \vdash A^-}{\Psi; \Gamma \vdash \tau \supset A^-}\ \supset R \qquad \frac{\Psi \vdash \tau \quad \Psi; \Gamma, A^- \vdash B}{\Psi; \Gamma, \tau \supset A^- \vdash B}\ \supset L$$

$$\frac{\Psi; \Gamma \vdash A^-}{\Psi; \Gamma \vdash {\downarrow} A^-}\ {\downarrow} R \qquad \frac{\Psi; \Gamma, A^- \vdash B}{\Psi; \Gamma, {\downarrow} A^- \vdash B}\ {\downarrow} L \qquad \frac{\Psi; \Gamma \vdash A^+}{\Psi; \Gamma \vdash {\uparrow} A^+}\ {\uparrow} R \qquad \frac{\Psi; \Gamma, A^+ \vdash B}{\Psi; \Gamma, {\uparrow} A^+ \vdash B}\ {\uparrow} L$$

Figure 4.1: Judgment for Polarized Intuitionistic Linear Logic

stream or stop might have the type:

$$\texttt{GStream} = \mu x. 1 \oplus (\texttt{Int} \wedge x)$$

Since explicitly naming μ-variables is somewhat tedious, we will generally write recurse types equationally, as in traditional functional languages. E.g., the type `GStream` could also be written as:

$$\texttt{GStream} = 1 \oplus (\texttt{Int} \wedge \texttt{GStream})$$

and then converted into the prior definition as needed.

Proofs of our propositions are formed from the judgement $\Psi; \Gamma \vdash A$, where Ψ is some set of assumptions from for our underlying types (i.e., for τs) equipped with its own judgment ($\Psi \vdash \tau$), Γ is a multiset of linear assumptions (i.e., As), and A is the goal proposition to prove. We assume that the normal structural rules hold for Ψ and only Exchange holds for Γ (implicit in specifying it as a multiset). The proof rules for this logic are given in Figure 4.1.

4.1.2 Cutting Apart Cut

Since Cut is a relatively complicated and metatheoretically important rule, let us consider it a bit more detail. Informally, cut consists of two operations: specifying a subproof to import and then adding the result of that import to the assumptions (possibly consuming some existing assumptions during the importing step). If we had a proposition for "A is provable from Ψ and Γ" we could split Cut into its two constituent operations. We will write this proposition as $\{A \leftarrow \vec{A}\}$ where A is the consequent of the judgment and $\vec{A}$ is some linearization of Γ. Since Γ is a multiset, the exact linearization is in some sense irrelevant, but will provide nicer syntax for the Curry-Howard style language we are driving towards. To avoid a proliferation of proposition kinds, we will assume that τ is extended with types of this form:

$$\tau ::= \ldots \mid \{A \leftarrow \vec{A}\}$$

Which comes with the following rules:

$$\frac{\Psi; \vec{A} \vdash A}{\Psi \vdash \{A \leftarrow \vec{A}\}}\ \{\}I \qquad \frac{\Psi \vdash \{B \leftarrow \vec{B}\} \quad \Psi; \Gamma, B \vdash A}{\Psi; \vec{B}, \Gamma \vdash A}\ \{\}E$$

Together, these allow us to reproduce Cut as a derived rule:

$$\frac{\overbrace{\Psi; \vec{A} \vdash C}^{\mathcal{A}} \quad \overbrace{\Gamma, C \vdash B}^{\mathcal{B}}}{\Psi; \vec{A}, \Gamma \vdash B}\ \textsc{Cut} \quad \longrightarrow \quad \frac{\dfrac{\overbrace{\Psi; \vec{A} \vdash C}^{\mathcal{A}}}{\Psi \vdash \{C \leftarrow \vec{A}\}}\ \{\}I \quad \overbrace{\Psi; \Gamma, C \vdash B}^{\mathcal{B}}}{\Psi; \vec{A}, \Gamma \vdash B}\ \{\}E$$

4.1.3 Syntax

The process of going from our proof rules to a type system for a language in the Curry-Howard style is has a few key steps [90]: we name all the assumptions used by each proof rule; create suggestive syntax for each proof rule, to permit writing proof trees in a fashion suitable for use with a text editor; and then write semantics to give an operational meaning to our syntax. Both of our kinds of propositions, τ and A, will correspond to some set of terms, e and P, respectively. We assume that channel names, c, are drawn from some countably infinite set of names and that x captures the variable names used in e. To maintain flexibility, we will leave e mostly unspecified (other than the syntax for $\{\}I$) and assume it corresponds to some simple functional language. To economize on syntax, we exploit the duality of our typing rules and say that each construct of P that corresponds to a particular kind of communication is differentiated between being the appropriate left or right rule by the by whether it uses the process's provided channel (this will be clearer in the full type listing Figure 4.3). Since the constructs of P will have an imperative flavor, we will occasionally refer to the outermost construct of a process expression as an instruction or command. The grammar for this language is presented in Figure 4.2 with comments to the side indicating to which rule(s) each construct corresponds.

$$
\begin{array}{llll}
a, b, c, d, f, g & ::= & \ldots & \text{(Channels names)} \\
x, y, z & ::= & \ldots & \text{(Variable names)} \\
e & ::= & \ldots \mid c \leftarrow \{P\} \mathrel{-\!\!\prec} \vec{c} & (\{\}I) \\
P, Q, R & ::= & c \leftarrow e \mathrel{-\!\!\prec} \vec{c} & (\{\}E) \\
 & \mid & c \leftarrow c & (\textsc{Id}) \\
 & \mid & \mathsf{wait}\ c; P & (1L) \\
 & \mid & \mathsf{close}\ c & (1R) \\
 & \mid & x \leftarrow \mathsf{recv}\ c; P & (\supset R, \wedge L) \\
 & \mid & \mathsf{send}\ c\ e; P & (\wedge R, \supset L) \\
 & \mid & c \leftarrow \mathsf{recv}\ c; P & (\multimap R, \otimes L) \\
 & \mid & \mathsf{send}\ c\ (c \leftarrow P); P & (\otimes R, \multimap L) \\
 & \mid & \mathsf{case}\ c\ \mathsf{of} & (\& R, \oplus L) \\
 & & \quad \mathsf{inl} \to P & \\
 & & \quad \mathsf{inr} \to P & \\
 & \mid & \mathsf{send}\ c\ \mathsf{inl}; P & (\oplus R_1, \& L_1) \\
 & \mid & \mathsf{send}\ c\ \mathsf{inr}; P & (\oplus R_2, \& L_2) \\
 & \mid & \mathsf{shift} \leftarrow \mathsf{recv}\ c; P & (\uparrow R, \downarrow L) \\
 & \mid & \mathsf{send}\ c\ \mathsf{shift}; P & (\downarrow R, \uparrow L)
\end{array}
$$

Figure 4.2: SILL Syntax

4.1.4 Typing Rules

Now that we have fixed our types and syntax, we need to connect the two by adapting the proof rules of Figure 4.1. This gives us two type judgments, one for $\Psi \vdash \tau$ and one for $\Psi; \Gamma \vdash A$. The first will again be assumed to reuse its rules from an already existing $\Psi \vdash e : \tau$, where Ψ is a mapping from variable names to types from the underlying functional language (i.e., those generated by the underlying non-terminal τ), and will be extended with one case for the construct for $\{\}I$. The judgement for processes is given by $\Psi; \Gamma \vdash P :: c : A$, where Ψ is a mapping from variable names to types from the underlying functional language, Γ is a mapping from channel names to session types, P is the process expression, c is the channel provided by P, and A is the type of c (recall that A without a superscript denotes either A^+ or A^-). It may not be immediately obvious which channel is provided by a particular process expression: the answer can be found by examining the channel used by either $\textsc{Id}$ or $1R$. The full listing of rules is presented in Figure 4.3, where $\vec{c} : \vec{A}$ denotes a mapping of the channels of $\vec{c}$ to their counterparts in $\vec{A}$.

4.1.5 Semantics

The operational semantics of this system is given in terms of a substructural semantics [87], a semantics approach based on multiset rewriting [19]. In this style of semantics execution configurations consist of multisets of executing processes and transitions take the form of linear implications. This allows us to utilize linear logic's existing facilities for tracking resources along with persistent information to describe that a configuration evolves from, e.g., E to

$$\frac{\Psi; \vec{c} : \vec{A} \vdash P :: c : A}{\Psi \vdash c \leftarrow \{P\} \prec \vec{c} : \{A \leftarrow \vec{A}\}}\ \{\}I \qquad \frac{\Psi \vdash e : \{B \leftarrow \vec{A}\} \quad \Psi; \Gamma, b : B \vdash P :: c : C}{\Psi; \Gamma, \vec{a} : \vec{A} \vdash b \leftarrow e \prec \vec{a}; P :: c : C}\ \{\}E$$

$$\frac{}{\Psi; a : A \vdash c \leftarrow a :: c : A}\ \textsc{Id} \qquad \frac{}{\Psi; \emptyset \vdash \mathsf{close}\ c :: c : 1}\ 1R$$

$$\frac{\Psi; \Gamma \vdash P :: c : C}{\Psi; \Gamma, a : 1 \vdash \mathsf{wait}\ a; P :: c : C}\ 1L \qquad \frac{\Psi \vdash e : \tau \quad \Psi; \Gamma \vdash P :: c : C^+}{\Psi; \Gamma \vdash \mathsf{send}\ c\ e; P :: c : \tau \wedge C^+}\ \wedge R$$

$$\frac{\Psi, x : \tau; \Gamma, a : A^+ \vdash P :: c : C}{\Psi; \Gamma, a : \tau \wedge A^+ \vdash x \leftarrow \mathsf{recv}\ a; P :: c : C}\ \wedge L$$

$$\frac{\Psi, x : \tau; \Gamma \vdash P :: c : C^-}{\Psi; \Gamma \vdash x \leftarrow \mathsf{recv}\ c; P :: c : \tau \supset C^-}\ \supset R$$

$$\frac{\Psi \vdash e : \tau \quad \Psi; \Gamma, a : A^- \vdash P :: c : C}{\Psi; \Gamma, a : \tau \supset A^- \vdash \mathsf{send}\ a\ e; P :: c : C}\ \supset L$$

$$\frac{\Psi; \Gamma \vdash P :: a : A^+ \quad \Psi; \Gamma' \vdash Q :: c : C^+}{\Psi; \Gamma, \Gamma' \vdash \mathsf{send}\ c\ (a \leftarrow P); Q :: c : A^+ \otimes C^+}\ \otimes R$$

$$\frac{\Psi; \Gamma, a : A^+, b : B^+ \vdash P :: c : C}{\Psi; \Gamma, a : B^+ \otimes A^+ \vdash b \leftarrow \mathsf{recv}\ a; P :: c : C}\ \otimes L$$

$$\frac{\Psi; \Gamma, a : A^+ \vdash P :: c : C^-}{\Psi; \Gamma \vdash a \leftarrow \mathsf{recv}\ c; P :: c : A^+ \multimap C^-}\ \multimap R$$

$$\frac{\Psi; \Gamma \vdash P :: b : B^+ \quad \Psi; \Gamma, a : A^- \vdash Q :: c : C}{\Psi; \Gamma, a : B^+ \multimap A^- \vdash \mathsf{send}\ a\ (b \leftarrow P); Q :: c : C}\ \multimap L$$

$$\frac{\Psi; \Gamma \vdash P :: c : A^+}{\Psi; \Gamma \vdash \mathsf{send}\ c\ \mathsf{inl} :: c : A^+ \oplus B^+}\ \oplus R_1 \qquad \frac{\Psi; \Gamma \vdash P :: c : B^+}{\Psi; \Gamma \vdash \mathsf{send}\ c\ \mathsf{inr} :: c : A^+ \oplus B^+}\ \oplus R_2$$

$$\frac{\Psi; \Gamma, a : A^+ \vdash P :: c : C \quad \Psi; \Gamma, a : B^+ \vdash Q :: c : C}{\Psi; \Gamma, a : A^+ \oplus B^+ \vdash \begin{pmatrix} \mathsf{case}\ a\ \mathsf{of} \\ \mathsf{inl} \to P \\ \mathsf{inr} \to Q \end{pmatrix} :: c : C}\ \oplus L$$

$$\frac{\Psi; \Gamma \vdash P :: c : A^- \quad \Psi; \Gamma \vdash Q :: c : B^-}{\Psi; \Gamma \vdash \begin{pmatrix} \mathsf{case}\ c\ \mathsf{of} \\ \mathsf{inl} \to P \\ \mathsf{inr} \to Q \end{pmatrix} :: c : A^- \& B^-}\ \& R$$

$$\frac{\Psi; \Gamma, a : A^- \vdash P :: c : C}{\Psi; \Gamma, a : A^- \& B^- \vdash \mathsf{send}\ a\ \mathsf{inl}; P :: c : C}\ \& L_1$$

$$\frac{\Psi; \Gamma, a : B^- \vdash P :: c : C}{\Psi; \Gamma, a : A^- \& B^- \vdash \mathsf{send}\ a\ \mathsf{inr}; P :: c : C}\ \& L_2$$

$$\frac{\Psi; \Gamma \vdash P :: c : A^-}{\Psi; \Gamma \vdash \mathsf{send}\ c\ \mathsf{shift}; P :: c : \downarrow A^-}\ \downarrow R \qquad \frac{\Psi; \Gamma, a : A^- \vdash P :: c : C}{\Psi; \Gamma, a : \downarrow A^- \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ a; P :: c : C}\ \downarrow L$$

$$\frac{\Psi; \Gamma \vdash P :: c : A^+}{\Psi; \Gamma \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c; P :: c : \uparrow A^+}\ \uparrow R \qquad \frac{\Psi; \Gamma, a : A^+ \vdash P :: c : C}{\Psi; \Gamma, a : \uparrow A^+ \vdash \mathsf{send}\ a\ \mathsf{shift}; P :: c : C}\ \uparrow L$$

Figure 4.3: SILL Type System

E' by proving it from the semantic implications. Parallelism in a programming language then becomes the non-determinism inherent in deciding how to utilize these implications; side conditions become persistent (i.e., !) propositions; and fresh variables are represented by existential quantification. We will assume that $\otimes$ binds tighter than $\multimap$ or quantification. To see this in action, let us examine the following artificial rule, where all Greek variables are atomic predicates:

$$\phi \otimes \rho \otimes (!\alpha) \multimap \exists x.\gamma \otimes \delta \otimes \beta$$

This says that if our set of resources has a ϕ and a ρ and can supply at least one α, we can transition to a state that replaces ϕ and ρ with γ, δ and β, where x is visible only to γ, δ and β. Since $!\alpha$ is persistent even after executing the rule we still have $!\alpha$. Additionally, while this rule does not bind some larger context, we can apply this rule in a larger context. For example, if we started from the multiset of assumptions $\{\phi, \phi, \rho, !\alpha\}$ we could prove the larger transition,

$$\phi \otimes \phi \otimes \rho \otimes (!\alpha) \multimap \phi \otimes \exists x.\gamma \otimes \delta \otimes \beta$$

by composing our starting transition with $\otimes R/L$. This is similar to the frame rule of Separation Logic [80].

In asynchronous communication each linear channel contains a message queue [36], which can be related directly to the proof system via continuation channels [28]. Sending adds to the queue on one end and receiving takes from the other. Because session-based communication goes in both directions, the queue switches direction at certain times. Moreover, the queue must maintain some information on the direction of the queue so that a process that performs a send followed by a receive does not incorrectly read its own message. Fortunately, session typing guarantees that there is no send/receive mismatch.

Our configurations, consisting of both linear and persistent resources, as in Linear Logic (section 2.4), utilize two sorts of linear propositions: executing processes and queues that buffer messages between them. Executing processes are of the form $\mathsf{exec}_c(P)$ where c is the channel provided by P. Queues are of the form $\mathsf{que}(a, M, b)$ where a is the channel name that a process can use to read from the queue, b is the channel name that a process can use to write to the queue, and M is a list of messages whose elements are from the following grammar:

$$K ::= v \mid c \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{end} \mid \mathsf{shift} \mid \mathsf{fwd}^{+}(c) \mid \mathsf{fwd}^{-}(c)$$

The constructs of K are: v for values from our underlying functional language; c for channels; inl and inr for choices; end for messages indicating process termination; and $\mathsf{fwd}^{+}(c)/\mathsf{fwd}^{-}(c)$ indicating that the current queue "continues" by reading the queue at c (the polarity annotations are only needed for later proofs).

Before examining the operational rules, we introduce a persistent predicate, $e \to e'$, indicating that the underlying functional language can take a step, from e to e', via an assumed small-step semantics (big-step would also work, but makes our progress theorem more complicated). First we have two rules that involve evaluating expressions from the underlying language as normal for small step semantics.

$$\begin{aligned}
&\textsc{Bind}_{\mathsf{step}} : \mathsf{exec}_c(a \leftarrow e \multimap\!\!< \vec{a}; P) \otimes !(e \to e') \multimap \mathsf{exec}_c(a \leftarrow e' \multimap\!\!< \vec{a}; P)\\
&\textsc{Data}_{\mathsf{step}} : \mathsf{exec}_c(\mathsf{send}\ b\ e; P) \otimes !(e \to e') \multimap \mathsf{exec}_c(\mathsf{send}\ b\ e'; P)
\end{aligned}$$

We will defer the details of evaluating $(a \leftarrow v \multimap\!\!< \vec{a})$ until later, since it is relatively complicated. In the rules for sending and receiving data we assume that values can be substituted freely for variables (of the correct type). These rules work by either appending the value to the end of the appropriate queue or reading a value from the head of the queue.

$$\begin{aligned}
&\textsc{Send}_{\mathsf{data}} : \mathsf{exec}_c(\mathsf{send}\ b\ v;\ P) \otimes \mathsf{que}(a, M, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M\ v, b)\\
&\textsc{Recv}_{\mathsf{data}} : \mathsf{exec}_c(x \leftarrow \mathsf{recv}\ a; P) \otimes \mathsf{que}(a, v\ M, b) \multimap \mathsf{exec}_c(P[v/x]) \otimes \mathsf{que}(a, M, b)
\end{aligned}$$

When we close a channel its providing process disappears.

$$\begin{aligned}
&\textsc{Send}_{\mathsf{end}} : \mathsf{exec}_c(\mathsf{close}\ c) \otimes \mathsf{que}(a, M, c) \multimap \mathsf{que}(a, M\ \mathsf{end}, c)\\
&\textsc{Recv}_{\mathsf{end}} : \mathsf{exec}_c(\mathsf{wait}\ a; P) \otimes \mathsf{que}(a, \mathsf{end}, b) \multimap \mathsf{exec}_c(P)
\end{aligned}$$

When we transmit a channel, we know the direction the new queue provided by the new process must point (processes spawned this way always provide a positively typed channel).

$$\begin{aligned}
\textsc{Send}_{\mathsf{chan}} : &\ \mathsf{exec}_c(\mathsf{send}\ b\ (d \leftarrow P); Q) \otimes \mathsf{que}(a, M, b)\\
&\multimap \exists f, g. \mathsf{exec}_c(Q) \otimes \mathsf{que}(a, M\ g, b) \otimes \mathsf{exec}_f(P[f/d]) \otimes \mathsf{que}(g, \cdot, f)\\
\textsc{Recv}_{\mathsf{chan}} : &\ \mathsf{exec}_c(d \leftarrow \mathsf{recv}\ a; P) \otimes \mathsf{que}(a, f\ M, b)\\
&\multimap \mathsf{exec}_c(P[f/d]) \otimes \mathsf{que}(a, M, b)
\end{aligned}$$

Choice constructs require us to branch on reception.

$$\begin{aligned}
&\textsc{Send}_{\mathsf{inl}} : \mathsf{exec}_c(\mathsf{send}\ b\ \mathsf{inl}; P) \otimes \mathsf{que}(a, M, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M\ \mathsf{inl}, b)\\
&\textsc{Send}_{\mathsf{inr}} : \mathsf{exec}_c(\mathsf{send}\ b\ \mathsf{inr}; P) \otimes \mathsf{que}(a, M, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M\ \mathsf{inr}, b)\\
&\textsc{Recv}_{\mathsf{inl}} : \mathsf{exec}_c\begin{pmatrix}\mathsf{case}\ a\ \mathsf{of}\\ \mathsf{inl} \to P\\ \mathsf{inr} \to Q\end{pmatrix} \otimes \mathsf{que}(a, M\ \mathsf{inl}, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M, b)\\
&\textsc{Recv}_{\mathsf{inr}} : \mathsf{exec}_c\begin{pmatrix}\mathsf{case}\ a\ \mathsf{of}\\ \mathsf{inl} \to P\\ \mathsf{inr} \to Q\end{pmatrix} \otimes \mathsf{que}(a, M\ \mathsf{inr}, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M, b)
\end{aligned}$$

When receiving, but not when sending, a shift, we reverse the direction of the queue.

$$\begin{array}{l}\textsc{Send}_{\mathsf{shift}} : \mathsf{exec}_c(\mathsf{send}\ b\ \mathsf{shift}; P) \otimes \mathsf{que}(a, M, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M\ \mathsf{shift}, b) \\ \textsc{Recv}_{\mathsf{shift}} : \mathsf{exec}_c(\mathsf{shift} \leftarrow \mathsf{recv}\ a; P) \otimes \mathsf{que}(a, \mathsf{shift}, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(b, \cdot, a)\end{array}$$

The remaining rules all need a more careful accounting of the directionality of channels involved in communication. For example, when using $a \leftarrow e \multimap\!\!< \vec{a}$, we will create a new process and a new que corresponding to the channel that the new process provides. Since our queues are directed and the newly spawned process can provide either a positive or negative type, we need to know which direction to point this newly created que. Unfortunately, we cannot easily discover the direction from the process expression syntax. An easy way to fix this would be to split this construct into a pair of polarized constructs, e.g., $a \overset{+}{\leftarrow} e \multimap\!\!< \vec{a}$ and $a \overset{-}{\leftarrow} e \multimap\!\!< \vec{a}$. Instead, we assume that any typechecker will be able to resolve this ambiguity and record the information so that at run time we will execute the appropriate choice of operational rule. In these rules we use the persistent predicates (i.e., ones that we can weaken and contract as needed), $\textsc{Pos}(a)$ and $\textsc{Neg}(a)$, that indicate the channel a initially has either a positive or negative type in the given process expression, respectively.

$$\begin{array}{ll}\textsc{Bind}_{+}: & \mathsf{exec}_c(a \leftarrow (b \leftarrow \{Q\} \multimap\!\!< \vec{b}) \multimap\!\!< \vec{a}; P) \otimes !(\textsc{Pos}(a)) \\ & \quad \multimap \exists d, f. \mathsf{exec}_c(P[d/a]) \otimes \mathsf{exec}_f(Q[f, \vec{a}/b, \vec{b}]) \otimes \mathsf{que}(d, \cdot, f) \\ \textsc{Bind}_{-}: & \mathsf{exec}_c(a \leftarrow (b \leftarrow \{Q\} \multimap\!\!< \vec{b}) \multimap\!\!< \vec{a}; P) \otimes !(\textsc{Neg}(a)) \\ & \quad \multimap \exists d, f. \mathsf{exec}_c(P[d/a]) \otimes \mathsf{exec}_f(Q[f, \vec{a}/b, \vec{b}]) \otimes \mathsf{que}(f, \cdot, d)\end{array}$$

To implement the process $(c \leftarrow d)$ we want to send either $\mathsf{fwd}^{+}(c)$ or $\mathsf{fwd}^{-}(d)$ depending on whether c is positive in this process, so that we respect the directionality of channels. As with $\textsc{Bind}$ we assume this choice will be resolved either by type checking or, not pursued here, by splitting the construct into two polarized versions. Sending forwarding messages is easy:

$$\begin{array}{ll}\textsc{Send}_{\mathsf{fwd}^+}: & \mathsf{que}(a, M, c) \otimes \mathsf{exec}_c(c \leftarrow d) \otimes !(\textsc{Pos}(c)) \multimap \mathsf{que}(a, M\ \mathsf{fwd}^{+}(d), c) \\ \textsc{Send}_{\mathsf{fwd}^-}: & \mathsf{que}(a, M, d) \otimes \mathsf{exec}_c(c \leftarrow d) \otimes !(\textsc{Neg}(c)) \multimap \mathsf{que}(a, M\ \mathsf{fwd}^{-}(c), d)\end{array}$$

However, receiving forwarding messages is slightly harder. Since there is no "receive a forward message" instruction, processes must be ready to receive one any time they perform a receive operation. In the following rules we will constrain P to be one of the following:

$$\mathsf{wait}\ a; Q \qquad x \leftarrow \mathsf{recv}\ a; Q \qquad b \leftarrow \mathsf{recv}\ a; Q \qquad \mathsf{shift} \leftarrow \mathsf{recv}\ a; Q \qquad \begin{array}{r}\mathsf{case}\ a\ \mathsf{of} \\ \mathsf{inl} \rightarrow Q \\ \mathsf{inr} \rightarrow R\end{array}$$

In the following rules for receiving a forward, two versions of $\textsc{Recv}_{\mathsf{fwd}^+}$ should be unneeded but seem required for the proof of Session Fidelity (Theorem 6):

$$\begin{aligned}
\textsc{Recv}_{\mathsf{fwd}^-}: \quad & \mathsf{exec}_d(P) \otimes \mathsf{que}(a, \mathsf{fwd}^-(c), d) \otimes \mathsf{que}(c, M, b) \\
& \quad \multimap \mathsf{exec}_c(P[c/a]) \otimes \mathsf{que}(c, M, b) \\
\textsc{Recv}_{\mathsf{fwd}^+_{\leftarrow}}: \quad & \mathsf{exec}_f(P) \otimes \mathsf{que}(f, M, g) \otimes \mathsf{que}(a, \mathsf{fwd}^+(d), c) \\
& \quad \multimap \mathsf{exec}_f(P[d/a]) \otimes \mathsf{que}(f, M, g) \\
\textsc{Recv}_{\mathsf{fwd}^+_{\rightarrow}}: \quad & \mathsf{exec}_f(P) \otimes \mathsf{que}(g, M, f) \otimes \mathsf{que}(a, \mathsf{fwd}^+(d), c) \\
& \quad \multimap \mathsf{exec}_f(P[d/a]) \otimes \mathsf{que}(g, M, f)
\end{aligned}$$

4.1.6 Syntactic Sugar

Before we see an extended example, we need to introduce some common syntactic sugar. First, while logically motivated, using $\otimes R$ or $\multimap L$ to send a preexisting channel is fairly awkward. Thus, we introduce the following syntax to send a single preexisting channel, with its desugaring after the $\longrightarrow$ and a fresh:

$$\mathsf{send}\ c\ d; P \quad \longrightarrow \quad \mathsf{send}\ c\ (a \leftarrow a \leftarrow d); P$$

When dealing with recursively defined processes, it is common to perform a *tail-bind*, which binds a new process and immediately forwards from it. In the following desugaring a is assumed to be fresh:

$$c \leftarrow e \multimap\!\!\prec \vec{c} \quad \longrightarrow \quad a \leftarrow e \multimap\!\!\prec \vec{c}; c \leftarrow a$$

As we will see in section 5.2 and section 5.5, this construct enables an important optimization akin to tail call optimization.

Our reification of processes into the underlying functional language allows us to lift constructs from the underlying functional language to the process level as syntactic sugar rather than as new typing rules. This is similar to the power provided by monadic expressions in Haskell [93]. To give a flavor of this, we present the following lifted if-then-else construct, where c is the channel that the current process provides and $\vec{c}$ are all the currently in-scope channels (notice the use of tail-binding here):

$$\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \quad \longrightarrow \quad c \leftarrow \begin{pmatrix} \mathsf{if}\ e \\ \mathsf{then}\ c \leftarrow \{P\} \multimap\!\!\prec \vec{c} \\ \mathsf{else}\ c \leftarrow \{Q\} \multimap\!\!\prec \vec{c} \end{pmatrix} \multimap\!\!\prec \vec{c}$$

Another important feature to borrow from the underlying functional language is top-level definitions. We introduce the following syntax to enable processes to appear at the top level:

$$c \leftarrow x\ \vec{y} \multimap\!\!\prec \vec{c} = P \quad \longrightarrow \quad x\ \vec{y} = c \leftarrow \{P\} \multimap\!\!\prec\ \vec{c}$$

Since some processes only provide a channel while utilizing no arguments we introduce the following abbreviation, where $\cdot$ represents an empty list of channel arguments:

$$\{A\} \quad \longrightarrow \quad \{A \leftarrow \cdot\}$$

Similarly, uses of $\{\}I$ and $\{\}E$ both can omit their channel arguments:

$$c \leftarrow e; P \quad \longrightarrow \quad c \leftarrow e \multimapinv \cdot; P \qquad \text{and} \qquad c \leftarrow \{P\} \quad \longrightarrow \quad c \leftarrow \{P\} \multimapinv \cdot$$

4.1.7 Example: Prime Sieve

In this section, we work through a small example: building a prime sieve. To accomplish this we will define a process that provides an infinite stream of natural numbers by counting up from some starting point, a process expression that filters the output of another process, and a function that uses these to recursively filter out newly identified composite numbers after each newly discovered prime.

First we start with a variant of our `Stream` type for natural numbers, represented by the type `Nat`.

$$\mathsf{type}\ \texttt{NStream} = \mu t^-.(\uparrow 1) \& \uparrow(\texttt{Nat} \wedge \downarrow t^-)$$

Next we define our counting process, which either terminates, if directed to do so, or sends the current natural number n and recurses, via a tail-bind, to a process that counts up from $n+1$.

```
countup : Nat → {NStream}
c ← countup n =
  case c of
    inl → shift ← recv c;
          close c
    inr → shift ← recv c;
          send c n;
          send c shift;
          c ← countup (n + 1)
```

Next we define a filter that takes a `NStream` and a predicate, with type `Nat` $\to$ `Bool`, and either kills the `NStream` or requests an element from it until one is found that passes the predicate. Notice that in the case where our predicate fails we cannot tail-bind the recursive call to `filter` since we must cope with the initial $\&\uparrow$. Additionally, we can see that while we will be able to guarantee deadlock freedom (Theorem 8), we cannot guarantee productivity: a `filter` process may never find a satisfactory element.

```
$\texttt{filter} : (\texttt{Nat} \to \texttt{Bool}) \to \{\texttt{NStream} \leftarrow \texttt{NStream}\}$
$c \leftarrow \texttt{filter}\ p \multimapinv d =$
 case $c$ of
   inl $\to$ shift $\leftarrow$ recv $c$;
             send $d$ inl;
             send $d$ shift;
             wait $d$;  close $c$
   inr $\to$ send $d$ inr;
             send $d$ shift;
             $x \leftarrow$ recv $d$;
             if $p\ x$
               then shift $\leftarrow$ recv $c$;
                    send $c\ x$;
                    send $c$ shift;
                    shift $\leftarrow$ recv $d$;
                    $c \leftarrow \texttt{filter}\ p \multimapinv d$
               else shift $\leftarrow$ recv $d$;
                    $a \leftarrow \texttt{filter}\ p \multimapinv d$;
                    send $a$ inr;
                    send $a$ shift;
                    $c \leftarrow a$
```

Last, we define our sieving process by taking in a `NStream` of potential primes (where the first is assumed to be prime) and then recursively filtering the tail of that the stream to remove multiples of that prime.

```
$\texttt{sieve} : \{\texttt{NStream} \leftarrow \texttt{NStream}\}$
$c \leftarrow \texttt{sieve} \multimapinv d =$
 case $c$ of
   inl $\to$ send $d$ inl;
             send $d$ shift;
             wait $d$;
             close $c$
   inr $\to$ send $d$ inr;
             send $d$ shift;
             $x \leftarrow$ recv $d$;
             shift $\leftarrow$ recv $d$;
             shift $\leftarrow$ recv $c$;
             send $c\ x$;
             send $c$ shift
             $a \leftarrow \texttt{filter}\ (\lambda y. x \nmid y) \multimapinv d$
             $c \leftarrow \texttt{sieve} \multimapinv a$
```

Lastly, we can define a small wrapper that initiates the sieving process from the initial prime.

$$\begin{aligned}&\texttt{sieve} : \{\texttt{NStream}\}\\&c \leftarrow \texttt{sieve} =\\&\quad a \leftarrow \texttt{countup}\ 2;\\&\quad c \leftarrow \texttt{sieve} \multimapinv a\end{aligned}$$

4.1.8 Well-typed Polarized Configurations

To prove progress and preservation for SILL, we need to define two notions: a well-typed queues and well-typed execution configurations. A queue provides two different names for the logical channel and a buffer of messages that mediates between its users' views of the channel. This means that queues connect two different types based on their message contents. To express this we create a typing judgment for queues $\Gamma \vdash M : A \leftrightsquigarrow B$ which says that we could present a channel of type B as having type A by prepending the messages of M to whatever would normally be sent over that channel. Queues ending in either forwarding messages or end cannot be prepended to anything, so we will allow B to be either a session type or $\bullet$ (Griffith and Pfenning [75] present an alternative version using wildcard channel names rather than $\bullet$), indicating that this queue cannot be extended (in ML terms, a session type `option`). Unfortunately, the information contained in $\mathsf{fwd}^-(c)$ messages will not quite be enough to define well-typed queues or well-typed configurations. To enable this we annotate $\mathsf{fwd}^-(c)$ with a type to record the type of c, becoming $\mathsf{fwd}^-_{C^-}(c)$. Instead of presenting a fully annotated semantics, we will leave the straightforward changes implicit in the proof of the preservation theorem. The rules for this judgment are presented in Figure 4.4. Additionally, we will occasionally need a queue concatenation lemma, though we will mostly use it to append a single message onto the end of a queue.

Lemma 4. *The following rules are admissible:*

$$\frac{\Gamma \vdash M : A^+ \leftrightsquigarrow B^+ \quad \Gamma' \vdash M' : B^+ \leftrightsquigarrow C}{\Gamma, \Gamma' \vdash M\ M' : A^+ \leftrightsquigarrow C}\ \mathit{trans}^+$$

$$\frac{\Gamma \vdash M : A^- \leftrightsquigarrow B^- \quad \Gamma' \vdash M' : B^- \leftrightsquigarrow C}{\Gamma, \Gamma' \vdash M\ M' : A^- \leftrightsquigarrow C}\ \mathit{trans}^-$$

Proof. By induction on the proof for M and then by cases on the last proof rule used.

Case $\emptyset_{\mathsf{q}}$: Use the proof of $\Gamma' \vdash M'$.

Case $\mathsf{end}_{\mathsf{q}}$: There cannot be a proof for M', so this case is vacuously true.

Other cases: Use the inductive hypothesis.

□

$$\frac{}{\emptyset \vdash \cdot : A \leftsquigarrow A}\ \emptyset_{\mathsf{q}} \qquad \frac{}{\emptyset \vdash \mathsf{end} : 1 \leftsquigarrow \bullet}\ \mathsf{end}_{\mathsf{q}}$$

$$\frac{}{b : B^+ \vdash \mathsf{fwd}^+(b) : B^+ \leftsquigarrow \bullet}\ \mathsf{fwd}^+_{\mathsf{q}} \qquad \frac{}{\vdash \mathsf{fwd}^-_{B^-}(b) : B^- \leftsquigarrow \bullet}\ \mathsf{fwd}^-_{\mathsf{q}}$$

$$\frac{\emptyset \vdash v : \tau \quad \Gamma \vdash M : A^+ \leftsquigarrow C}{\Gamma \vdash v\ M : \tau \wedge A^+ \leftsquigarrow C}\ \wedge_{\mathsf{q}} \qquad \frac{\emptyset \vdash v : \tau \quad \Gamma \vdash M : A^- \leftsquigarrow C}{\Gamma \vdash v\ M : \tau \supset A^- \leftsquigarrow C}\ \supset_{\mathsf{q}}$$

$$\frac{\Gamma \vdash M : A^+ \leftsquigarrow B}{\Gamma, c : C^+ \vdash c\ M : C^+ \otimes A^+ \leftsquigarrow B}\ \otimes_{\mathsf{q}} \qquad \frac{\Gamma \vdash M : A^- \leftsquigarrow B}{\Gamma, c : C^+ \vdash c\ M : C^+ \multimap A^- \leftsquigarrow B}\ \multimap_{\mathsf{q}}$$

$$\frac{\Gamma \vdash M : A^+ \leftsquigarrow C}{\Gamma \vdash \mathsf{inl}\ M : A^+ \oplus B^+ \leftsquigarrow C}\ \oplus^{\mathsf{inl}}_{\mathsf{q}} \qquad \frac{\Gamma \vdash M : B^+ \leftsquigarrow C}{\Gamma \vdash \mathsf{inr}\ M : A^+ \oplus B^+ \leftsquigarrow C}\ \oplus^{\mathsf{inr}}_{\mathsf{q}}$$

$$\frac{\Gamma \vdash M : A^- \leftsquigarrow C}{\Gamma \vdash \mathsf{inl}\ M : A^- \& B^- \leftsquigarrow C}\ \&^{\mathsf{inl}}_{\mathsf{q}} \qquad \frac{\Gamma \vdash M : B^- \leftsquigarrow C}{\Gamma \vdash \mathsf{inr}\ M : A^- \& B^- \leftsquigarrow C}\ \&^{\mathsf{inr}}_{\mathsf{q}}$$

$$\frac{}{\emptyset \vdash \mathsf{shift} : {\downarrow} A^- \leftsquigarrow A^-}\ \downarrow_{\mathsf{q}} \qquad \frac{}{\emptyset \vdash \mathsf{shift} : {\uparrow} A^+ \leftsquigarrow A^+}\ \uparrow_{\mathsf{q}}$$

Figure 4.4: Well-typed Queues

Informally, a well-typed configuration is one where: every process provides exactly one queue; every queue has at most two users; and the provider and client of each queue agree on its type (adjusted by $\leftsquigarrow$). These goals are accomplished by the following rules for a well-typed configuration, where $\Gamma^\bullet$ respresents a typing environment where every entry is of the form $(c : \bullet)$:

$$\frac{}{\Gamma^\bullet \vdash \cdot}\ \mathsf{WF}_\bullet$$

$$\frac{\emptyset; \Gamma' \vdash P :: c : C^- \quad \Gamma'' \vdash M : C^- \leftsquigarrow B \quad \Gamma, \Gamma', \Gamma'' \vdash E}{\Gamma, b : B \vdash \mathsf{exec}_c(P), \mathsf{que}(c, M, b), E}\ \mathsf{WF}_{\leftarrow}$$

$$\frac{\emptyset; \Gamma' \vdash P :: c : C \quad \Gamma'' \vdash M : B^+ \leftsquigarrow C \quad \Gamma, \Gamma', \Gamma'' \vdash E}{\Gamma, b : B^+ \vdash \mathsf{exec}_c(P), \mathsf{que}(b, M, c), E}\ \mathsf{WF}_{\rightarrow}$$

$$\frac{\Gamma' \vdash M\ \mathsf{end} : A \leftsquigarrow \bullet \quad \Gamma, \Gamma', b : \bullet \vdash E}{\Gamma, a : A \vdash \mathsf{que}(a, M\ \mathsf{end}, b), E}\ \mathsf{WF}_{\mathsf{end}}$$

$$\frac{\Gamma' \vdash M\ \mathsf{fwd}^+(c) : A^+ \leftsquigarrow \bullet \quad \Gamma, \Gamma' \vdash E}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M\ \mathsf{fwd}^+(c), b), E}\ \mathsf{WF}_{\mathsf{fwd}^+}$$

$$\frac{\Gamma' \vdash M : C^- \leftsquigarrow D \quad \Gamma, \Gamma', b : \bullet \vdash \mathsf{que}(a, M'\ \mathsf{fwd}^-_{C^-}(c), b), E}{\Gamma, d : D \vdash \mathsf{que}(c, M, d), \mathsf{que}(a, M'\ \mathsf{fwd}^-_{C^-}(c), b), E}\ \mathsf{WF}_{\mathsf{fwd}^-_{\leftarrow}}$$

$$\frac{\Gamma' \vdash M : D^+ \leftsquigarrow C^- \quad \Gamma, \Gamma', b : \bullet \vdash \mathsf{que}(a, M'\ \mathsf{fwd}^-_{C^-}(c), b), E}{\Gamma, d : D^+ \vdash \mathsf{que}(d, M, c), \mathsf{que}(a, M'\ \mathsf{fwd}^-_{C^-}(c), b), E}\ \mathsf{WF}_{\mathsf{fwd}^-_{\rightarrow}}$$

Notice that we can resolve the dualizing ambiguity present in $\leftsquigarrow$ by relying on provider annotations (i.e., the subscripts on exec). Since this judgement gives us a tree structure of our processes and every process must provide a channel, our initial typing context cannot be the trivial $\emptyset$ but, instead, will be the next simplest context, $c : 1$, for some specially designated top level channel c.

4.1.9 Theorems

The main theorems we will show about our system are type preservation, also called *session fidelity* in the literature, and progress, which when combined with preservation implies deadlock-freedom. Before doing so we will need a few simple lemmas.

The first lemma allows us to "rotate" the queues used by some process to be adjacent to it in our well-typing proof tree.

Lemma 5. *If we have a well-typedness proof* $\Gamma, a : A \vdash E$*, then we can prove this by starting with a rule that uses* a*.*

Proof. By induction on given proof and then by cases on the last proof rule used.

Case $\mathbf{WF}_\bullet$: Vacuous.

Otherwise: If the current rule starts with the provider for a, we are done. Otherwise, notice that the current rule cannot use a, and then use the inductive hypothesis and a final transposition to finish this case.

□

We can then prove the expected preservation result, with an assumption of preservation for the underlying functional language.

Theorem 6 (Preservation)**.** *If* $\Gamma \vdash E$ *and* $E \to E'$ *then* $\Gamma \vdash E'$*.*

Proof. Proof by cases on the transitions rule. In general this proceeds by replacing the well-typed subproof that directly uses the process mentioned in the case's transition with a new subproof for the results of the transition.

Case $\textsc{Bind}_{\mathsf{step}}$: Use the assumption of preservation of the underlying language to create the new proof.

Case $\textsc{Bind}_{+}$: While this can occur with both $\mathsf{WF}_{\leftarrow}$ and $\mathsf{WF}_{\rightarrow}$, we only show examine the first case, since the differences are minor. We are given:

$$\mathcal{A} = \dfrac{\dfrac{\overbrace{\emptyset; \vec{b} : \vec{A} \vdash P :: b : A}^{\mathcal{P}}}{\emptyset \vdash b \leftarrow \{P\} \prec \vec{b} : \{A \leftarrow \vec{A}\}}\ \{\}I \qquad \overbrace{\emptyset; \Gamma', b : A \vdash Q :: c : C}^{\mathcal{Q}}}{\emptyset; \Gamma', \vec{a} : \vec{A} \vdash a \leftarrow (b \leftarrow \{P\} \prec \vec{b}) \prec \vec{a}; Q :: c : C}\ \{\}E$$

$$\dfrac{\mathcal{A} \quad \overbrace{\Gamma'' \vdash M : C \leftsquigarrow C'}^{\mathcal{M}} \quad \overbrace{\Gamma, \Gamma', \vec{a} : \vec{A}, \Gamma'' \vdash E}^{\mathcal{E}}}{\Gamma, c' : C' \vdash \mathsf{exec}_c(a \leftarrow (b \leftarrow \{P\} \prec \vec{b}) \prec \vec{a}; Q), \mathsf{que}(c, M, c'), E}\ \mathsf{WF}_{\leftarrow}$$

And replace it with the following, where d and d' are fresh and substitution on proofs is used to denote renaming:

$$\dfrac{\mathcal{Q}[d/a] \quad \mathcal{M} \quad \dfrac{\mathcal{P}[d',\vec{a}/b,\vec{b}] \quad \dfrac{}{\emptyset : A \leftrightsquigarrow A}\,\emptyset_{\mathsf{q}} \quad \mathcal{E}}{\mathsf{exec}_{d'}(P[d',\vec{a}/b,\vec{b}]), \mathsf{que}(d,\cdot,d'), E}\,\mathsf{WF}_{\leftarrow}}{\Gamma, c{:}C' \vdash \mathsf{exec}_c(Q[d/a]), \mathsf{que}(c,M,c'), \mathsf{exec}_{d'}(P[d',\vec{a}/b,\vec{b}]), \mathsf{que}(d,\cdot,d'), E}\,\mathsf{WF}_{\leftarrow}$$

Case **Bind$_{-}$:** As with Bind$_{+}$ but swapping the direction of the newly created que and, consequently, using $\mathsf{WF}_{\rightarrow}$.

Case **Send$_{\mathsf{fwd}^+}$:** We are given:

$$\dfrac{\dfrac{}{\emptyset; d : C^+ \vdash c \leftarrow d :: c : C^+}\,\textsc{Id} \quad \overbrace{\Gamma' \vdash M : B^+ \leftrightsquigarrow C^+}^{\mathcal{M}} \quad \overbrace{d : C^+, \Gamma' \vdash E}^{\mathcal{E}}}{\Gamma, b : B^+ \vdash \mathsf{que}(b, M, c), \mathsf{exec}_c(c \leftarrow d), E}\,\mathsf{WF}_{\rightarrow}$$

Which we replace with the following to finish this case:

$$\dfrac{\dfrac{\mathcal{M} \quad \dfrac{}{d : C^+ \vdash C^+ \leftrightsquigarrow \bullet}\,\mathsf{fwd}^+_{\mathsf{q}}}{\Gamma', d : C^+ \vdash M\ \mathsf{fwd}^+(d) : B^+ \leftrightsquigarrow \bullet}\,\mathsf{trans}^+ \quad \mathcal{E}}{\Gamma, b : B^+ \vdash \mathsf{que}(b, M\ \mathsf{fwd}^+(d), c), E}\,\mathsf{WF}_{\mathsf{fwd}^+}$$

Case **Send$_{\mathsf{fwd}^-}$:** There are two cases to consider, when the forwarding process is used with $\mathsf{WF}_{\leftarrow}$ and when it is used with $\mathsf{WF}_{\rightarrow}$. Starting with $\mathsf{WF}_{\leftarrow}$ case we are given, after use of Lemma 5:

$$\mathcal{Z} = \dfrac{\overbrace{\emptyset; \Gamma'' \vdash P :: b : B^-}^{\mathcal{P}} \quad \overbrace{\Gamma''' \vdash M' : B^- \leftrightsquigarrow C^-}^{\mathcal{M}'} \quad \overbrace{\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash E}^{\mathcal{E}}}{\Gamma, d : C^-, \Gamma' \vdash \mathsf{que}(b, M', d), \mathsf{exec}_b(P), E}\,\mathsf{WF}_{\leftarrow}$$

$$\dfrac{\dfrac{}{\emptyset; d : C^- \vdash c \leftarrow d :: c : C^-}\,\textsc{Id} \quad \overbrace{\Gamma' \vdash M' : C^- \leftrightsquigarrow A}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, a : A \vdash \mathsf{que}(c, M, a), \mathsf{exec}_c(c \leftarrow d), \mathsf{que}(b, M', d), \mathsf{exec}_b(P), E}\,\mathsf{WF}_{\leftarrow}$$

Which we replace with the following:

$$\dfrac{\mathcal{M} \quad \dfrac{\mathcal{P} \quad \dfrac{\mathcal{M}' \quad \dfrac{}{\emptyset \vdash \mathsf{fwd}^-_{C^-}(c) : C^- \leftrightsquigarrow \bullet}\,\mathsf{fwd}_{\mathsf{q}}}{\Gamma''' \vdash M'\ \mathsf{fwd}^-_{C^-}(c) : B^- \leftrightsquigarrow \bullet}\,\mathsf{trans}^- \quad \mathcal{E}}{\Gamma, \Gamma', d : \bullet \vdash \mathsf{que}(b, M'\ \mathsf{fwd}^-_{C^-}(c), d), \mathsf{exec}_b(P), E}\,\mathsf{WF}_{\leftarrow}}{\Gamma, a : A \vdash \mathsf{que}(c, M, a), \mathsf{que}(b, M'\ \mathsf{fwd}^-_{C^-}(c), d), \mathsf{exec}_b(P), E}\,\mathsf{WF}_{\mathsf{fwd}^-_{\leftarrow}}$$

When the forwarding process is used with $\mathsf{WF}_{\rightarrow}$ are given, after use of

Lemma 5:

$$\mathcal{Z} = \dfrac{\overbrace{\emptyset; \Gamma'' \vdash P :: b : B^-}^{\mathcal{P}} \quad \overbrace{\Gamma''' \vdash M' : B^- \leftsquigarrow C^-}^{\mathcal{M}'} \quad \overbrace{\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash E}^{\mathcal{E}}}{\Gamma, d : C^-, \Gamma' \vdash \mathsf{que}(b, M', d), \mathsf{exec}_b(P), E}\ \mathsf{WF}_{\leftarrow}$$

$$\dfrac{\dfrac{}{\emptyset; d : C^- \vdash c \leftarrow d :: c : C^-}\ \textsc{Id} \quad \overbrace{\Gamma' \vdash M' : C^- \leftsquigarrow A^+}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M, c), \mathsf{exec}_c(c \leftarrow d), \mathsf{que}(b, M', d), \mathsf{exec}_b(P), E}\ \mathsf{WF}_{\rightarrow}$$

Which we replace with the following:

$$\dfrac{\mathcal{M} \quad \dfrac{\mathcal{P} \quad \dfrac{\mathcal{M}' \quad \dfrac{}{\emptyset \vdash \mathsf{fwd}^-_{C^-}(c) : C^- \leftsquigarrow \bullet}\ \mathsf{fwd_q}}{\Gamma''' \vdash M'\ \mathsf{fwd}^-_{C^-}(c) : B^- \leftsquigarrow \bullet}\ \mathsf{trans}^- \quad \mathcal{E}}{\Gamma, \Gamma', d : \bullet \vdash \mathsf{que}(b, M'\ \mathsf{fwd}^-_{C^-}(c), d), \mathsf{exec}_b(P), E}\ \mathsf{WF}_{\leftarrow}}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M, c), \mathsf{que}(b, M'\ \mathsf{fwd}^-_{C^-}(c), d), \mathsf{exec}_b(P), E}\ \mathsf{WF}_{\mathsf{fwd}^-_{\rightarrow}}$$

Case $\textsc{Recv}_{\mathsf{fwd}^+_{\rightarrow}}$: We are given, after uses of Lemma 5 and assuming P is of the appropriate form (i.e., a receiving instruction using a):

$$\mathcal{Z} = \dfrac{\dfrac{}{b : A^+ \vdash \mathsf{fwd}(b) : A^+ \leftsquigarrow \bullet}\ \mathsf{fwd_q} \quad \overbrace{\Gamma, \Gamma', \Gamma'', b : A^+ \vdash E}^{\mathcal{E}}}{\Gamma, \Gamma', a : A^+, \Gamma'' \vdash \mathsf{que}(a, \mathsf{fwd}^+(b), d), E}\ \mathsf{WF}_{\mathsf{fwd}^+}$$

$$\dfrac{\overbrace{\emptyset; \Gamma', a : A^+ \vdash P :: c : C}^{\mathcal{P}} \quad \overbrace{\Gamma'' \vdash M : C' \leftsquigarrow C}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, c' : C' \vdash \mathsf{que}(c, M, c'), \mathsf{exec}_c(P), \mathsf{que}(a, \mathsf{fwd}^+(b), d), E}\ \mathsf{WF}_{\leftarrow}$$

Which we replace with the following to finish this case, using substitution to denote renaming:

$$\dfrac{\mathcal{P}[b/a] \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, c' : C' \vdash \mathsf{que}(c, M, c'), \mathsf{exec}_c(P[b/a]), E}\ \mathsf{WF}_{\leftarrow}$$

Case $\textsc{Recv}_{\mathsf{fwd}^+_{\leftarrow}}$: Use the proof for $\textsc{Recv}_{\mathsf{fwd}^+_{\rightarrow}}$, but swap the initial use of $\mathsf{WF}_{\rightarrow}$ for a use of $\mathsf{WF}_{\leftarrow}$.

Case $\textsc{Recv}_{\mathsf{fwd}^-}$: We only show the case for $\mathsf{WF}_{\mathsf{fwd}^-_{\leftarrow}}$, the case for $\mathsf{WF}_{\mathsf{fwd}^-_{\rightarrow}}$ is similar. After uses of Lemma 5, we have:

$$\mathcal{Z} = \dfrac{\overbrace{\emptyset \vdash P :: a : C^-}^{\mathcal{P}} \quad \dfrac{}{\emptyset \vdash \mathsf{fwd}^-_{C^-}(c) : C^- \leftsquigarrow \bullet}\ \mathsf{fwd_q} \quad \overbrace{\Gamma, \Gamma' \vdash E}^{\mathcal{E}}}{\Gamma, \Gamma', b : \bullet \vdash \mathsf{que}(a, \mathsf{fwd}^-_{C^-}(c), b), \mathsf{exec}_a(P), E}\ \mathsf{WF}_{\leftarrow}$$

$$\dfrac{\overbrace{\Gamma' \vdash M : C^- \leftsquigarrow D}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, d : D \vdash \mathsf{que}(a, \mathsf{fwd}^-_{C^-}(c), b), \mathsf{exec}_a(P), \mathsf{que}(c, M, d), E}\ \mathsf{WF}_{\mathsf{fwd}^-_{\leftarrow}}$$

And replace it with the following:

$$\dfrac{\mathcal{P}[c/a] \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, d : D \vdash \mathsf{exec}_c(P[c/a]), \mathsf{que}(c, M, d), E}\ \mathsf{WF}_{\leftarrow}$$

Case $\textbf{S}\textsc{end}_{\mathsf{end}}$: We are given:

$$\dfrac{\dfrac{}{\emptyset;\emptyset \vdash \mathsf{close}\ c :: c : 1}\ 1R \quad \overbrace{\Gamma' \vdash M : A^+ \leftsquigarrow 1}^{\mathcal{M}} \quad \overbrace{\Gamma, \Gamma' \vdash E}^{\mathcal{E}}}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M, c), \mathsf{exec}_c(\mathsf{close}\ c), E}\ \mathsf{WF}_{\rightarrow}$$

Which becomes:

$$\dfrac{\dfrac{\mathcal{M} \quad \dfrac{}{\emptyset \vdash \mathsf{end} : 1 \leftsquigarrow \bullet}\ \mathsf{end_q}}{\Gamma' \vdash M\ \mathsf{end} : A^+ \leftsquigarrow \bullet}\ \mathsf{trans}^+ \quad \mathcal{E}}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M\ \mathsf{end}, c), E}\ \mathsf{WF_{end}}$$

Case $\textbf{R}\textsc{ecv}_{\mathsf{end}}$: There are two case here, when the initial rule is $\mathsf{WF}_{\leftarrow}$ or $\mathsf{WF}_{\rightarrow}$. We only show the first, the other is very similar. We are given, after use of Lemma 5:

$$\mathcal{Z} = \dfrac{\dfrac{}{\emptyset \vdash \mathsf{end} : 1 \leftsquigarrow \bullet}\ \mathsf{end_q} \quad \overbrace{\Gamma, \Gamma', \Gamma'' \vdash E}^{\mathcal{E}}}{\Gamma, \Gamma', a : 1, \Gamma'' \vdash \mathsf{que}(a, \mathsf{end}, b), E}\ \mathsf{WF_{end}}$$

$$\dfrac{\dfrac{\overbrace{\emptyset; \Gamma' \vdash P :: c : C}^{\mathcal{P}}}{\emptyset; \Gamma', a : 1 \vdash \mathsf{wait}\ a; P :: c : C}\ 1L \quad \overbrace{\Gamma'' \vdash M : C \leftsquigarrow D}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, d : D \vdash \mathsf{que}(c, M, d), \mathsf{exec}_c(\mathsf{wait}\ a; P), \mathsf{que}(a, \mathsf{end}, b), E}\ \mathsf{WF}_{\leftarrow}$$

And replace this with:

$$\dfrac{\mathcal{P} \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, c : C \vdash \mathsf{que}(c, M, d), \mathsf{exec}_c(P), E}\ \mathsf{WF_P}$$

Case $\textbf{S}\textsc{tep}_{\mathsf{date}}$: Use preservation for the underlying language.

Case $\textbf{S}\textsc{end}_{\mathsf{data}}$: There are two cases, corresponding to $\wedge R$ and $\supset L$. The second is only slightly more complicated, so we only show the first. We are given:

$$\mathcal{E} = \Gamma; \Gamma'; \Gamma' \vdash E$$

$$\dfrac{\dfrac{\emptyset \vdash v : \tau \quad \emptyset; \Gamma' \vdash P :: c : C^+}{\emptyset; \Gamma' \vdash \mathsf{send}\ c\ v; P :: c : \tau \wedge C^+}\ \wedge R \quad \Gamma'' \vdash M : B^+ \leftsquigarrow \tau \wedge C^+ \quad \mathcal{E}}{\Gamma, b : B^+ \vdash \mathsf{que}(b, M, c), \mathsf{exec}_c(\mathsf{send}\ c\ v; P), E}\ \mathsf{WF}_{\rightarrow}$$

We can replace this with:

$$\dfrac{\mathcal{P} \quad \dfrac{\mathcal{M} \quad \dfrac{\dfrac{}{\emptyset \vdash \cdot : C^+ \leftrightsquigarrow C^+}\;\emptyset_{\mathsf{q}}}{\emptyset \vdash \tau \wedge C^+ \leftrightsquigarrow C^+}\;\wedge_{\mathsf{q}}}{\Gamma'' \vdash M\ v : B^+ \leftrightsquigarrow C^+}\;\mathsf{trans}^+ \quad \mathcal{E}}{\Gamma, b : B^+ \vdash \mathsf{que}(b, M\ v, c), \mathsf{exec}_c(P), E}\;\mathsf{WF}_{\rightarrow}$$

Case $\textbf{R}\textsc{ecv}_{\mathsf{data}}$**:** Again there are two cases, one for $\supset R$ and one for $\wedge L$. Since they are fundamentally the same, we focus on the simplier $\supset R$ case. We are given:

$$\mathcal{Z} = \dfrac{\emptyset \vdash v : \tau \quad \overbrace{\Gamma'' \vdash C^- \leftrightsquigarrow B}^{\mathcal{M}}}{\Gamma'' \vdash M : \tau \supset C^- \leftrightsquigarrow B}\;\supset_{\mathsf{q}}$$

$$\dfrac{\dfrac{\overbrace{x : \tau; \Gamma' \vdash P :: c : C^-}^{\mathcal{P}}}{\emptyset; \Gamma' \vdash x \leftarrow \mathsf{recv}\ c; P :: c : \tau \supset C^-}\;\supset R \quad \mathcal{Z} \quad \overbrace{\Gamma, \Gamma', \Gamma'' \vdash E}^{\mathcal{E}}}{\Gamma, b : B \vdash \mathsf{que}(c, v\ M, b), \mathsf{exec}_c(x \leftarrow \mathsf{recv}\ c; P), E}\;\mathsf{WF}_{\leftarrow}$$

Which we can replace with the following, $\mathcal{P}[v/x]$ denotes using substitutabililty in the underlying language:

$$\dfrac{\mathcal{P}[v/x] \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, b : B \vdash \mathsf{que}(c, M, b), \mathsf{exec}_c(P[v/x]), E}\;\mathsf{WF}_{\leftarrow}$$

Case $\textbf{S}\textsc{end}_{\mathsf{chan}}$**:** As $\textsc{Send}_{\mathsf{data}}$, but with $\otimes R$ and $\multimap L$.

Case $\textbf{R}\textsc{ecv}_{\mathsf{chan}}$**:** As $\textsc{Recv}_{\mathsf{data}}$, but with $\multimap R$ and $\otimes L$.

Case $\textbf{S}\textsc{end}_{\mathsf{inl}}$**:** As $\textsc{Send}_{\mathsf{data}}$, but with $\oplus R_1$ and $\& L_1$.

Case $\textbf{R}\textsc{ecv}_{\mathsf{inl}}$**:** As $\textsc{Recv}_{\mathsf{data}}$, but with $\& R$ and $\oplus L$.

Case $\textbf{S}\textsc{end}_{\mathsf{inr}}$**:** As $\textsc{Send}_{\mathsf{data}}$, but with $\oplus R_2$ and $\& L_2$.

Case $\textbf{R}\textsc{ecv}_{\mathsf{inr}}$**:** As $\textsc{Recv}_{\mathsf{data}}$, but with $\& R$ and $\oplus L$.

Case $\textbf{S}\textsc{end}_{\mathsf{shift}}$**:** As $\textsc{Send}_{\mathsf{data}}$, but with $\downarrow R$ and $\uparrow L$.

Case $\textbf{R}\textsc{ecv}_{\mathsf{shift}}$**:** Two cases, $\uparrow R$ and $\downarrow L$. Start with $\uparrow R$. We are given:

$$\mathcal{E} = \Gamma, \Gamma' \vdash E$$

$$\dfrac{\dfrac{\overbrace{\emptyset; \Gamma' \vdash P :: c : C^+}^{\mathcal{P}}}{\emptyset; \Gamma' \vdash \mathsf{shift} \leftarrow \mathsf{recv}\ c; P :: c : \uparrow C^+}\;\uparrow R \quad \dfrac{}{\emptyset \vdash \mathsf{shift} : \uparrow C^+ \leftrightsquigarrow C^+}\;\uparrow_{\mathsf{q}} \quad \mathcal{E}}{\Gamma, a : C^+ \vdash \mathsf{que}(c, \mathsf{shift}, a), \mathsf{exec}_c(\mathsf{shift} \leftarrow \mathsf{recv}\ c; P), E}\;\mathsf{WF}_{\leftarrow}$$

Which we replace with:

$$\dfrac{\mathcal{P} \quad \dfrac{}{\emptyset \vdash \cdot : C^+ \leftsquigarrow C^+}\,\emptyset_{\mathsf{q}} \quad \mathcal{E}}{\Gamma, a : C^+ \vdash \mathsf{que}(a,\cdot,c), \mathsf{exec}_c(\mathsf{shift} \leftarrow \mathsf{recv}\ c; P), E}\ \mathsf{WF}_{\to}$$

For ${\downarrow}L$, there are two cases, corresponding to whether the inital rule is $\mathsf{WF}_{\leftarrow}$ or $\mathsf{WF}_{\to}$. Since they are very similar we only show the case for $\mathsf{WF}_{\leftarrow}$. After use of Lemma 5, we have:

$$\mathcal{Z} = \dfrac{\overbrace{\emptyset; \Gamma''' \vdash Q :: d : B^-}^{\mathcal{Q}} \quad \dfrac{}{\emptyset \vdash \mathsf{shift} : {\downarrow}B^- \leftsquigarrow B^-}\,{\downarrow}_{\mathsf{q}} \quad \overbrace{\Gamma,\Gamma',\Gamma'',\Gamma''' \vdash E}^{\mathcal{E}}}{\Gamma,\Gamma', b : {\downarrow}B^-, \Gamma'' \vdash \mathsf{que}(b,\mathsf{shift},d), \mathsf{exec}_d(Q), E}\ \mathsf{WF}_{\to}$$

$$\dfrac{\dfrac{\overbrace{\emptyset;\Gamma', b : B^+ \vdash P :: c : C}^{\mathcal{P}}}{\emptyset;\Gamma', b : {\downarrow}B^+ \vdash \mathsf{shift} \leftarrow \mathsf{recv}\, b; P :: c : C}\,{\downarrow}L \quad \overbrace{\Gamma'' \vdash M : A \leftsquigarrow C}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, a{:}A \vdash \mathsf{que}(a,M,c), \mathsf{exec}_c(\mathsf{shift} \leftarrow \mathsf{recv}\, b; P), \mathsf{que}(b,\mathsf{shift},d), \mathsf{exec}_d(Q), E}\ \mathsf{WF}_{\to}$$

Which we replace with:

$$\dfrac{\mathcal{P} \quad \mathcal{M} \quad \dfrac{\mathcal{Q} \quad \dfrac{}{\emptyset \vdash \cdot : B^- \leftsquigarrow B^-}\,\emptyset_{\mathsf{q}} \quad \mathcal{E}}{\Gamma,\Gamma', b : B^-, \Gamma'' \vdash \mathsf{que}(d,\cdot,b), \mathsf{exec}_d(Q), E}\ \mathsf{WF}_{\leftarrow}}{\Gamma, a{:}A \vdash \mathsf{que}(a,M,c), \mathsf{exec}_c(P), \mathsf{que}(d,\cdot,b), \mathsf{exec}_d(Q), E}\ \mathsf{WF}_{\to}$$

□

To prove the progress theorem, we first need a notion of which configurations should legitimately be unable to transition (i.e., which do not count as deadlocked). A configuration should be *stuck* if it is waiting on an interaction with the outside world (i.e., there is some context that it could be inserted into and transition). We call such configurations *reactive.*

Definition 7 (Reactive)**.** *We call a queue,* $\mathsf{que}(c, M, d)$ reactive*, if it is non-empty. A process* $\mathsf{exec}_c(P)$ *is* reactive *if its top level construct is one of* $\supset R$*,* $\multimap R$*,* $\& R$*, or* ${\uparrow}R$ *and its provided queue is empty (i.e.,* $\mathsf{que}(c, \cdot, d)$ *for some* d*). A configuration* E *is* reactive *if each of its queues is reactive or provided by a reactive process.*

Progress then says that either a configuration can take a step or it is reactive.

Theorem 8 (Progress)**.** *If* $\Gamma \vdash E$ *then either:* $E \to E'$ *or* E *is reactive.*

Proof. By induction on $\Gamma \vdash E$. There are four cases of last rule used to consider:

Case $\mathsf{WF}_{\bullet}$: Vacuously true.

Case $\mathsf{WF}_{\mathsf{end}}$: The mentioned queue must be non-empty, so it is reactive. By the induction hypotheses the remainder of the configuration can either transition or is reactive. If the subconfiguration can transition, so can the overall configuration. If the subconfiguration is reactive, adding a reactive queue to it makes the overall configuration reactive.

Case $\mathsf{WF}^{+}_{\mathsf{fwd}}$: As with the previous case, the queue must be non-empty, so either E is reactive or the subconfiguration can transition.

Case $\mathsf{WF}_{\leftarrow}$: By the induction hypothesis, the subconfiguration is either able to make a transition or is reactive. This first case means that the overall configuration can transition. The second case is more involved. If the top-level instruction for P is one of of the right rules, it can either transition, e.g., via one of the Send rules, or is reactive. If the top-level instruction is one of the left rules, then, since the subconfiguration is reactive, it can perform the appropriate transition. To see this, consider the case where the appropriate transition is a Send and the case where it is a Recv. In the Send case, the appropriate transition can be used since the corresponding queue is reactive and thus empty and pointed in the correct direction. In the Recv, the appropriate transition can be used since the corresponding queue has something to receive.

Case $\mathsf{WF}_{\rightarrow}$: Like the WF_{+} case but dualizing the rules involved.

Case $\mathsf{WF}_{\mathsf{fwd}^{-}_{\leftarrow}}$: This cannot directly transition, but, by the inductive hypothesis, if E, excluding the process that provides a (or queue that metions a in the case of multiple forwards), cannot transition it must be reactive. As before the exluded process either can transition by interacting with the remainder of E, or it can receive a message from the forwarding queue.

Case $\mathsf{WF}_{\mathsf{fwd}^{-}_{\rightarrow}}$: As above.

□

Recall that the top level process was typed with $c : 1 \vdash E$. Thanks to Theorem 8 we know that if this reaches a configuration, E', where it is stuck it must be reactive. By preservation and inverting the well-typedness judgment we then know that E' must be $\mathsf{que}(c, \mathsf{end}, b)$, i.e., we know that either we can make progress or all our processes have terminated.

Polarization allows us to place upper bounds on the size of queues that might be needed at run time [36]. To make this precise we define a notion of a bounded type and the (maximum) queue size of that type.

Definition 9 (Bounded)**.** *A proposition A is* bounded *with* queue size *n, for $n \geq 0$ if it satisfies the judgment $n \vdash A$ of Figure 4.5 for some n. Additionally, we define the following bounding function $A|^{n}_{k}$ where $n \geq 1$, $k \geq 0$, and $n \geq k$ (n is the upper bound on queue size and k is a counter tracking how many more*

$$
\frac{n \geq 1}{n \vdash 1}\;1 \qquad \frac{n \geq 0 \quad n \vdash A^{+}}{n \vdash \uparrow A^{+}}\;\uparrow \qquad \frac{n \geq 0 \quad n \vdash A^{+}}{n \vdash \downarrow A^{-}}\;\downarrow
$$

$$
\frac{n-1 \vdash A^{+}}{n \vdash \tau \wedge A^{+}}\;\wedge \qquad \frac{n-1 \vdash A^{-}}{n \vdash \tau \supset A^{-}}\;\supset
$$

$$
\frac{n-1 \vdash A^{+} \quad n-1 \vdash B^{+}}{n \vdash A^{+} \otimes B^{+}}\;\otimes \qquad \frac{n-1 \vdash A^{+} \quad n-1 \vdash B^{-}}{n \vdash A^{+} \multimap B^{-}}\;\multimap
$$

$$
\frac{n-1 \vdash A^{+} \quad n-1 \vdash B^{+}}{n \vdash A^{+} \oplus B^{+}}\;\oplus \qquad \frac{n-1 \vdash A^{-} \quad n-1 \vdash B^{-}}{n \vdash A^{-} \& B^{-}}\;\&
$$

Figure 4.5: Boundedness Judgment

steps before a polarity shift must be inserted):

$$
\begin{array}{ccc}
A^{+}|_{0}^{n} = \downarrow\uparrow(A^{+}|_{n}^{n}) & A^{-}|_{0}^{n} = \uparrow\downarrow(A^{-}|_{n}^{n}) & 1|_{k}^{n} = 1
\end{array}
$$

$$
\begin{array}{rr}
(\uparrow A^{-})|_{k}^{n} = \uparrow(A^{-}|_{n}^{n}) & (\downarrow A^{-})|_{k}^{n} = \downarrow(A^{-}|_{n}^{n}) \\
(\tau \wedge A^{+})|_{k}^{n} = \tau \wedge (A^{+}|_{k-1}^{n}) & (\tau \supset A^{-})|_{k}^{n} = \tau \supset (A^{+}|_{k-1}^{n}) \\
(A^{+} \otimes B^{+})|_{k}^{n} = (A^{+}|_{k-1}^{n}) \otimes (B^{+}|_{k-1}^{n}) & (A^{+} \oplus B^{+})|_{k}^{n} = (A^{+}|_{k-1}^{n}) \oplus (B^{+}|_{k-1}^{n}) \\
(A^{+} \multimap B^{-})|_{k}^{n} = (A^{+}|_{k}^{n}) \otimes (B^{-}|_{k-1}^{n}) & (A^{+} \& B^{+})|_{k}^{n} = (A^{-}|_{k-1}^{n}) \& (B^{-}|_{k-1}^{n})
\end{array}
$$

There is a bit of tension between this definition of boundedness and our operational semantics. Specifically, the operational semantics stores shift messages in its queues, while this boundness judgment treats $\uparrow$ and $\downarrow$ as needing no space for its messages. Before seeing why we chose this treatment, we need a few lemmas.

Lemma 10. *For any $k \geq 0$, $k \vdash A|_{k}^{n}$.*

Proof. By induction on A.

Case $A = 1$: Trivial.

Case $k = 0$: We need either $1 \vdash \downarrow\uparrow A^{+}$ or $1 \vdash \uparrow\downarrow A^{-}$. Both of these follow from the rules for $\uparrow$ and $\downarrow$ when combined with the inductive hypothesis.

Case $k > 0$: These follow from the relevant rule plus the inductive hypothesis.

□

The following results states that our intuitions about upper bounds on queue sizes is respected by well-typed configurations. Note, a slightly tighter result could be proven by defining initial queue sizes as those starting from a type's root.

Lemma 11. *If $\Gamma \vdash E$ and que(a, M, b) is in E and a is assigned type A in a proof of $\Gamma \vdash E$, then the length of M, ignoring any shift or fwd$^{\pm}$, is at most the queue size of A.*

Proof. Find the proof of $A \leftsquigarrow B$ needed to type $\Gamma' \vdash (\mathsf{que}(a, M, b), E')$. The size of that proof is the size of the number of elements of M. Since these queue proofs cannot extend past a polarity shift, the queue size of A provides an upper bound on the size of this queue proof and, thus, an upper bound on the size of M. □

We have ignored recursive types when discussing boundedness. Our boundedness judgement can be adapted to handle this case by adding a mapping the judgment to hold assumed queue size bounds for any μ-bound subterm of the type. Then two rules need to be added to handle the μ case (letting ζ represent the mapping from subterms to bounds):

$$\frac{m \leq n \quad \zeta, (\mu x.A : m); m \vdash A[\mu x.A/x]}{\zeta; n \vdash \mu x.A}\ \mu \qquad \frac{m \leq n}{\zeta, (\mu x.A : m); n \vdash \mu x.A}\ \mu'$$

This rule works by letting the proof assert that $\mu x.A$ has a queue size bound of m and then confirming that assertion, while possibly assuming that bound in the process. To force boundedness without thought towards minimizing the number of inserted polarity shifts, we can insert a polarity shift at every μ.

To convert an asynchronous program to a synchronous one, use $A|_1^1$ on all of the types of the program and insert shift instructions (i.e., uses of $\uparrow R$, $\uparrow L$, $\downarrow R$, and $\downarrow L$) as needed. While the underlying semantics is asynchronous, a type with queue size 1 can only send one "interesting" message at a time. Essentially, the extraneous $\uparrow$s and $\downarrow$s force us to implement a handshaking protocol for synchronous communication over an asynchronous network.

4.1.10 Related Work

The most directly related work is Toninho et al.'s work [90]. They explore a language based on a Curry-Howard style connection with, unpolarized, Intuitionistic Linear Logic by providing a synchronous semantics. Since they do not need to track queues they give combined Send/Recv rules like the following:

$$\begin{aligned} \textsc{Comm}_{\mathsf{data}} : \mathsf{exec}_c(\mathsf{send}\ c\ v; P) \otimes \mathsf{exec}_d(x \leftarrow \mathsf{recv}\ c; Q) \\ \multimap \mathsf{exec}_c(P) \otimes \mathsf{exec}_d(Q[v/x]) \end{aligned}$$

In addition to a enabling a somewhat shorter presentation, this enables a stronger notion of synchronization. Specifically, their semantics allows for no operations to occur between steps of communication (there are no distinct steps). To see this more concretely, contrast the following two processes that utilize two channels with the type $\texttt{Int} \wedge 1$. On the left is one in suitable for use with their

$$\begin{aligned}
1|^{+} &= 1 & 1|^{-} &= \downarrow 1 \\
(\tau \wedge A)|^{+} &= \tau \wedge (A|^{+}) & (\tau \wedge A)|^{-} &= \uparrow(\tau \wedge (A|^{+})) \\
(\tau \supset A)|^{+} &= \downarrow(\tau \supset (A|^{-})) & (\tau \supset A)|^{-} &= \tau \supset (A|^{-}) \\
(A \otimes B)|^{+} &= (A|^{+} \otimes (B|^{+})) & (A \otimes B)|^{-} &= \uparrow(A|^{+} \otimes (B|^{+})) \\
(A \multimap B)|^{+} &= \downarrow(A|^{+} \multimap (B|^{-})) & (A \multimap B)|^{-} &= (A|^{+} \multimap (B|^{+})) \\
(A \oplus B)|^{+} &= (A|^{+} \oplus (B|^{+})) & (A \oplus B)|^{-} &= \uparrow(A|^{+} \oplus (B|^{+})) \\
(A \& B)|^{+} &= \downarrow(A|^{-} \& (B|^{-})) & (A \& B)|^{-} &= (A|^{-} \& (B|^{-}))
\end{aligned}$$

Figure 4.6: Polarization Function

synchronous system; on the right is a one using synchronized types in SILL:

```
foo : {1 ← Int ∧ 1; Int ∧ 1}    bar : {1 ← Int ∧ ↓↑1; Int ∧ ↓↑1}
c ← foo ⤙ a b =                 c ← bar ⤙ a b =
  x ← recv a;                     x ← recv a;
  y ← recv b;                     y ← recv b;
  wait a;                         shift ← recv a;
  wait b;                         shift ← recv b;
  close c                         send a shift;
                                  send b shift;
                                  wait a;
                                  wait b;
                                  close c
```

In addition to being more verbose, the synchronized types in SILL allow the provider of b to send its `Int` before the provider of a knows its `Int` has been received. This can be avoided by requiring shifts to occur immediately after their associated recv.

To go from an unpolarized logic proposition A to polarized logic we can use the pair of polarization functions, $A|^{+}$ and $A|^{-}$, shown in Figure 4.6. The two superscripts indicate whether the resulting polarized proposition should be positive or negative. Since the initial context could be viewed as either negative or positive, we generally want to use the result of $A|^{+}$ or $A|^{-}$ that has fewer shifts. After converting types in this way, the programmer (or possibly the typechecker) will still need to insert the appropriate shift instructions.

Another higher-order integration of Session Types is that of Monstrous and Yoshida [62]. Our logical foundation makes the language presentation simpler and we feel that a monadic integration of process expressions into our functional language is cleaner than their approach of passing closures between processes.

$$\begin{aligned}
\textsc{Recv}_{\mathsf{d}} &: \mathsf{exec}_c(x \leftarrow \mathsf{recv}\ d; P) \otimes \mathsf{que}(d, v\ M\ T, c') \\
&\quad \multimap \mathsf{exec}_c(P[v/x]) \otimes \mathsf{que}(d, M\ T, c') \\
\textsc{Recv}_{\mathsf{c}} &: \mathsf{exec}_c(a \leftarrow \mathsf{recv}\ d; P) \otimes \mathsf{que}(d, b\ M\ T, c') \\
&\quad \multimap \mathsf{exec}_c(P[b/a]) \otimes \mathsf{que}(d, M\ T, c') \\
\textsc{Recv}_{\mathsf{inl}} &: \mathsf{exec}_c\begin{pmatrix}\mathsf{case}\ d\ \mathsf{of} \\ \mathsf{inl} \rightarrow P_1 \\ \mathsf{inr} \rightarrow P_2\end{pmatrix} \otimes \mathsf{que}(d, \mathsf{inl}\ M\ T, c') \\
&\quad \multimap \mathsf{exec}_c(P_1) \otimes \mathsf{que}(d, M\ T, c') \\
\textsc{Recv}_{\mathsf{inr}} &: \mathsf{exec}_c\begin{pmatrix}\mathsf{case}\ d\ \mathsf{of} \\ \mathsf{inl} \rightarrow P_1 \\ \mathsf{inr} \rightarrow P_2\end{pmatrix} \otimes \mathsf{que}(d, \mathsf{inr}\ M\ T, c') \\
&\quad \multimap \mathsf{exec}_c(P_2) \otimes \mathsf{que}(d, M\ T, c')
\end{aligned}$$

Figure 4.7: Altered Operational Rules for Bundled Messages

4.2 Focusing

In high performance systems, reducing the overall number of communications performed by sending fewer larger messages can be an important optimization [63], assuming that communciation is relatively expensive. With polarization this can be accomplished by revising our operational rules to prohibit reading until a phase of communication is complete. Consequently, the queues can be viewed merely as large *bundled messages*, incrementally constructed, to be sent all at once. While this semantics that enable this optimization look very similar to that presented in subsection 4.1.5, it will never read partial messages (i.e., the queues are never used concurrently other than when one process blocks).

We implement this optimization by tracking the phase terminating messages, shift and end, as well as $\mathsf{fwd}^{\pm}(c)$, in our semantics by creating a new non-terminal T. Otherwise our configurations are constructed as before (subsection 4.1.5), with M now representing lists of Ks possible with a single T at the end (i.e., $M = K^*T|K^*$ as a regular expression):

$$T ::= \mathsf{shift} \mid \mathsf{end} \mid \mathsf{fwd}^+(c) \mid \mathsf{fwd}^-(c) \qquad K ::= v \mid \mathsf{inl} \mid \mathsf{inr} \mid c$$

Using our new classification, we restrict the various receiving rules to only perform their receive when they can see both the component message they care about and the end of the bundled message. These restricted rules are shown in Figure 4.7, otherwise rules are reused from subsection 4.1.5. Whether a que has a phase terminating message encodes whether it models a message that is under construction in some local buffer (if it lacks a phase terminator) or it models a message that has been sent and is ready for consumption. In addition to the generic communication overhead savings, if we are willing to track transmitted and untransmitted messages separately, a bundled system allows to entirely omit the shift and end message components. While we will not pursue this further, this may be useful while constructing efficient real-world implementations.

One drawback of this approach is that for unbounded (i.e., not bounded per Definition 9) types we may never actually transmit the bundled message because we never, dynamically, encounter a T-message. For most applications this maybe acceptable, e.g., we may have an external termination argument indicating that a phase terminates without being able to represent it in SILL's type system. Additionally, as we saw in Lemma 10, it is always possible to force boundedness for any session type by inserting extra shifts, but this might cause us to miss some opportunities for bundling messages that would be possible otherwise. In practice, setting a bound that forces each bundled message to be less than some system appropriate size (e.g., to ensure all bundled messages fit in a single jumbo frame) should make this concern irrelevant.

4.2.1 Theorems

We can prove the expected preservation result reasonably easily.

Theorem 12 (Preservation)**.** *If $\Gamma \vdash E$ and $E \to E'$, then $\Gamma \vdash E'$.*

Proof. By cases on the transition. The proof proceeds as in Theorem 6. □

Before proving progress, we need to modify our definition of reactive slightly. The proof of Theorem 8 relies on being able to ensure that a Recv transition can be used if the queue used by that rule is reactive. With the restrictions on these rules in the bundled messaging semantics, we need a somewhat more restrictive definition of reactive.

Definition 13 (Completely Reactive)**.** *A queue, $\mathsf{que}(c, M, d)$, is* completely reactive*, if $M = M'\ T$ (i.e., it has a terminating message). A process $\mathsf{exec}_c(P)$ is* completely reactive *if its top level construct is one of $\supset R$, $\multimap R$, $\& R$, or $\uparrow R$ and its provided queue is empty (i.e., $\mathsf{que}(c, \cdot, d)$ for some d). A configuration E is* completely reactive *if each of its queues is completely reactive or provided by a completely reactive process.*

Notice that for the initial process reactive and completely reactive coincide, so progress still ensures deadlock freedom.

Theorem 14 (Progress)**.** *If $\Gamma \vdash E$ then either $E \to E'$ or E is completely reactive.*

Proof. Use the same argument of Theorem 8, but substitute completely reactive for reactive. □

4.2.2 Focused Logics

While bundled messages can be an important operation, their use may lead to mysterious executions. Specifically, it breaks the normally tight linkage between when messages are sent and when they can first be received. Of course, one answer, common in HPC settings, is to tell programmers "tough, learn a new quirk," however, by utilizing focused logics we can force the programmer to write a program that both "obviously" enables the bundling optimization while retaining our logical motivation.

Polarization is, traditionally, a precursor to requiring proofs to be in a *focused* normal form. Focusing is a normal form [51], originally studied in the context of proof search, that structures proofs as an alternating sequence of synchronous or asynchronous rules where all possible asynchronous rules are applied in each asynchronous phase and during each synchronous phase synchronous rules are applied to a single formula (and its resulting subformulae) until we can no longer apply synchronous rules to this formula. While rooted in the literature, the uses of "synchronous rules" and "asynchronous rules" in this section are unfortunately overloaded with the notion of synchronous and asynchronous communication. Thanks to the tightly controlled nature of the synchronous phase, this normal form can be viewed as using small rules to generate a class of big-step rules for use in a, more complicated, proof system that has no notion of phasing.

From a Curry-Howard perspective the synchronous rules correspond to those that send information and the asynchronous rules correspond to rules that receive information. Thanks to the "single (sub)formula(e)" portion of the normal form this means that all of the sends along a channel in a given communication phase occur consecutively not just from the channel's perspective but are actually adjacent in the program text as well.

Let us now turn to how to describe this system formally. First, we need to augment our judgments to allow for focusing annotations of the form $[A]$ to appear either in Γ or as the goal proposition. Letting $\overline{\Gamma}$ indicate that Γ contains no focusing annotations, we can enforce the focusing restrictions with the weakly focused system [49] of Figure 4.8. In this weakly focused system, rules can be classified into two different kinds: active rules, which contain no focusing annotation, and focused rules, which contain exactly one focusing annotation. Thanks to the design of the rules, this means that as many focused rules are used on the same formula as possible. The Curry-Howard connection here is as before, with the assumption that uses of Focus^{+} and Focus^{-} are implicit. As a result, all uses of Send rules along a channel in a single phase of communication occur consecutively, exactly as if the programmer were explicitly building and then finalizing a bundled message.

In addition to the constraints imposed by weakly focused logic, we could instead use a focused logic [9, 51]. A non-weakly focused logic enforces an extra constraint on the unfocused propositions present in the focused rules: assumptions must all be negative and the consequent must be positive. Thus, a proof must first apply all possible active rules before choosing to initiate a focused phase. Historically, this difference has been most important when performing proof search, it ensures that all invertible rules are used before initiating a

$$\frac{}{\Psi; A^+ \vdash [A^+]}\ \textsc{Id}^+ \qquad \frac{}{\Psi; [A^-] \vdash A^-}\ \textsc{Id}^- \qquad \frac{}{\Psi; \emptyset \vdash [1]}\ 1R \qquad \frac{\Psi; \overline{\Gamma} \vdash A}{\Psi; \overline{\Gamma}, 1 \vdash A}\ 1L$$

$$\frac{\Psi; \overline{\Gamma_1} \vdash B \quad \Psi; \overline{\Gamma_2}, B \vdash A}{\Psi; \overline{\Gamma_1}, \overline{\Gamma_2} \vdash A}\ \textsc{Cut} \qquad \frac{\Psi; \overline{\Gamma} \vdash [A^+]}{\Psi; \overline{\Gamma} \vdash A^+}\ \textsc{Focus}^+ \qquad \frac{\Psi; \overline{\Gamma}, [A^-] \vdash B}{\Psi; \overline{\Gamma}, A^- \vdash B}\ \textsc{Focus}^-$$

$$\frac{\Psi \vdash \tau \quad \Psi; \overline{\Gamma} \vdash [A^+]}{\Psi; \overline{\Gamma} \vdash [\tau \wedge A^+]}\ \wedge R \qquad \frac{\Psi, \tau; \overline{\Gamma}, A^+ \vdash B}{\Psi; \overline{\Gamma}, \tau \wedge A^+ \vdash B}\ \wedge L \qquad \frac{\Psi, \tau; \overline{\Gamma} \vdash A^-}{\Psi; \overline{\Gamma} \vdash \tau \supset A^-}\ \supset R$$

$$\frac{\Psi \vdash \tau \quad \Psi; \overline{\Gamma}, [A^-] \vdash B}{\Psi; \overline{\Gamma}, [\tau \supset A^-] \vdash B}\ \supset L \qquad \frac{\Psi; \overline{\Gamma_1}, \overline{\Gamma_2} \vdash [A^+] \quad \Psi; \overline{\Gamma_2} \vdash [B^+]}{\Psi; \overline{\Gamma_1}, \overline{\Gamma_2} \vdash [A^+ \otimes B^+]}\ \otimes R$$

$$\frac{\Psi; \overline{\Gamma}, A^+, B^+ \vdash C}{\Psi; \overline{\Gamma}, A^+ \otimes B^+ \vdash C}\ \otimes L \qquad \frac{\Psi; \overline{\Gamma}, A^+ \vdash B^-}{\Psi; \overline{\Gamma} \vdash A^+ \multimap B^-}\ \multimap R$$

$$\frac{\Psi; \overline{\Gamma_1} \vdash [A^+]}{\Psi; \overline{\Gamma_1}, \overline{\Gamma_2}, [A^+ \multimap B^-] \vdash C}\ \multimap L \qquad \frac{\Psi; \overline{\Gamma} \vdash A_i^+}{\Psi; \overline{\Gamma} \vdash A_1^+ \oplus A_2^+}\ \oplus R_i$$

$$\frac{\Psi; \overline{\Gamma}, A^+ \vdash C \quad \Psi; \overline{\Gamma}, B^+ \vdash C}{\Psi; \overline{\Gamma}, A^+ \oplus B^+ \vdash C}\ \oplus L \qquad \frac{\Psi; \overline{\Gamma} \vdash A^- \quad \Psi; \overline{\Gamma} \vdash B^-}{\Psi; \overline{\Gamma} \vdash A^- \& B^-}\ \& R$$

$$\frac{\Psi; \overline{\Gamma}[A_i^-] \vdash B}{\Psi; \overline{\Gamma}[A_1^- \& A_2^-] \vdash B}\ \& L_i \qquad \frac{\Psi; \overline{\Gamma} \vdash A^-}{\Psi; \overline{\Gamma} \vdash [{\downarrow} A^-]}\ {\downarrow} R \qquad \frac{\Psi; \overline{\Gamma}, A^- \vdash B}{\Psi; \overline{\Gamma}, {\downarrow} A^- \vdash B}\ {\downarrow} L$$

$$\frac{\Psi; \overline{\Gamma} \vdash A^+}{\Psi; \overline{\Gamma} \vdash {\uparrow} A^+}\ {\uparrow} R \qquad \frac{\Psi; \overline{\Gamma}, A^+ \vdash B}{\Psi; \overline{\Gamma}, [{\uparrow} A^+] \vdash B}\ {\uparrow} L$$

Figure 4.8: Weakly Focused Logic

phase that may require backtracking. Operationally, this corresponds to reading bundled messages relatively promptly after receiving them, which may allow for lower memory usage by freeing up space used to hold the bundled message and by more promptly exposing data from those messages to the garbage collector.

Focusing can force a program to take extra memory, as witnessed by the following unfocused and focused programs, where the unfocused definition is on the left):

$$\texttt{foo} : \{\texttt{Int} \wedge \texttt{Float} \wedge 1 \leftarrow \texttt{Int} \wedge \texttt{Float} \wedge 1\}$$

$$\begin{array}{ll}
c \leftarrow \texttt{foo} \multimap d = & c \leftarrow \texttt{foo} \multimap d = \\
\quad i \leftarrow \mathsf{recv}\ d; & \quad i \leftarrow \mathsf{recv}\ d; \\
\quad \mathsf{send}\ c\ i; & \quad f \leftarrow \mathsf{recv}\ d; \\
\quad f \leftarrow \mathsf{recv}\ d; & \quad \mathsf{wait}\ d; \\
\quad \mathsf{send}\ c\ f; & \quad \mathsf{send}\ c\ i; \\
\quad \mathsf{wait}\ d; & \quad \mathsf{send}\ c\ f; \\
\quad \mathsf{close}\ c & \quad \mathsf{close}\ c
\end{array}$$

When contrasted with its unfocused counterpart, the focused version of `foo` requires us to store both i and f simultaneously. It is unclear how problematic this is in practice. Under the original asynchronous semantics the que that supplied d might have needed to store both i and f simultaneously as well (i.e., the focused version always achieves the worst case memory usage in this example).

4.2.3 Related Work

Focusing, but not its Curry-Howard interpretation, has been studied by wide varieties of authors [50, 51, 58]. Some of the closest logics to the system we utilize are those that examine focusing for intuistionistic linear logic [58] and that of Baelde [9] that examines focusing in linear logic with fixed points. Bealde's work uses an inductive rule instead of CUT, allowing proofs to replace problematic fixed points with other types. Roughly translated into our system the inductive rule might look like

$$\frac{C \vdash A[C/x] \quad \Psi; \Gamma, C \vdash B}{\Psi; \Gamma, \mu x.A \vdash B}\ \mu L$$

where C can mention x (which would also require some changes to our assumption of closed types). This rule is challenging to operationalize. Specifically, it requires us to replace the running process that provides $\mu x.A$ with one that provides C.

Additionally, instead of a pair of general identity rules, Baelde reduces to an atomic propositional identity rule and an "atomic fixedpoint" identity rule that operates on *frozen* recursive propositions. A frozen recursive proposition is one that is forbidden from being unfolded. Thus Bealde's proof system allows proofs to declare that they are done using recursive proposition in nontrivial ways. This is important from a focusing perspective because the system uses frozen propositions to avoid requiring too much decomposition during focusing. Operationally, there are reasons to want to utilize only atomic notions of forwarding (e.g., it corresponds to utilizing a long running repeater process to perform the forwarding instead of `que` redirection), however, there should be no situations where "atomic fixedpoint" forwarding is executable but a general forwarding rule is not.

The practical importance of the gains to performance offered by bundling messages can been seen by noticing that some high performance computing systems sometimes adopt message bundling in their communication layers to enable efficient usage of, relatively, slow networks. One system that attempts to do this for its users is GRAPPA [63], which stores outgoing messages from a given node to a target node (i.e., messages along a channel in our terms) and only sends bundled messages when either a maximum bundled message sized is reached or some component message has been delayed by a maximum delay time (i.e., it is does not do type directed bundling).

Session Java [64] provides a Java implementation of session types, where channel are rendered as objects in their system and communication primitives transform this object (and its type) over the course of communication. The message sending primitives are structured so that method chaining allows for relatively compact specification of multiple message sends utilizing the same session. A more compact interface is also provided by making the send method variadic so that all sends along a session in one phase of communication can be done via a single method call. It is unclear if this is more than a programmer convenience, but it superficially resembles the textual adjacency that focusing requires. At a higher level the Session Java work does not try to connect to a logic via a Curry-Howard style connection, so, even allowing for operational similarity, our system makes an improvement in that respect.

4.3 Racy Programs

Focusing eliminates one sort of nondeterminism from our proofs by forcing us to apply all the asynchronous rules possible to a given formula at once. This still leaves some degrees of nondeterminism present: the choice of order in which to apply the asynchronous rules and the choice of which formula to focus to initiate a focused phase. It is unclear what removing the latter nondeterminism would do, but more careful treatment of asynchronous phase nondeterminism enables SILL to tackle a class of problems that our prior systems cannot handle.

Consider trying to write a process that sells a single ticket to one of two client processes. We might give each of the clients the type

$$\texttt{Client} = {\downarrow}((\texttt{Int} \supset {\uparrow}1) \& {\uparrow}1)$$

where the ticket vending machine will, after receiving a request (represented here by a shift message) from the client, either tell the client no ticket is available (i.e., send inr) after which that client will terminate, or the client will receive a inl followed by its reservation number. A simple attempt at a vending process then might be

```
$\texttt{vend} : \{1 \leftarrow \texttt{Client}; \texttt{Client}\}$
$c \leftarrow \texttt{vend} \multimap a\ b =$
    shift $\leftarrow\ a$;
    shift $\leftarrow\ b$;
    send $a$ inl;
    send $a$ $42$;
    send $a$ shift;
    wait $a$;
    send $a$ inr;
    send $b$ shift;
    wait $b$;
    close $c$
```

This vending machine has a major flaw: the d client always receives the reservation, i.e., the vending machine is unfair. To cope with this we might introduce some sort of explicit randomization to our process, i.e., vend could flip a coin before deciding which client gets the reservation. Unfortunately, this does not help us implement a "first come first served" notion of fairness. The issue seems to be that while our system is asynchronous, it does not permit any truly racy behavior.

To enable this, we will develop a notion of non-deterministically selecting over a list of channels that could potentially be ready for reception. Specifically, we will split our proofs into phases as in focused logics, i.e., create synthetic rules. This time, the rules will explicitly witness that the nondeterminism inherent in choosing which asynchronous rule to apply at a given step is unimportant. The changes we make to our system are the following:

- Introduce two new rules SEL and SEL$_R$ that demonstrate we could start our proof via any one of a nonempty set of active rules
- Introduce a notion of selected judgments that encode which of our asynchronous rules must be used first
- For each active rule we move it to either a left selected judgment or right judgment as appropriate

Together, these mean that when we use one of the SEL rules, we demonstrate that our proof could safely proceed starting from any of the listed formulae (the selected judgments ensure that the appropriate rule is applied first). As a result, proofs in the original system can reasonably be viewed as a normal form of the

Select Rules:

$$\frac{|\overrightarrow{A^+}| \geq 1 \quad \Psi; \Gamma, \overrightarrow{A^+} \overset{A_0^+}{\vdash} B \quad \dots \quad \Psi; \Gamma, \overrightarrow{A^+} \overset{A_n^+}{\vdash} B}{\Psi; \Gamma, \overrightarrow{A^+} \vdash B} \textsc{Sel}$$

$$\frac{\Psi; \Gamma, \overrightarrow{A^+} \overset{B^-}{\vdash} B^- \quad \Psi; \Gamma, \overrightarrow{A^+} \overset{A_0^+}{\vdash} B \quad \dots \quad \Psi; \Gamma, \overrightarrow{A^+} \overset{A_n^+}{\vdash} B}{\Psi; \Gamma, \overrightarrow{A^+} \vdash B^-} \textsc{Sel}_R$$

Left Selected Rules: $\Delta \overset{A^+}{\vdash} B$

$$\frac{\Psi; \Gamma, A^- \vdash B}{\Psi; \Gamma, {\downarrow}A^- \overset{\downarrow A^-}{\vdash} B} {\downarrow}L \qquad \frac{\Psi; \Gamma \vdash B}{\Psi; \Gamma, 1 \overset{1}{\vdash} B} 1L \qquad \frac{\Psi; \Gamma, A^+, B^+ \vdash C}{\Psi; \Gamma, A^+ \otimes B^+ \overset{\tau \otimes A^+}{\vdash} C} \otimes L$$

$$\frac{\Psi; \Gamma, A^+ \vdash C \quad \Psi; \Gamma, B^+ \vdash C}{\Psi; \Gamma, A^+ \oplus B^+ \overset{A^+ \oplus B^+}{\vdash} C} \oplus L$$

Right Selected Rules: $\Delta \overset{A^-}{\vdash} A^-$

$$\frac{\Psi; \Gamma \vdash A^+}{\Psi; \Gamma \overset{\uparrow A^+}{\vdash} {\uparrow}A^+} {\uparrow}R \qquad \frac{\Psi; \Gamma, A^+ \vdash B^-}{\Psi; \Gamma \overset{A^+ \multimap B^-}{\vdash} A^+ \multimap B^-} \multimap R \qquad \frac{\Psi; \Gamma \vdash A^- \quad \Psi; \Gamma \vdash B^-}{\Psi; \Gamma \overset{A^- \& B^-}{\vdash} A^- \& B-} \& R$$

Figure 4.9: New Racy Rules

system presented in Figure 4.9 that chooses one of the possible ways to proceed at each selection point.

Since our new proof system is a very slight alteration of the original one, our soundness and completeness theorems are relatively easy.

Theorem 15 (Soundness)**.** *If* $\Psi; \Gamma \vdash A$ *in the system of Figure 4.9 then* $\Psi; \Gamma \vdash A$ *in the system of subsection 4.1.1.*

Proof. By erasing annotations and replacing uses of Sel and Sel$_R$ with one of their subproofs we can get a proof in the original system. □

Theorem 16 (Completeness)**.** *If* $\Psi; \Gamma \vdash A$ *in the system of subsection 4.1.1 then* $\Psi; \Gamma \vdash A$ *in the system of Figure 4.9.*

Proof. A proof in the original system is almost a proof in the racy system. It can be transformed into such by replacing usages of ${\downarrow}L$, $1L$, $\otimes L$, and $\oplus L$ with unary uses of Sel with the appropriate rule. Similarly, we can replace uses of ${\uparrow}R$, $\multimap R$, and $\& R$ with uses of Sel$_R$ followed by the appropriate rule. □

```
vend : {1 ← Client; Client}
c ← vend ⤙ d e =
    select d e
    | d → shift ← recv d;
          send d success;
          send d 42;
          send d shift;
          shift ← recv e;
          send e failure;
          send e shift;
          wait d; wait e;
          close c
    | e → shift ← recv e;
          send e success;
          send e 42;
          send e shift;
          shift ← recv d;
          send d failure;
          send d shift;
          wait e; wait d;
          close c
```

Figure 4.10: First Come First Served vend Definition

Operational Rules

Before we can define the operational rules for programs involving SEL and SEL$_R$ we augment the program syntax with the following construct for them (overloading based on usage of the provided channel as normal):

$$P ::= \ldots \mid \mathsf{select}\ \vec{c}\ \{c_i \to P_i\}$$

With this new construct we can express the ticket vending machine that implements a "First Come First Served" notion for resolving conflicts between clients. The code is listed in Figure 4.10 and uses select to choose between which of d and e receive the ticket based on which one's request (i.e., shift) is received first.

As an additional convenience, we will permit the order of the channels supplied to syntax as a list of channels and in the cases to be permutations of each other, thus permitting something other than the provided channel to be the first case when using the typing rule for SEL$_R$. Additionally, we will adopt the convention that unary uses of select may be implicit. The operational rule for select is straightforward: find a waiting message on one of the selected channels and transition to the appropriate process.

$$\textsc{Select} : \mathsf{exec}_c \begin{pmatrix} \mathsf{select}\ \vec{d} \\ d_i \to P_i \end{pmatrix} \otimes \mathsf{que}(d_i, K\ M, a) \otimes !(\forall b. K \notin \{\mathsf{fwd}^+(b), \mathsf{fwd}^-(b)\})$$
$$\multimap \mathsf{exec}_c(P_i) \otimes \mathsf{que}(d_i, K\ M, a)$$

Additionally, we add select to the rules for receiving a forward. It would be safe to omit this update, but it would run the risk of committing to a channel that only has a forwarding message and is not truly ready.

4.3.1 Theorems

Well-typed configurations work almost as in the basic polarized system. Unfortunately, our tagged left and right selection rules will require us to split our well-typed configuration rules. We leave the case where P should be judged using the basic judgment unannotated and add either L or R superscripts to indicate that P should be judged using the left or right selected judgments, respectively. As written, these rules are not syntactically distinct w.r.t. P, but this can be resolved by differentiating between rules that use recv instruction vs those that do not. Notice that there is no case where a hypothetical $\mathsf{WF}^R_\rightarrow$ could be legally applied. We present the new rules below:

$$\frac{\Gamma', a : A^+ \overset{A^+}{\vdash} P :: c : C^- \quad \Gamma'' \vdash M :: C^- \leftsquigarrow B \quad \Gamma, a : A^+, \Gamma', \Gamma'' \vdash E}{\Gamma, d : B \vdash \mathsf{exec}_c(P), \mathsf{que}(c, M, d), E}\ \mathsf{WF}^L_\leftarrow$$

$$\frac{\Gamma' \overset{A^-}{\vdash} P :: c : A^- \quad \Gamma'' \vdash M :: A^- \leftsquigarrow B \quad \Gamma, \Gamma', \Gamma'' \vdash E}{\Gamma, d : B \vdash \mathsf{exec}_c(P), \mathsf{que}(c, M, d), E}\ \mathsf{WF}^R_\leftarrow$$

$$\frac{\Gamma', a : A^+ \vdash P :: c : C^+ \quad \Gamma'' \vdash M :: B \leftsquigarrow C^+ \quad \Gamma, a : A^+, \Gamma', \Gamma'' \vdash E}{\Gamma, d : B \vdash \mathsf{exec}_c(P), \mathsf{que}(d, M, c), E}\ \mathsf{WF}^L_\rightarrow$$

Theorem 17 (Preservation). *If* $\Gamma \vdash E$ *and* $E \to E'$ *then* $\Gamma \vdash E'$.

Proof. Proof by cases on the transition used. Most cases can be handled by reusing the proof of Theorem 6 with appropriate substitution of the selected rules for their basic equivalents. We need one extra case for the select rule. In this case, we fix the subproof for the transitioning process by taking using one of the selected rules. To see this in action, let's examine the case for Select, when the channel with a waiting message is the channel provided by the current process. We are given:

$$\frac{\dfrac{\overbrace{\emptyset; \Gamma' \vdash P :: c : C^-}^{\mathcal{P}} \quad \dots}{\emptyset; \Gamma' \vdash \begin{pmatrix} \text{select } \vec{c} \text{ of} \\ c \to P \\ \dots \end{pmatrix} :: c : C^-}\ \textsc{Sel}_R \qquad \overbrace{\Gamma'' \vdash K\ M :: C^- \leftsquigarrow A}^{\mathcal{M}} \quad \overbrace{\Gamma, \Gamma', \Gamma'' \vdash E}^{\mathcal{E}}}{\Gamma, a : A \vdash \mathsf{exec}_c \begin{pmatrix} \text{select } \vec{c} \text{ of} \\ c \to P \\ \dots \end{pmatrix}, \mathsf{que}(c, K\ M, a), E}\ \mathsf{WF}_\leftarrow$$

Which we replace with:

$$\frac{\mathcal{P} \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, a : A \vdash \mathsf{exec}_c(P), \mathsf{que}(c, K\ M, a), E}\ \mathsf{WF}^R_{\leftarrow}$$

□

Our progress theorem can almost be stated as before, however we first need to alter our definition of reactive. In addition to the prior ways that a process can be considered reactive we add that processes that starts with the construct for $\textsc{Sel}_R$ counts as reactive if all the queues used by that instruction are empty. Notice, that we treat $\textsc{Sel}_R$ as a receiving rule despite the fact that its operational rule does not remove anything from a queue. If, however, we look at the system defined by the "big-step" rules fusing SELECT with the normal receiving rules, this confusion would disappear.

Theorem 18 (Progress)**.** *If $\Gamma \vdash E$ then either $E \to E'$ to E is reactive.*

Proof. By induction, reusing the proof of Theorem 8 with the slight adjustment to handle the splitting of WF_- and WF_+. The cases for SEL and $\textsc{Sel}_R$ are treated just like other receiving rules. □

4.3.2 Related Work

A common operation (going by various names: `select`/`poll`/`epoll`) provided by Unix/POSIX systems for working with file descriptors is the ability to ask the operating system which of a collection of file descriptors can perform some operation (e.g., which file descriptor, if any, has data ready to be read). This is very similar to the facilities provided by the select extension (and the source of its name), however, select provides a simpler but less powerful interface. The unix tools allow for choosing among both read and write operations instead of just read operations. This extra functionality is meaningless in SILL: polarization means that well-typed programs that wish to write can always immediately do so. Describing which file descriptors to wait on is also handled differently, `epoll` creates an empty set of file descriptors and then adds to them via repeated calls to `epoll_ctl`. Since we cannot manipulate channels directly in the functional language (e.g., store them in a list) all channels that our process could possibly select on must be statically known, and, thus, the variadic syntax is as flexible as might ever be needed. Lastly, `epoll` provides a limited amount of message filtering (e.g., to check for high priority messages preferentially). Potentially, this might be an interesting addition to our system, however, message filtering should be explored as an independent language feature first, as some actor systems permit [20]. In addition to these notable differences, `epoll` provides a few concerns that do not directly translate into our context (e.g., interactions with the operating system's powersaving features).

Within the session typing world, the most common way to enable racy behavior is by using the non-deterministic accept/request primitives. Returning to our vending machine example, the vending machine would accept a ticket selling session twice over its lifetime and actually sell the ticket in the first of these sessions while returning an error message in the second. Unfortunately, giving a logical account of these instructions is an open problem and may require weaker progress theorems. The racy extension of SILL does, however permit us to write explicitly the non-deterministic matching that goes on during accept/request, but the tree structure imposed by our well-typed configuration judgement prohibits the most "interesting" (i.e., unsafe) cyclic uses of this.

Some presentations of focusing work on a non-commutative version Γ in their asynchronous rules [23]. Essentially, this fixes the focused normal form more tightly than in section 4.2. Since these extra constraints might hurt completeness, an extra companion lemma needs to be proven to ensure that the mandated order is safe (e.g., by proving that all permutations of Γ preserve provability). This is similar to what select forces us to provide witnesses for, but it is stronger in that select does not require us to show that all possible starting points would be safe to proceed from and imposes no constraints on subproofs (i.e., they may be unfocused).

4.4 Asynchronous Reading

So far we have seen how to incorporate asynchronous sending into SILL, but a related notion, asynchronous receiving, is missing. This section seeks to rectify that. Before turning to the logical basis for asynchronous receiving, let us consider what we might want from such an extension operationally. Operationally, asynchronous receiving should move the point where a recving process must stall from where the receive textually occurs to just before the atomic message that would be received is actually needed. Briefly, we can examine where each of our message types are actually used:

- A data value, v, is used anytime that an expression from our underlying functional language is used, namely in $c \leftarrow e \multimap \vec{c}$ and send $c\ e$. We could be a bit more precise if we were willing to inspect e (i.e., we could determine that e does not need v). For many possible choices of underlying language this is reasonable, particluarly, since we already assume that e can use v by substitution. If we view substitution as merely a concrete instance of the higher level goal of "give the value v to the underlying language", this may be undesirable (e.g., e might be opaque, separately compiled code).
- A channel is used when it is mentioned later in the process. This could be loosened in the case of $\{\}E$ if we allowed for the transference of which process is waiting for the channel to be received. That is, if for some $c \leftarrow e \multimap d$, where d is a channel that will eventually be received by the

process containing the $\{\}E$, we would could transfer the obligation to wait for d to the newly spawned process specified by e. This seems relatively complicated to cooridnate, so we will not explore this option further.

- For inl or inr, these are used immediately as part of a branch. A more permissive approach might run all branches until they try do something irrevocable and treat that as the use point (only performing the correct irrevocable action). This idea has been used in some security tools [8] to ensure privacy policies are respected and in software transactional memory [29].
- For end, the use point could reasonably be any of: never, since we gain no information other than synchronization from it; immediately because we gain no information other than synchronization from it and, so, that must actually be important; or whenever the process closes or forwards, if we want to emphasize that all children of a process must terminate before the process does.
- For shift, the answer depends on whether we wish to reutilize our message queues (as the previous semantics have assumed). If we plan reuse our queues, then it must be viewed as being used before the next write on this channel (so the queue has been emptied before being written to). If we allocate a new queue for each round of communication, then shift is never used.
- The message $\mathsf{fwd}^{\pm}(c)$ is never used and has no construct that directly receives it, thus we do not need to worry about it.

While there is no need to change SILL's type system to accommodate asynchronous receiving, the semantics will, unsurprisingly, change a bit. A direct context reduction semantics for asynchronous receiving should be fairly straightforward to define by allowing for Recv transitions contained deep within executing processes, but this seems to make preservation significantly harder to prove. Instead we will define the semantics by extending our existing transition system with one new rule.

Before defining the new transition, we define a helper operation $\textsc{Reorder}(P)$ that takes a process and produces a set of programs that correspond to receive-use dependency respecting reorderings of the process. The inductive definition of this set is given in Figure 4.11. The members of this set are preserve the typing of the original set. Notice that $\textsc{Reorder}(P)$ is always non-empty.

Lemma 19. *If $\Psi; \Gamma \vdash P :: c : A$ and $P' \in \textsc{Reorder}(P)$, then $\Psi; \Gamma \vdash P' :: c : A$.*

Proof. By induction on the proof of $P' \in \textsc{Reorder}(P)$. □

To perform a receive asynchronously we can think of asking the question "How late can we delay this receive instruction?" and our discussion of when

$$\frac{}{P \in \textsc{Reorder}(P)} \qquad \frac{}{Q_2; Q_1; P \in \textsc{Reorder}(Q_1; Q_2; P)}$$

$$\frac{}{\begin{pmatrix}\mathsf{case}\ f\ \mathsf{of} \\ \ \mathsf{inl} \to Q_1; P_1 \\ \ \mathsf{inr} \to Q_1; P_2\end{pmatrix} \in \textsc{Reorder} \begin{pmatrix} Q_1; \mathsf{case}\ f\ \mathsf{of} \\ \qquad \mathsf{inl} \to P_1 \\ \qquad \mathsf{inr} \to P_2\end{pmatrix}}$$

$$\frac{P'' \in \textsc{Reorder}(P') \quad P' \in \textsc{Reorder}(P)}{P'' \in \textsc{Reorder}(P)}$$

$$\text{where} \quad Q_1 \in \left\{\begin{matrix} x \leftarrow \mathsf{recv}\ c, d \leftarrow \mathsf{recv}\ c, \\ \mathsf{shift} \leftarrow \mathsf{recv}\ c, , \mathsf{wait}\ c, \mathsf{send}\ c\ \mathsf{inl}, \\ \mathsf{send}\ c\ \mathsf{inr}, \mathsf{send}\ c\ \mathsf{shift}, a \leftarrow e \multimap \vec{a} \end{matrix}\right\}$$

$$\text{and} \quad Q_2 \in \left\{\begin{matrix} y \leftarrow \mathsf{recv}\ f, a \leftarrow \mathsf{recv}\ f, \\ \mathsf{shift} \leftarrow \mathsf{recv}\ f, \mathsf{wait}\ f, \mathsf{send}\ f\ \mathsf{inl}, \\ \mathsf{send}\ f\ \mathsf{inr}, \mathsf{send}\ f\ \mathsf{shift}, b \leftarrow e' \multimap \vec{b} \end{matrix}\right\}$$

Figure 4.11: Definition of Reorder

received values can be used provides guidance to answering this question. A related question that Reorder will let us answer is "The current receive cannot be performed, what other later operations would it be safe to do while we wait?" This is accomplished by replacing the currently executing process P with some $P' \in \textsc{Reorder}(P)$ with the hope that P' will be able to make progress.

$$\textsc{Reorder}_{\mathsf{any}} : \quad \mathsf{exec}_c(P) \otimes !(P' \in \textsc{Reorder}(P)) \multimap \mathsf{exec}_c(P')$$

While this rule is appealingly simple, it can introduce non-termination via repeated cyclical uses of $\textsc{Reorder}_{\mathsf{any}}$ even in otherwise terminating programs. In practice, we should execute the system that fuses this general rule with each of the applicable basic operational rules that additionally ensures that P' can immediately transition. As a simple example of this, consider the following fused rule for wait:

$$\begin{aligned}\textsc{Reorder}_{\mathsf{wait}}\!: \quad & \mathsf{exec}_c(P) \otimes !((\mathsf{wait}\ a; P') \in \textsc{Reorder}(P)) \otimes \mathsf{que}(a, \mathsf{end}, b) \\ & \multimap \mathsf{exec}_c(P')\end{aligned}$$

Notice that these rules enable the execution of deeply located, but still Reorderable, instructions even when they are not the first such instruction. Concretely, this can be seen by examining the execution context in Figure 4.12. The process cannot directly execute its first instruction, but, by using Reorder, can execute either the second or third instruction. A more obvious asynchronous semantics might require the second instruction to execute preferentially, but this seems to require a more complicated semantics. For instance, we could require that all channels used between the topmost instruction of a process and the instruction that would be Reordered to the top be unable to react if they were Reordered themselves.

$$\mathsf{exec}_c \begin{pmatrix} x \leftarrow \mathsf{recv}\ a; \\ y \leftarrow \mathsf{recv}\ b; \\ z \leftarrow \mathsf{recv}\ c; \\ \vdots \end{pmatrix}, \mathsf{que}(a, \cdot, a'), \mathsf{que}(b, 5, b'), \mathsf{que}(c, 8, b'), E$$

Figure 4.12: Out-of-order vs Asynchronous Receiving

Logically, performing a Reorder corresponds to trying to perform a commutative Cut during cut elimination to convert a case into a principal cut.

Theorem 20 (Preservation)**.** *If* $\Gamma \vdash E$ *and* $E \rightarrow E'$ *then* $\Gamma \vdash E'$*.*

Proof. Case on the transition used. For the cases for previously existing rules, reuse the corresponding proof case from Theorem 6. In cases where one of the new fused Reorder rules is used, reuse the corresponding case proof composed with Lemma 19. □

Reactive is no longer a tight characterization of stuck processes, i.e., some processes (e.g., in Figure 4.12) that are reactive are not stuck due to the possibility of the Reorder rules being used. Unlike in subsection 4.2.1, asynchronous receiving will only allow us to make progress more often. Thus, we do not need a tighter definition of reactive for the progress result.

Theorem 21 (Progress)**.** *If* $\Gamma \vdash E$ *then either:* $E \rightarrow E'$ *or* E *is reactive.*

Proof. Reuse the proof for Theorem 8. Our new rules can only enable us to transition more readily than before. □

4.4.1 Related Work

The fusion calculus [71] provides a similar ability to perform receives asynchronously, but accomplishes this in a radically different manner: the fusion calculus uses unification to combine names across processes. This allows a receiving process that does not need to know the value of message to proceed with the hopes that unification will have filled in a value by the time that it is required. A realistic implementation should attempt to avoid arbitrary, potentially global, unifications. This lends itself to implementation via Futures [10], which allow for placeholders to be used to coordinate the final transmission of values from senders to users. For a distributed system, this seems a bit unsatisfactory: while there are distributed futures systems [96], these assume a mechanism for communicating between processes in addition to channels.

The most common example of out of order execution is in processors. In this setting, assembly instructions (or microcode microps) are reordered in an attempt to maximize throughput when executing programs. While this is a sufficiently different domain that it is unclear how many lessons for SILL this has, there are a few simple ones. First, it emphasizes that Reorder may be too expensive to use naively in its full generality: we may need either an algorithm to enable faster reordering or we may wish to truncate our searches for usefully reorderable instructions before trying all possibilities. We have already seen one such algorithm expressed in the reduction of asynchronous receiving to synchronous receiving with select, at a worrying code size cost. The other lesson is that nondeterminism is important and may be unavoidable when performing reordering [57], as we have already seen hinted at in section 4.3.

4.5 Polymorphism

For the rest of this thesis we will assume that we start extending SILL from the version presented in section 4.1. We exclude the two alternate semantics since neither are extensions to the base language. The exclusion of select is merely for convenience, it would noticeably complicate the following work, but the idea of allowing selection over invertible rules remains compatible with the rest of SILL.

To add polymorphism to SILL we will explore the addition of atomic propositions and quantification to SILL. Instead of fixing a set of atomic propostions, we will make them explicit in the logic. Specifically, we will augment our propositions with a set of atomic proposition names:

$$\begin{aligned}
\alpha^+, \beta^+, \gamma^+ &::= \ldots && \text{(positive atomic propositions)}\\
\alpha^-, \beta^-, \gamma^- &::= \ldots && \text{(negative atomic propositions)}\\
A^+ &::= \ldots \mid \alpha^+ \\
A^- &::= \ldots \mid \alpha^-
\end{aligned}$$

As with A^+ and A^-, we will omit superscripts when they are unimportant.

To ensure that these atomic proposition names are actually drawn from the set of atomic propositions, we add two well-formed type judgements, $\Delta \vdash \tau$ and $\Delta \vdash A$, that confirm all atomic propositions used in these propositions are in Δ. To do this we resuse the rules from an assumed well-formed type judgement for τ and augment it with the following $\{\}$ rule, in addition to creating the rules for $\Delta \vdash A$ (we will continue to require that propositions are contractive):

$$\frac{\Delta \vdash A \quad \text{for all } i\text{: } \Delta \vdash B_i}{\Delta \vdash \{A \leftarrow \vec{B}\}}\ \{\} \qquad \frac{}{\Delta, \alpha \vdash \alpha}\ \textsc{Elem} \qquad \frac{}{\Delta \vdash 1}\ 1 \qquad \frac{\Delta \vdash A^+}{\Delta \vdash \uparrow A^+}\ \uparrow$$

$$\frac{\Delta \vdash A^-}{\Delta \vdash \downarrow A^-}\ \downarrow \qquad \frac{\Delta \vdash \tau \quad \Delta \vdash A^+}{\Delta \vdash \tau \wedge A^+}\ \wedge \qquad \frac{\Delta \vdash \tau \quad \Delta \vdash A^-}{\Delta \vdash \tau \supset A^-}\ \supset \qquad \frac{\Delta \vdash A^+ \quad \Delta \vdash B^+}{\Delta \vdash A^+ \otimes B^+}\ \otimes$$

$$\frac{\Delta \vdash A^+ \quad \Delta \vdash B^-}{\Delta \vdash A^+ \multimap B^-}\ \multimap \qquad \frac{\Delta \vdash A^+ \quad \Delta \vdash B^+}{\Delta \vdash A^+ \oplus B^+}\ \oplus \qquad \frac{\Delta \vdash A^- \quad \Delta \vdash B^-}{\Delta \vdash A^- \& B^-}\ \&$$

The full logic for this system, presented in Figure 4.13, augments each rule with a Δ to track which atomic propositions can occur in its propopsitions

$$\dfrac{\Delta; \Psi; \vec{A} \vdash A}{\Delta; \Psi \vdash \{A \leftarrow \vec{A}\}}\ \{\}I \qquad \dfrac{\Delta; \Psi \vdash \{A \leftarrow \vec{A}\} \quad \Delta; \Psi; \Gamma, A \vdash B}{\Delta; \Psi; \Gamma, \vec{A} \vdash B}\ \{\}E \qquad \dfrac{}{\Delta; \Psi; A \vdash A}\ \textsc{Id}$$

$$\dfrac{}{\Delta; \Psi; \emptyset \vdash 1}\ 1R \qquad \dfrac{\Delta; \Psi; \Gamma \vdash A}{\Delta; \Psi; \Gamma, 1 \vdash A}\ 1L \qquad \dfrac{\Delta; \Psi; \Gamma \vdash A^+ \quad \Delta; \Psi; \Gamma' \vdash B^+}{\Delta; \Psi; \Gamma, \Gamma' \vdash A^+ \otimes B^+}\ \otimes R$$

$$\dfrac{\Delta; \Psi; \Gamma, A^+, B^+ \vdash C}{\Delta; \Psi; \Gamma, A^+ \otimes B^+ \vdash C}\ \otimes L \qquad \dfrac{\Delta; \Psi; \Gamma, A^+ \vdash B^-}{\Delta; \Psi; \Gamma \vdash A^+ \multimap B^-}\ \multimap R$$

$$\dfrac{\Delta; \Psi; \Gamma \vdash A^+ \quad \Delta; \Psi; \Gamma', B^- \vdash C}{\Delta; \Psi; \Gamma, \Gamma', A^+ \multimap B^- \vdash C}\ \multimap L \qquad \dfrac{\Delta; \Psi; \Gamma \vdash A^- \quad \Delta; \Psi; \Gamma \vdash B^-}{\Delta; \Psi; \Gamma \vdash A^- \& B^-}\ \& R$$

$$\dfrac{\Delta; \Psi; \Gamma, A^- \vdash C}{\Delta; \Psi; \Gamma, A^- \& B^- \vdash C}\ \& L_1 \qquad \dfrac{\Delta; \Psi; \Gamma, B^- \vdash C}{\Delta; \Psi; \Gamma, A^- \& B^- \vdash C}\ \& L_2$$

$$\dfrac{\Delta; \Psi; \Gamma \vdash A^+}{\Delta; \Psi; \Gamma \vdash A^+ \oplus B^+}\ \oplus R_1 \qquad \dfrac{\Delta; \Psi; \Gamma \vdash B^+}{\Delta; \Psi; \Gamma \vdash A^+ \oplus B^+}\ \oplus R_2$$

$$\dfrac{\Delta; \Psi; \Gamma, A^+ \vdash C \quad \Gamma, B^+ \vdash C}{\Delta; \Psi; \Gamma, A^+ \oplus B^+ \vdash C}\ \oplus L \qquad \dfrac{\Delta; \Psi \vdash \tau \quad \Delta; \Psi, \Gamma \vdash A^+}{\Delta; \Psi, \Gamma \vdash \tau \wedge A^+}\ \wedge R$$

$$\dfrac{\Delta; \Psi, \tau; \Gamma, A^+ \vdash B}{\Delta; \Psi; \Gamma, \tau \wedge A^+ \vdash B}\ \wedge L \qquad \dfrac{\Delta; \Psi, \tau; \Gamma \vdash A^-}{\Delta; \Psi; \Gamma \vdash \tau \supset A^-}\ \supset R$$

$$\dfrac{\Delta; \Psi \vdash \tau \quad \Delta; \Psi; \Gamma, A^- \vdash B}{\Delta; \Psi; \Gamma, \tau \supset A^- \vdash B}\ \supset L \qquad \dfrac{\Delta; \Psi; \Gamma \vdash A^-}{\Delta; \Psi; \Gamma \vdash {\downarrow} A^-}\ {\downarrow} R \qquad \dfrac{\Delta; \Psi; \Gamma, A^- \vdash B}{\Delta; \Psi; \Gamma, {\downarrow} A^- \vdash B}\ {\downarrow} L$$

$$\dfrac{\Delta; \Psi; \Gamma \vdash A^+}{\Delta; \Psi; \Gamma \vdash {\uparrow} A^+}\ {\uparrow} R \qquad \dfrac{\Delta; \Psi; \Gamma, A^+ \vdash B}{\Delta; \Psi; \Gamma, {\uparrow} A^+ \vdash B}\ {\uparrow} L$$

Figure 4.13: Polarized Logic with Atomic Propositions

and presuppose that all types are well-formed w.r.t. their Δ. Since atomic propositions cannot be deconstructed, the meaning of *atomic*, we do not need any new logical rules to handle them.

Unfortunately, this will not quite give us a useful programming language. Instead we will want to allow for explicit quantification of session types on top level definitions and explicit instatiation of these variables. To handle this we will add a new kind of proposition δ, which will only be used for expressing relatively simple polymorphism for τ.

$$\delta ::= \forall \vec{\alpha}.\tau$$

We can then extend our existing logic by letting Ψ be a set of δ instead of τ, letting τ abbreviate $\forall.\tau$, and adding the following three rules, one for well-formed types and two logical rules:

$$\dfrac{\Delta, \vec{\alpha} \vdash \tau}{\Delta \vdash \forall \vec{\alpha}.\tau}\ \textsc{delt} \qquad \dfrac{\vec{\beta} \notin (\Delta, \vec{\alpha}) \quad \Delta, \vec{\beta}; \Psi \vdash \tau[\vec{\beta}/\vec{\alpha}]}{\Delta; \Psi \vdash \forall \vec{\alpha}.\tau}\ \textsc{Top}$$

$$\dfrac{\text{for all } i\text{: } \Delta \vdash A}{\Delta; \Psi, \forall \vec{\alpha}.\tau \vdash \tau[\vec{A}/\vec{\alpha}]}\ \textsc{TyApp}$$

```
empty : {queue α⁺}
c ← empty =
    case c of
    | inl →  x ← recv c;
             e ← empty;
             d ← elem ⤙ x e
             c ← d
    | inr →  shift ← recv c;
             send c inr;
             close c
```

```
elem : {queue α⁺ ← α⁺; queue α⁺}
c ← elem ⤙ x d =
    case c of
    | inr →  y ← recv c;
             d.enq;
             send d y;
             e ← elem ⤙ x d;
             c ← e
    | inr →  shift ← recv c;
             send cinr;
             send c x;
             send c shift;
             c ← d
```

Figure 4.14: Queue Example

We then add syntax for these constructs, letting G be the non-terminal for global, i.e., top-level, definitions:

$$
\begin{array}{lcll}
G & ::= & \ldots \mid x : \delta = e & \text{(Top-level session polymorphism)} \\
e & ::= & \ldots \mid x\langle \vec{A} \rangle & \text{(Session polymorphism instantiation)}
\end{array}
$$

which come with the following type rules:

$$
\frac{\vec{\beta} \notin (\Delta, \vec{\alpha}) \quad \Delta, \vec{\beta}; \Psi, x : (\forall \vec{\alpha}.\tau) \vdash e[\vec{\beta}/\vec{\alpha}] : \tau[\vec{\beta}/\vec{\alpha}]}{\Delta; \Psi \vdash x : (\forall \vec{\alpha}.\tau) = e} \ \textsc{Top}
$$

$$
\frac{\text{for all } i\text{: } \Delta \vdash A_i}{\Delta; \Psi, x : \forall \vec{\alpha}.\tau \vdash x\langle \vec{A} \rangle : \tau[\vec{A}/\vec{\alpha}]} \ \textsc{TyApp}
$$

4.5.1 Example: Queue of Channels

As an example of using polymorphism, let us define the type of a queue of channels. Inspired by object oriented languages, a queue should provide the choice between two operations: enqueuing and dequeuing an channel. Since a queue that is requested to dequeue might be empty we will assume that the queue will tell us whether it can successfully return an element before fulfilling that request. Putting this together, we get the following type for a queue of channels of type α^+:

$$
\texttt{queue}\ \alpha^+ = (\alpha^+ \multimap \alpha^+) \& {\uparrow}(1 \oplus (\alpha^+ \otimes {\downarrow}\alpha^+))
$$

We implement this via a mutually recursive definition of two process expressions (Figure 4.14). The two processes, `empty` and `elem`, represent the empty queue and a non-empty queue consisting of an element consed onto the head of a queue, respectively. The empty queue takes no arguments and, since it has nothing else to do, immediately branches on the requested queue operation. In the deque case, inr, there is nothing to do so the process sends inl and terminates itself. In the enqueue case, i.e., where inl is chosen, the process receives the new channel to store and then transforms into an instance of `elem`. The `elem` process takes two channel arguments, one for the channel it stores and one for the tail of the queue. In the `deq` case the process sends its stored channel and then behaves as the tail of the queue. In the enqueue case, it enqueues the new channel into the tail of the queue and recurses to itself.

4.5.2 Quantifiers

With the machinery used for atomic propositions, we can additionally add quantification to our logic. Quantifiers extend the existing with both existential and universal quantification (we will see why these polarities are correct soon):

$$\begin{aligned} A^+ &::= \ldots \mid \exists \alpha^+ . A^+ \\ A^- &::= \ldots \mid \forall \alpha^+ . A^- \end{aligned}$$

Since we have new type constructors, we also need to extend our notion of well-formed types to account for them:

$$\frac{\Delta, \alpha \vdash A^+}{\Delta \vdash \exists \alpha . A^+}\ \exists \qquad \frac{\Delta, \alpha \vdash A^-}{\Delta \vdash \forall \alpha . A^-}\ \forall$$

The new constructs have the expected type rules, either providing a concrete type in the extensional case or showing that the proof is completable for a fresh type variable name. Notice that our choice of how to polarize the quantifiers agrees with the invertibiilty of their rules. In the following rules, we assume that α, β, and C have the same polarity, so that the substitutions do not introduce ill-formed types:

$$\frac{\Delta \vdash C \quad \Delta; \Psi; \Gamma \vdash A^+[C/\alpha]}{\Delta; \Psi; \Gamma \vdash \exists \alpha . A^+}\ \exists R \qquad \frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma, A^+[\beta/\alpha] \vdash B}{\Delta; \Psi; \Gamma, \exists \alpha . A^+ \vdash B}\ \exists L$$

$$\frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma \vdash A^-[\beta/\alpha]}{\Delta; \Psi; \Gamma \vdash \forall \alpha . A^-}\ \forall R \qquad \frac{\Delta \vdash C \quad \Delta; \Psi; \Gamma, A^-[C/\alpha] \vdash B}{\Delta; \Psi; \Gamma, \forall \alpha . A^- \vdash B}\ \forall L$$

Next we extend our process syntax to add one (overloaded) construct for each new logical rule:

$$\begin{array}{rcll} P & ::= & \ldots & \\ & | & \mathsf{send}\ c\ A; P & (\exists R, \forall L) \\ & | & \alpha \leftarrow \mathsf{recv}\ c; P & (\forall R, \exists L) \end{array}$$

The typing rules for these constructs are the obvious Curry-Howard translations:

$$\frac{\Delta \vdash B \quad \Delta; \Psi; \Gamma \vdash P :: c : C^{+}[B/\alpha]}{\Delta; \Psi; \Gamma \vdash \mathsf{send}\ c\ B; P :: c : \exists\alpha.C^{+}}\ \exists R$$

$$\frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma, a : A^{+}[\beta/\alpha] \vdash P :: c : C}{\Delta; \Psi; \Gamma, a : \exists\alpha.A^{+} \vdash \alpha \leftarrow \mathsf{recv}\ a; P :: c : C}\ \exists L$$

$$\frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma \vdash P :: c : C^{-}[\beta/\alpha]}{\Delta; \Psi; \Gamma \vdash \alpha \leftarrow \mathsf{recv}\ c; P :: c : \forall\alpha.C^{-}}\ \forall R$$

$$\frac{\Delta \vdash B \quad \Delta; \Psi; \Gamma, a : A^{-}[B/\alpha] \vdash P :: c : C}{\Delta; \Psi; \Gamma, a : \forall\alpha.A^{-} \vdash \mathsf{send}\ a\ B; P :: c : C}\ \forall L$$

The operational rules will require us to extend our type of messages with types:

$$K ::= \dots \mid A$$

And use this new class of messages to implement type transmission:

$$\begin{aligned}
\textsc{Send}_{\mathsf{type}}: &\quad \mathsf{que}(a, M, c) \otimes \mathsf{exec}_c(\mathsf{send}\ c\ A; P) \multimap \mathsf{que}(a, M\ A, c) \otimes \mathsf{exec}_c(P) \\
\textsc{Recv}_{\mathsf{type}}: &\quad \mathsf{exec}_c(\alpha \leftarrow \mathsf{recv}\ c; P) \otimes \mathsf{que}(c, A\ M, d) \\
&\quad \multimap \mathsf{exec}_c(P[A/\alpha]) \otimes \mathsf{que}(c, M, d)
\end{aligned}$$

While we will not pursue it further, since it makes proving preservation quite complicated, we could also give a type erased version of this, witnessing that storing types at run-time might be unnecessary. On the other hand, in a distributed system it is reasonable to want to perform runtime checks to confirm that types are dynamically respected, which might force us to transmit the whole type.

4.5.3 Theorems

Before we get to more interesting theorems, we need to establish some basic properties of substitution. First substitution preserves well-formed types.

Lemma 22. *If $\Delta, \alpha \vdash A$ and $\Delta \vdash B$, then $\Delta \vdash A[B/\alpha]$.*

Proof. Take a witnessing relation for $\Delta, \alpha \vdash A$ and substitute B for α throughout. This new relation witnesses $\Delta \vdash A[B/\alpha]$. □

Letting $\Psi[B/\alpha]$ and $\Gamma[B/\alpha]$ stand for pointwise substitution, we also have that substitution is truth preserving.

Lemma 23. *If $\Delta, \alpha; \Psi; \Gamma \vdash A$ and $\Delta \vdash B$, then $\Delta; \Psi[B/\alpha]; \Gamma[B/\alpha] \vdash A[B/\alpha]$.*

Proof. By induction on $\Delta, \alpha; \Psi; \Gamma \vdash A$. Other than Id, the cases are either straightforward or follow from the inductive hypothesis. If Id does not use α, it is straightforward. Otherwise, we the results holds due to the reflexivity of type equality. □

Similarly, we can lift substitution to typing judgments.

Lemma 24. *If* $\Delta, \alpha; \Psi; \Gamma \vdash P :: c : C$ *and* $\Delta \vdash A$*, then* $\Delta; \Psi[A/\alpha]; \Gamma[A/\alpha] \vdash P :: c : C[A/\alpha]$.

Proof. As in Lemma 23, but with typing rules rather than logical ones. □

To prove preservation, we need to update the well-formed queue judgment with rules to handle the new type messages:

$$\frac{\emptyset \vdash C \quad \Gamma \vdash M :: A^-[C/\alpha] \leftsquigarrow B}{\Gamma \vdash C\ M :: \forall\alpha.A^- \leftsquigarrow B}\ \forall_{\mathsf{q}} \qquad \frac{\emptyset \vdash C \quad \Gamma \vdash M :: A^-[C/\alpha] \leftsquigarrow B}{\Gamma \vdash C\ M :: \exists\alpha.A^+ \leftsquigarrow B}\ \exists_{\mathsf{q}}$$

Notice that this requires us to only send closed types across channels. Other than needing to use the new type judgment, the well-formed configuration judgment is unchanged.

Unsurprisingly, queue concatenation works even with a new form message.

Lemma 25. *The following rules are admissible:*

$$\frac{\Gamma \vdash M :: A^+ \leftsquigarrow B^+ \quad \Gamma' \vdash M' :: B^+ \leftsquigarrow C}{\Gamma, \Gamma' \vdash M\ M' :: A^+ \leftsquigarrow C}\ \mathit{trans}^+$$

$$\frac{\Gamma \vdash M :: A^- \leftsquigarrow B^- \quad \Gamma' \vdash M' :: B^- \leftsquigarrow C}{\Gamma, \Gamma' \vdash M\ M' :: A^- \leftsquigarrow C}\ \mathit{trans}^-$$

Proof. As in Lemma 4. □

We can then prove preservation.

Theorem 26 (Preservation)**.** *If* $\vdash E$ *and* $E \to E'$*, then* $\vdash E'$

Proof. Follow the structure of Theorem 6. Other than needing Lemma 24, the new constructs behave like other SENDs and RECVs, but let us examine the two right rules to confirm.

Recall that the proof of Lemma 24 cased on the transition rule used after inducting on $\vdash E$.

Case SEND$_{\mathsf{type}}$: By assumption we have:

$$\mathcal{E} = \Gamma, \Gamma', \Gamma'' \vdash E$$

$$\frac{\dfrac{\overbrace{\emptyset; \emptyset; \Gamma' \vdash P :: c : C^+[A/\alpha]}^{\mathcal{P}}}{\emptyset; \emptyset; \Gamma' \vdash \mathsf{send}\ c\ A; P :: c : \exists\alpha.C^+}\ \exists R \qquad \overbrace{\Gamma'' \vdash M : D^+ \leftsquigarrow \exists\alpha.C^+}^{\mathcal{M}} \qquad \mathcal{E}}{\Gamma, d : D^+ \vdash \mathsf{exec}_c(\mathsf{send}\ c\ A; P), \mathsf{que}(d, M, c), E}\ \mathsf{WF}_{\to}$$

We can replace it with the following to finish this case:

$$\dfrac{\mathcal{P} \quad \dfrac{a \notin \Gamma'' \quad \mathcal{B} \quad \dfrac{\dfrac{}{\emptyset \vdash \cdot : C^+[A/\alpha] \leftsquigarrow C^+[A/\alpha]}\ \emptyset_{\mathsf{q}}}{\emptyset \vdash A : \exists\alpha.C^+ \leftsquigarrow C^+[A/\alpha]}\ \exists R}{\Gamma'' \vdash M\ A : D^+ \leftsquigarrow C^+[A/\alpha]}\ \mathsf{trans}^+ \quad \mathcal{E}}{\Gamma, d : D \vdash \mathsf{exec}_c(P), \mathsf{que}(d, M\ A, c), E}\ \mathsf{WF}_{\rightarrow}$$

Case $\textsc{Recv}_{\mathsf{type}}$: By assumption we have:

$$\mathcal{E} = \Gamma, \Gamma', \Gamma'' \vdash E$$

$$\mathcal{Z} = \dfrac{\emptyset \vdash C \quad \overbrace{\Gamma'' \vdash M :: C^-[A/\alpha] \leftsquigarrow D}^{\mathcal{M}}}{\Gamma'' \vdash A\ M :: \forall\alpha.C^- \leftsquigarrow D}\ \forall_{\mathsf{q}}$$

$$\dfrac{\dfrac{\beta \neq \alpha \quad \overbrace{\beta; \emptyset; \Gamma' \vdash P :: c : C^-[\beta/\alpha]}^{\mathcal{P}}}{\emptyset; \emptyset; \Gamma' \vdash \alpha \leftarrow \mathsf{recv}\ c; P :: c : \forall\alpha.C^-}\ \forall R \quad \mathcal{Z} \quad \mathcal{E}}{\Gamma, d : D \vdash \mathsf{exec}_c(\alpha \leftarrow \mathsf{recv}\ c; P), \mathsf{que}(c, A\ M, d), E}\ \mathsf{WF}_{\leftarrow}$$

which we can replace with the following to finish this case:

$$\dfrac{\overbrace{\emptyset; \emptyset; \Gamma' \vdash P :: c : A^-[C/\alpha]}^{\mathcal{P}\text{ with Lemma 24}} \quad \mathcal{Q} \quad \mathcal{E}}{\Gamma, d : D \vdash \mathsf{exec}_c(P), \mathsf{que}(c, M, d), E}\ \mathsf{WF}_{\leftarrow}$$

□

As normal, progress remains somewhat easier.

Theorem 27 (Progress)**.** *If* $\vdash E$ *then either* $E \to E'$ *or* E *is reactive.*

Proof. Reuse the proof of Theorem 8. The rules $\textsc{Send}_{\mathsf{type}}$ and $\textsc{Recv}_{\mathsf{type}}$ are handled like other Sends or Recvs. □

4.5.4 Related Work

Wadler [94] introduces a Curry-Howard interpretation of classical linear logic (also present in Caires et al. [16] with slightly different primitives) with explicit polymorphism analogous to our explicit type-passing constructs. This provides quantification (and, hence, polymorphism) only as part of the of the linear propositions, a cleaner approach. Unfortunately, it does not provide a clean way to share type variables across both sides of the arrow in types like $\{A \leftarrow B\}$, which we found to be commonly needed by our examples. At the expense of more verbose programs, the cleaner system can express many of these examples by forcing rewriting $\forall\alpha^+.\{A \leftarrow B\}$ as the ungainly and operationally awkward type $\{\forall\alpha.\{A \leftarrow B\} \wedge 1\}$. Wadler also shows how to translate a variant of Gay and Vasconcelos [36] into his calculus, but it does not integrate functional and concurrent computation, nor does it combine the linear, affine, and unrestricted modalities we will see later (section 4.6).

4.6 Polarized Adjoint Logic

Before discussing polarized adjoint logic, we pause to review persistence and discuss its reflection in processes and introduce Adjoint Logic.

4.6.1 Categorical Truth

The linear logic proposition $!A$ allows A to be used arbitrarily often in a proof—it functions as an unrestricted resource. In the intuitionistic reconstruction of linear logic [22], $!A$ internalizes a *categorical judgment*. We say that A is *valid* if it is true, and its proof does not depend on any assumptions about the truth of other propositions. Since we are working with a linear hypothetical judgment, this means that the proof of A does not depend on any resources. We further allow hypotheses Υ that are assumed to be valid (rather than merely true), and these are allowed in a proof A *valid*.

$$\Upsilon; \Delta \vdash C$$

The meaning of validity is captured in the following two judgmental rules:

$$\frac{\Upsilon; \emptyset \vdash A \quad \Upsilon, A; \Gamma \vdash C}{\Upsilon; \Gamma \vdash C}\ \textsc{Cut}! \qquad \frac{\Upsilon, A; \Gamma, A \vdash C}{\Upsilon, A; \Gamma \vdash C}\ \textsc{Copy}$$

The first, Cut!, states that we are justified in assuming that A is valid if we can prove it without using any resources. The second, Copy, states that we are justified in assuming a copy of the resource A if A is known to be valid (essentially a fusion of $!_{\textsc{Contract}}$ and $!L$ of section 2.4). All the purely linear rules are generalized by adding an unrestricted context Υ which is propagated to all premises.

How do we think of these in terms of processes? We introduce a new form of channel, called a *shared channel* (denoted by u, w) which can be used arbitrarily often in a client, and by arbitrarily many clients. It is offered by a *persistent process*. Operationally, a persistent process offering A along w inputs a fresh linear channel c and spawns a new process P that offers A along c.

We have the following typing rules, first at the level of judgments.

$$\frac{\Upsilon, u : A; \Gamma, a : A \vdash P :: c : C}{\Upsilon, u : A; \Gamma \vdash (a \leftarrow \mathsf{send}\ u; P) :: c : C}\ \textsc{Copy}$$

$$\frac{\Upsilon; \emptyset \vdash P :: a : A \quad \Upsilon, u : A; \Gamma \vdash Q :: c : C}{\Upsilon; \Gamma \vdash u \leftarrow !(a \leftarrow \mathsf{recv}\ u; P); Q :: c : C}\ \textsc{Cut}$$

The Copy rule has a slightly strange process expression,

$$a \leftarrow \mathsf{send}\ u; P$$

It expresses that we send a *new* channel a along u. The continuation P refers to a so it can communicate along this new channel. This pattern will be common for sending fresh channels in a variety of constructs in the remainder of this thesis. This combined step of generating a fresh channel and sending it stands in contrast to the ν construct of more traditional session typed languages and is a result of the intuistionistic tight association between channels and processes.

We see that the Cut! rule incorporates two steps: creating a new shared channel u and then immediately receiving a linear channel a along u. There is no simple way to avoid this, since P in the first premise offers along a linear channel a. We will see alternatives in later in this section.

In the operational semantics we write $!\mathsf{exec}_w(P)$ for a persistent process, offering along shared channel w. In substructural specifications [87], $!\mathsf{exec}_w(P)$ on the left-hand side of a rule means that it has to match a persistent proposition. We therefore do not need to repeat it on the right-hand side: it will continue to appear in the state (i.e., we can use contraction to keep a spare copy). In this section we will give a synchronous semantics, deferring the asynchronous version until subsection 4.6.3. In this notation, the operational semantics is as follows:

$$\begin{array}{ll} \textsc{Copy}: & !\mathsf{exec}_w(a \leftarrow \mathsf{recv}\ w; P) \otimes \mathsf{exec}_c(b \leftarrow \mathsf{send}\ w; Q) \\ & \quad \multimap \exists f, g.\ \mathsf{exec}_f(P[f/a]) \otimes \mathsf{exec}_c(Q[g/b]) \end{array}$$

$$\begin{array}{ll} \textsc{Cut}!: & \mathsf{exec}_c(u \leftarrow !(a \leftarrow \mathsf{recv}\ u; P); Q) \\ & \quad \multimap \exists w.\ !\mathsf{proc}_w(a \leftarrow \mathsf{recv}\ w; P_y) \otimes \mathsf{exec}_c(Q[w/u]) \end{array}$$

The validity judgment realized by persistent processes offering along unrestricted channels can be internalized as a proposition $!A$ with the following rules. Note that the linear context must be empty in the $!R$ rule, since validity is a categorical judgment. Allowing dependence on linear channels would violate their linearity.

$$\frac{\Upsilon; \emptyset \vdash P :: a : A}{\Upsilon; \emptyset \vdash u \leftarrow \mathsf{send}\ c; !(a \leftarrow \mathsf{recv}\ u; P) :: c : !A}\ !R$$

$$\frac{\Upsilon, u : A; \Gamma \vdash Q :: c : C}{\Upsilon; \Gamma, a : !A \vdash u \leftarrow \mathsf{recv}\ a; Q :: c : C}\ !L$$

Again the $!R$ rule combines two steps: sending a new persistent channel u along c and then receiving a linear channel a along u. Operationally:

$$\begin{array}{ll} \textsc{Bang}: & \mathsf{exec}_c(u \leftarrow \mathsf{recv}\ d; Q) \otimes \mathsf{exec}_a(u \leftarrow \mathsf{send}\ a; !(b \leftarrow \mathsf{recv}\ u; P)) \\ & \quad \multimap \exists w.\ \mathsf{exec}_c(Q[w/u]) \otimes !\mathsf{proc}_w(b \leftarrow \mathsf{recv}\ w; P) \end{array}$$

As expected, the persistent process spawned by the Bang computation rule has exactly the same form as the one spawned by CUT!, because a linear cut for a proposition $!A$ becomes a persistent cut for a proposition A.

Let's analyze the two-step rule in more detail.

$$\frac{\Upsilon; \emptyset \vdash P :: a : A}{\Upsilon; \emptyset \vdash u \leftarrow \mathsf{send}\ c\ !(a \leftarrow \mathsf{recv}\ u; P) :: c : !A}\ !R$$

The judgment A *valid* (corresponding to an unrestricted hypothesis $u{:}A$) is elided on the right-hand side: we jump directly from the truth of $!A$ to the truth of A. Writing it out as an intermediate step appears entirely reasonable. We do not even mention the linear hypotheses in the intermediate step, since the validity of A depends only on assumptions of validity in Γ.

$$\frac{\dfrac{\Upsilon; \emptyset \vdash P :: a : A}{\Upsilon \vdash a \leftarrow \mathsf{recv}\ u; P :: u : A}\ \text{VALID}}{\Upsilon; \emptyset \vdash u \leftarrow \mathsf{send}\ c; !(a \leftarrow \mathsf{recv}\ u; P) :: c : !A}\ !R$$

We emphasize that $!A$ is positive (in the sense of polarized logic), so it corresponds to a send, while A *valid* is negative as a judgment, so it corresponds to a receive. In the next section we elevate this from a judgmental to a first-class logical step.

Revisiting the example, recall that if we are the client of a channel with the type `queue` A, we must use this channel. This means we have to explicitly dequeue all its elements. In fact, we have to explicitly consume each of the elements as well, since they are also linear. However, if we know that each element in the queue is in fact unrestricted, we can destroy it recursively with the following program.

```
destroy : {1 ← queue (!A)}

c ← destroy ⤙ q =
  send q inr;
  case q of
   inl → wait q; close c
   inr → a ← recv q;          % obtain element a
         u ← recv a;          % receive shared channel u, using a
         c ← destroy ⤙ q      % recurse, ignoring u
```

Unfortunately, this forces us to store unrestricted channels instead of the linear channels that our `queue` previously stored. Instead, we can use an unrestricted destructor process, with type $!(A \multimap 1)$, to destroy each individually stored queue. Since this destructor can be copied as many or few times as we wish, we know that we will always be able to produce a destructor for each queue element, regardless of how many there are. Notice that $!(A \multimap 1)$ denotes a persistent channel with type $(A \multimap 1)$ not a linear channel of type $!(A \multimap 1)$.

```
destroy : {1 ← queue(A); !(A ⊸ 1)}
c ← destroy ⤙ q u =
    send q inl;
    case q of
    | none → wait q; close c
    | some → a ← recv q;
             d ← copy u;
             send d a;
             c ← destroy ⤙ q u
```

4.6.2 Adjoint Logic

Adjoint logic is based on the idea that instead of a modality like $!A$ that remains within a given language of propositions, we have two mutually dependent languages and *two* modalities going back and forth between them. For this to make sense, the operators have to satisfy certain properties that pertain to the semantics of the two languages. We have in fact three language layers, which we call *linear propositions* A_{L}, *affine propositions* A_{F}, and *unrestricted propositions* A_{U}. They are characterized by the structural properties they satisfy: linear propositions are subject to none (they must be used exactly once), affine proposition can be weakened (they can be used at most once), and unrestricted propositions can be contracted and weakened (they can be used arbitrarily often). The order of propositions in the context matters for none of them. The hierarchy of structural properties is reflected in a hierarchy of modes of truth:

$$\mathrm{U} > \mathrm{F} > \mathrm{L}$$

U is *stronger* than F in the sense that unrestricted hypotheses can be used to prove affine conclusions, but not vice versa, and similarly for the other relations. Contexts Γ combine assumptions with all modes. We write $\geq$ for the reflexive and transitive closure of $>$ and define

$$\Gamma \geq k \quad \text{if } m \geq k \text{ for every } B_m \text{ in } \Gamma$$

and

$$\Gamma \vdash A_k \qquad \text{presupposes } \Gamma \geq k$$

We use the notation $\uparrow_k^m A_k$ for an operator going from mode k *up* to mode m, and $\downarrow_k^m A_m$ for an operator going *down* from mode m to mode k. In both cases we presuppose $m > k$.

Taking this approach we obtain the following language:

$$\begin{array}{llcll}
\text{Modes} & m, k, r & ::= & \text{U} \mid \text{F} \mid \text{L} & \\
\text{Propositions} & A_m, B_m & ::= & 1_m \mid A_m \otimes_m B_m \mid A_m \oplus_m B_m \mid \tau \wedge_m B_m & \\
 & & \mid & A_m \multimap_m B_m \mid A_m \&_m B_m \mid \tau \supset_m B_m & \\
 & & \mid & {\uparrow}_k^m A_k & (m > k) \\
 & & \mid & {\downarrow}_m^r A_r & (r > m)
\end{array}$$

Because both $!A$ and A are linear propositions the exponential $!A$ decomposes into two connectives:

$$!A = {\downarrow}_{\text{L}}^{\text{U}}, {\uparrow}_{\text{L}}^{\text{U}} A_{\text{L}}$$

Because linear and affine propositions behave essentially the same way except that affine channels need not be used, we reuse all the same syntax (both for propositions and for process expressions) at these two layers. Unrestricted propositions would behave quite differently, so we specify that there are no unrestricted propositions besides ${\uparrow}_{\text{L}}^{\text{U}} A_{\text{L}}$ and ${\uparrow}_{\text{F}}^{\text{U}} A_{\text{F}}$.

In the following logical rules we always presuppose that the sequent in the conclusion is well-formed and add enough conditions to verify the presupposition in the premises.

$$\frac{\Gamma \vdash A_k}{\Gamma \vdash {\uparrow}_k^m A_k}\;{\uparrow}R \qquad\qquad \frac{k \geq r \quad \Gamma, A_k \vdash C_r}{\Gamma, {\uparrow}_k^m A_k \vdash C_r}\;{\uparrow}L$$

$$\frac{\Gamma \geq m \quad \Gamma \vdash A_m}{\Gamma \vdash {\downarrow}_k^m A_m}\;{\downarrow}R \qquad\qquad \frac{\Gamma, A_m \vdash C_r}{\Gamma, {\downarrow}_k^m A_m \vdash C_r}\;{\downarrow}L$$

The rules with no condition on the modes are invertible, while the others are not invertible. This means ${\uparrow}A$ is *negative* while ${\downarrow}A$ is *positive* (in the terminology of section 4.1). We already noted that processes offering a negative type receive, while processes offering a positive type send. But what do we send or receive? Thinking of channels as intrinsically linear, affine, or shared suggests that we should send and receive fresh channels of different modes. Following this reasoning we obtain, after denoting channel modalities via subscripts:

$$\frac{\Gamma \vdash P :: a_k : A_k}{\Gamma \vdash a_k \leftarrow \mathsf{recv}\ b_m; P :: b_m : {\uparrow}_k^m A_k}\;{\uparrow}R$$

$$\frac{k \geq r \quad \Psi, a_k : A_k \vdash Q :: c_r : C_r}{\Gamma, b_m : {\uparrow}_k^m A_k \vdash a_k \leftarrow \mathsf{send}\ b_m; Q :: c_r : C_r}\;{\uparrow}L$$

While we annotate each channel with its mode, we should note that it may not be strictly necessary. Operationally:

$$\begin{array}{lll}
\text{UP}_k^m & : & \mathsf{exec}_{a_r}(a_k \leftarrow \mathsf{send}\ c_m; Q) \otimes \mathsf{exec}_{c_m}(b_k \leftarrow \mathsf{exec}\ c_m; P) \\
 & & \multimap \exists c_k.\ \mathsf{exec}_{a_r}(Q[c_k/a_k]) \otimes \mathsf{exec}_{c_k}(P[c_k/b_k])
\end{array}$$

And for the other modality:

$$\frac{\Gamma \geq m \quad \Psi \vdash Q_{x_m} :: a_m : A_m}{\Gamma \vdash a_m \leftarrow \mathsf{send}\ b_k ; Q :: b_k : {\downarrow}_k^m A_m} \ {\downarrow}R$$

$$\frac{\Gamma, a_m : A_m \vdash P :: c_r : C_r}{\Gamma, b_k : {\downarrow}_k^m A_m \vdash a_m \leftarrow \mathsf{recv}\ b_k ; P :: c_r : C_r} \ {\downarrow}L$$

Operationally:

$$\begin{array}{lcl} \textsc{down}_k^m & : & \mathsf{exec}_{a_r}(a_m \leftarrow \mathsf{recv}\ c_k ; P) \otimes \mathsf{exec}_{c_k}(b_m \leftarrow \mathsf{send}\ c_k ; Q) \\ & & \multimap \exists d_m . \mathsf{exec}_{a_r}(P[d_m/a_m]) \otimes \mathsf{exec}_{d_m}(Q[d_m/b_m]) \end{array}$$

Since processes offering along unrestricted channels are persistent, we use here the (admittedly dangerous) notational convention that all processes offering along unrestricted channels c_{U} are implicitly marked persistent. In particular, we should read $\textsc{up}_k^{\text{U}}$ and $\textsc{down}_k^{\text{U}}$ as

$$\begin{array}{lcl} \textsc{up}_k^{\text{U}} & : & \mathsf{exec}_{a_r}(a_k \leftarrow \mathsf{send}\ c_{\text{U}} ; Q) \otimes !\mathsf{exec}_{c_{\text{U}}}(b_k \leftarrow \mathsf{exec}\ c_{\text{U}} ; P) \\ & & \multimap \exists c_k .\ \mathsf{exec}_{a_r}(Q[c_k/a_k]) \otimes \mathsf{exec}_{c_k}(P[c_k/b_k]) \\ \textsc{down}_k^{\text{U}} & : & \mathsf{exec}_{a_r}(a_{\text{U}} \leftarrow \mathsf{recv}\ c_k ; P) \otimes \mathsf{exec}_{c_k}(b_{\text{U}} \leftarrow \mathsf{send}\ c_k ; Q) \\ & & \multimap \exists d_{\text{U}} . \mathsf{exec}_{a_r}(P[d_{\text{U}}/a_{\text{U}}]) \otimes !\mathsf{exec}_{d_{\text{U}}}(Q[d_{\text{U}}/b_{\text{U}}]) \end{array}$$

At this point we have achieved that every logical connective, including the up and down modalities, correspond to exactly one matching send and receive action. Moreover, as we can check, the compound rules for $!A$ decompose into individual steps.

Returning to our queue example (Figure 4.14), we can now specify that our queue is supposed to be affine, that is, that we can safely decide to ignore it. We annotate defined types and type variables with their mode (U, F, or L), but we overload the logical connectives since their meanings, when defined, are consistent. The elements of an affine queue should also be affine. If we make them linear, as in

$$\mathtt{Queue}_{\text{F}}\ A_{\text{L}} = ({\uparrow}_{\text{L}}^{\text{F}} A_{\text{L}} \multimap \mathtt{Queue}\ A_{\text{L}}) \& (1 \oplus ({\uparrow}_{\text{L}}^{\text{F}} A_{\text{L}} \otimes \mathtt{Queue}\ A_{\text{L}}))$$

then we could never use $a_{\text{F}} : {\uparrow}_{\text{L}}^{\text{F}} A_{\text{L}}$ in a process offering an affine service (rule ${\uparrow}L$) since $\text{L} \not\geq \text{F}$, i.e., we could only define "queues" that discarded everything placed in them. So instead we should define an affine queue as

$$\mathtt{Queue}_{\text{F}}\ A_{\text{F}} = (A_{\text{F}} \multimap \mathtt{Queue}\ A_{\text{F}}) \& (1 \oplus (A_{\text{F}} \otimes \mathtt{Queue}_{\text{F}}\ A_{\text{F}}))$$

so that all types in the definition (including A_{F}) are affine. Now we no longer need to explicitly destroy a queue, we can just abandon it and the runtime system will deallocate it by a form of garbage collection (subsection 4.6.6).

If we want to enforce a linear discipline, destroying a queue with linear elements will have to rely on a consumer for the elements of the queue. This consumer must be unrestricted because it is used for each element. Channels are linear by default, so in the example we only annotate affine and unrestricted channels with their mode.

```
destroy : $\{1 \leftarrow \texttt{Queue}_{\mathrm{L}}\, A_{\mathrm{L}}, {\uparrow}_{\mathrm{L}}^{\mathrm{U}}(A_{\mathrm{L}} \multimap 1)\}$

$c \leftarrow$ destroy $\multimap\!\!\prec q\ u_{\mathrm{U}} =$
    send $q$ inr;
    case $q$ of
     inl $\rightarrow$ wait $q$; close $c$
     inr $\rightarrow$ $a \leftarrow$ recv $q$;
            $d \leftarrow$ send $u_{\mathrm{U}}$;                % obtain instance $d$ of $u_{\mathrm{U}}$
            send $d$ $a$; wait $d$;          % use $d$ to consume $a$
            $c \leftarrow$ destroy $\multimap\!\!\prec q\ u_{\mathrm{U}}$       % recurse, reusing $u_{\mathrm{U}}$
```

4.6.3 Polarized Adjoint Logic

Now we are ready to combine the ideas from adjoint logic in subsection 4.6.2 with polarization in section 4.1. Amazingly, they are fully consistent. The two differences to the polarized presentation are that (a) the modalities go between positive and negative propositions (already anticipated by the fact that $\downarrow$ is positive and $\uparrow$ is negative), and (b) the modalities $\downarrow_k^m A$ and $\uparrow_k^m A$ allow $m \geq k$ rather than presupposing $m > k$ as before. We no longer index the connectives (except, occasionally, for clarity), overloading their meaning at the different layers.

Pos. props	A_m^+, B_m^+	$::=$	1	send end and terminate
		$\mid$	$A_m^+ \otimes B_m^+$	send channel of type A_m^+
		$\mid$	$A_m^+ \oplus B_m^+$	send inl or inr
		$\mid$	$\tau \wedge B_m^+$	send value of type τ
		$\mid$	$\downarrow_m^r A_r^-$	$(r \geq m)$, send shift, then receive
Neg. props	A_m^-, B_m^-	$::=$	$A_m^+ \multimap B_m^-$	receive channel of type A_m^+
		$\mid$	$A_m^- \& B_m^-$	receive inl or inr
		$\mid$	$\tau \supset B_m^-$	receive value of type τ
		$\mid$	$\uparrow_k^m A_k^+$	$(m \geq k)$, receive shift, then send

A shift staying at the same level just changes the polarity but is otherwise not subject to any restrictions. We can see this from the rules, now annotated with a polarity: if $m = k$ in $\uparrow L$, then $k \geq r$ by presupposition since $(\Gamma, \uparrow_k^m A_k^+) \geq r$.

Similarly, in $\downarrow R$, $\Gamma \geq m$ by presupposition if $m = k$.

$$\frac{\Delta; \Psi; \Gamma \vdash A_k^+}{\Delta; \Psi; \Gamma \vdash \uparrow_k^m A_k^+} \uparrow R \qquad \frac{k \geq r \quad \Delta; \Psi; \Gamma, A_k^+ \vdash C_r}{\Delta; \Psi; \Gamma, \uparrow_k^m A_k^+ \vdash C_r} \uparrow L$$

$$\frac{\Gamma \geq m \quad \Delta; \Psi; \Gamma \vdash A_m^-}{\Delta; \Psi; \Gamma \vdash \downarrow_k^m A_m^-} \downarrow R \qquad \frac{\Delta; \Psi; \Gamma, A_m^- \vdash C_r}{\Delta; \Psi; \Gamma, \downarrow_k^m A_m^- \vdash C_r} \downarrow L$$

Adding process expressions in a straightforward manner generalizes the shift to carry a fresh channel because there may now be a change in modes associated with the shift. We have the following new syntax

$$\begin{array}{lcll} P, Q & ::= & \mathsf{shift}\ a_k \leftarrow \mathsf{send}\ c_m; P & (\text{send } \mathsf{shift}\ a_k, \text{ then recv. along } a_k \text{ in } P) \\ & \mid & \mathsf{shift}\ a_k \leftarrow \mathsf{recv}\ c_m; Q & (\text{recv. } \mathsf{shift}\ a_k, \text{ then send along } a_k \text{ in } Q) \end{array}$$

and the modified rules (we will allow channels with modality U to be duplicated/contracted as needed without creating variant rules):

$$\frac{\Delta; \Psi; \Gamma \vdash P :: a_k : A_k^+}{\Delta; \Psi; \Gamma \vdash \mathsf{shift}\ a_k \leftarrow \mathsf{recv}\ b_m; P :: b_m : \uparrow_k^m A_k^+} \uparrow R$$

$$\frac{k \geq r \quad \Delta; \Psi; \Gamma, a_k : A_k^+ \vdash Q :: c_r : C_r}{\Delta; \Psi; \Gamma, b_m : \uparrow_k^m A_k^+ \vdash \mathsf{shift}\ a_k \leftarrow \mathsf{send}\ b_m; Q :: c_r : C_r} \uparrow L$$

$$\frac{\Gamma \geq m \quad \Delta; \Psi; \Gamma \vdash Q :: a_m : A_m^-}{\Delta; \Psi; \Gamma \vdash \mathsf{shift}\ a_m \leftarrow \mathsf{send}\ b_k; Q :: b_k : \downarrow_k^m A_m^-} \downarrow R$$

$$\frac{\Delta; \Psi; \Gamma, a_m : A_m^- \vdash P :: c_r : C_r}{\Delta; \Psi; \Gamma, b_k : \downarrow_k^m A_m^- \vdash \mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ b_k; P :: c_r : C_r} \downarrow L$$

Operationally, we generalize the basic shift message with the following:

$$K ::= \ldots \mid \mathsf{shift}(c)$$

and generalize the rules for sending and receiving it:

$$\begin{array}{ll} \textsc{Send}_{\mathsf{shift}}: & \mathsf{que}(a_k, M, c_k) \otimes \mathsf{exec}_{c_k}(\mathsf{shift}\ b_m \leftarrow \mathsf{send}\ c_k; P) \\ & \quad \multimap \exists d_m, f_m. \mathsf{que}(a_k, M\ \mathsf{shift}(f_m), d_m) \otimes \mathsf{exec}(P[d_m/b_m]) \\ \textsc{Recv}_{\mathsf{shift}}: & \mathsf{exec}_{c_k}(\mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ c_k; P) \otimes \mathsf{que}(c_k, \mathsf{shift}(b_m), d_m) \\ & \quad \multimap \mathsf{que}(d_m, \cdot, b_m) \otimes \mathsf{exec}_{b_m}(P[b_m/a_m]) \end{array}$$

As pointed out in subsection 4.6.2, we have to assume that processes that offer along an unrestricted channel c_{U} are persistent. Also, this formulation introduces a new channel even when $m = k$, a slight redundancy best avoided in the syntax and semantics of a real implementation. Even when going between linear and affine channels, creating new channels might be avoided in favor of just changing some channel property.

Once again rewriting the linear version of the example, forcing synchronization.

$$\texttt{queue}^{-}\ A^{+} = (A^{+} \multimap \uparrow^{\mathrm{L}}_{\mathrm{L}}\downarrow^{\mathrm{L}}_{\mathrm{L}}(\texttt{queue}^{-}\ A^{+}))\&\uparrow^{\mathrm{L}}_{\mathrm{L}}(1 \oplus (A^{+} \otimes \downarrow^{\mathrm{L}}_{\mathrm{L}}\texttt{queue}^{-}\ A^{+}))$$

```
$\texttt{empty} : \{\texttt{queue}^{-}\ A^{+}\}$

$c \leftarrow \texttt{empty} =$
  case $c$ of
  | enq → $a \leftarrow$ recv $c$;
          shift $c \leftarrow$ recv $c$
          shift $c \leftarrow$ send $c$
          $b \leftarrow \texttt{empty}$;
          $c \leftarrow \texttt{elem} \multimap\!\!\prec a, b$
  | deq → shift $c \leftarrow$ recv $c$
          send $c$ inl;
          close $c$
```

```
$\texttt{elem} : \{\texttt{queue}^{-}\ A^{+} \leftarrow A^{+}; \texttt{queue}^{-}\ A^{+}\}$

$c \leftarrow \texttt{elem} \multimap\!\!\prec a\ d =$
  case $c$ of
    inl → $b \leftarrow$ recv $c$;
          shift $c \leftarrow$ recv $c$;   % shift to send
          shift $c \leftarrow$ send $c$;   % send ack
          send $d$ inl;
          send $d$ $b$;
          shift $d \leftarrow$ send $d$;   % shift to recv
          shift $d \leftarrow$ recv $d$;   % recv ack
          $c \leftarrow \texttt{elem} \multimap\!\!\prec x, d$
    inr → shift $c \leftarrow$ recv $c$;   % shift to send
          send $c$ inl;
          send $c$ $a$;
          shift $c \leftarrow$ send $c$;   % shift to recv
          $c \leftarrow d$
```

And destroying a linear queue with affine elements:

```
$\texttt{destroy} : \{1 \leftarrow \mathsf{queue}\,(\downarrow^{\mathrm{F}}_{\mathrm{L}} A_{\mathrm{F}})\}$

$c \leftarrow \texttt{destroy} \multimap\!\!\prec q =$
  send $q$ inr;
  shift $q \leftarrow$ send $q$;          % shift to recv
  case $q$ of
    inl → wait $q$;  close $c$
    inr → $x \leftarrow$ recv $q$;          % obtain element $x$
          shift $a_{\mathrm{F}} \leftarrow$ recv $x$;   % obtain affine $a_{\mathrm{F}}$, consuming $x$
          shift $q \leftarrow$ send $q$;    % shift to recv
          $c \leftarrow \texttt{destroy} \multimap\!\!\prec q$   % recurse, ignoring $a_{\mathrm{F}}$
```

4.6.4 Theorems

Our prior well-typed configuration judgment can almost be reused by merely letting the Γs involved include mappings for our new types, but we need a few slight changes. Since affine and persistent channels may be weakened, we want well-typed configurations to be able to have some residual affine or persistent processes that are "unreachable" from the important part of the configuration (i.e., the part demanded by the initial Γ). To enable this we replace $\mathsf{WF}_{\bullet}$ with the following:

$$\frac{E \geq \text{F}}{\Gamma^{\bullet} \vdash E}\ \mathsf{WF}_{\bullet}$$

where $E \geq \text{F}$ denotes that all processes and queues involved are either affine or persistent. Additionally, we need to slightly update our well-typed queue judgment to account for the new style of shift:

$$\frac{}{\emptyset \vdash \mathsf{shift}(c_m) : \uparrow_m^k A_m^- \leftsquigarrow A_m^-}\ \uparrow_{\mathsf{q}} \qquad \frac{}{\emptyset \vdash \mathsf{shift}(c_k) : \downarrow_m^k A_k^+ \leftsquigarrow A_k^+}\ \downarrow_{\mathsf{q}}$$

Lemma 28 (Preservation)**.** *If* $\Gamma \vdash E$ *and* $E \to E'$*, then* $\Gamma \vdash E'$*.*

Proof. Most of the proof can follow Theorem 26, adding uses of affine type rules where appropriate, but we need to examine the altered $\textsc{Send}_{\mathsf{shift}}$ and $\textsc{Recv}_{\mathsf{shift}}$ in more detail.

Case $\textsc{Send}_{\mathsf{shift}}$: There are two different cases corresponding to whether we use $\downarrow R$ or $\uparrow L$. For the $\downarrow R$ case, we are given:

$$\mathcal{E} = \Gamma, \Gamma', \Gamma'' \vdash E$$

$$\mathcal{M} = \Gamma'' \vdash M : A_k^+ \leftsquigarrow \downarrow_k^m B_m^-$$

$$\frac{\dfrac{\Gamma' \geq m \quad \overbrace{\emptyset; \emptyset; \Gamma' \vdash P :: b_m : B_m^-}^{\mathcal{P}}}{\emptyset; \emptyset; \Gamma' \vdash \mathsf{shift}\, b_m \leftarrow \mathsf{send}\, c_k; P :: c_k : \downarrow_k^m B_m^-}\ \downarrow R \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, a_k : A_k^+ \vdash \mathsf{que}(a_k, M, c_k), \mathsf{exec}_{c_k}(\mathsf{shift}\ b_m \leftarrow \mathsf{send}\ c_k; P), E}\ \mathsf{WF}_{\to}$$

which we replace with the following where d_m and f_m are fresh (notice that $\mathcal{P}[d_m/b_m]$ is only well-formed due to $\Gamma' \geq m$):

$$\frac{\mathcal{P}[d_m/b_m] \quad \dfrac{\mathcal{M} \quad \dfrac{}{\emptyset \vdash \mathsf{shift}(f_m) : \downarrow_k^m B_m^- \leftsquigarrow B_m^-}\ \downarrow_{\mathsf{q}}}{\Gamma'' \vdash M\ \mathsf{shift}(f_m) : A_k^+ \leftsquigarrow B_m^-}\ \mathsf{trans}^+ \quad \mathcal{E}}{\Gamma, a_k{:}A_k^+ \vdash \mathsf{que}(a_k, M\, \mathsf{shift}(f_m), d_m), \mathsf{exec}_{d_m}(P[d_m/b_m]), E}\ \mathsf{WF}_{\to}$$

For $\uparrow L$ we are given, after using Lemma 5 (showing only the case for an initial $\mathsf{WF}_\rightarrow$):

$$\mathcal{M} = \Gamma_3 \vdash M : A_r^+ \leftsquigarrow C_r^+$$

$$\dfrac{\overbrace{\emptyset;\emptyset;\Gamma_4 \vdash Q :: d_m : D_m^-}^{\mathcal{Q}} \quad \overbrace{\Gamma_5 \vdash M' : D_m^- \leftsquigarrow \uparrow_k^m B_k^+}^{\mathcal{M}'} \quad \overbrace{\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4, \Gamma_5 \vdash E}^{\mathcal{E}}}{\underbrace{\Gamma_1, \Gamma_2, \Gamma_3, c_m : \uparrow_k^m B_k^+ \vdash \mathsf{que}(d_m, M', c_m), \mathsf{exec}_{d_m}(Q), E}_{\mathcal{Z}}}\ \mathsf{WF}_\leftarrow$$

$$\dfrac{\dfrac{k \geq r \quad \overbrace{\emptyset;\emptyset;\Gamma_2, b_k : B_k^+ \vdash P :: a'_r : C_r^+}^{\mathcal{P}}}{\emptyset;\emptyset;\Gamma_2, c_m : \uparrow_k^m B_k^+ \vdash \mathsf{shift}\ b_k \leftarrow \mathsf{send}\ c_m; P :: a'_r : C_r^+}\ \uparrow L \quad \mathcal{M} \quad \mathcal{Z}}{\Gamma_1, a_r : A_r^+ \vdash \begin{array}{l} \mathsf{que}(a_r, M, a'_r), \mathsf{exec}_{a'_r}(\mathsf{shift}\ b_k \leftarrow \mathsf{send}\ c_m; P), \\ \quad \mathsf{que}(d_m, M', c_m), \mathsf{exec}_{d_m}(Q), E \end{array}}\ \mathsf{WF}_\rightarrow$$

which we replace with, where f_k and g_k are fresh:

$$\dfrac{\mathcal{Q} \quad \dfrac{\mathcal{M}' \quad \dfrac{}{\emptyset \vdash \mathsf{shift}(f_k) : \uparrow_m^k B_k^+}\ \uparrow_{\mathsf{q}}}{\Gamma_5 \vdash D_m^- \leftsquigarrow B_k^+}\ \mathsf{trans}^- \quad \mathcal{E}}{\underbrace{\Gamma_1, \Gamma_2, g_k : B_k^+ \vdash \mathsf{que}(d_m, M'\ \mathsf{shift}(f_k), g_k), \mathsf{exec}_{d_m}(Q), E}_{\mathcal{W}}}\ \mathsf{WF}_\leftarrow$$

$$\dfrac{\mathcal{P}[g_k/b_k] \quad \mathcal{M} \quad \mathcal{W}}{\Gamma_1, a_r : A_r^+ \vdash \begin{array}{l} \mathsf{que}(a_r, M, a'_r), \mathsf{exec}_{a'_r}(P[g_k/b_k]), \\ \quad \mathsf{que}(d_m, M'\ \mathsf{shift}(f_k), g_k), \mathsf{exec}_{d_m}(Q), E \end{array}}\ \mathsf{WF}_\rightarrow$$

Case $\textsc{Recv}_{\mathsf{shift}}$: There are two cases, one when the process is typed with two for $\uparrow R$ and two for $\downarrow L$. For $\uparrow R$ we are given:

$$\underbrace{\dfrac{}{\emptyset \vdash \mathsf{shift}(b_m) : \uparrow_m^k D_m^+ \leftsquigarrow D_m^+}\ \uparrow_{\mathsf{q}}}_{\mathcal{M}}$$

$$\dfrac{\dfrac{\Gamma' \geq m \quad \overbrace{\emptyset;\emptyset;\Gamma' \vdash P :: a_m : D_m^+}^{\mathcal{P}}}{\emptyset;\emptyset;\Gamma' \vdash \mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ c_k; P :: c_k : \uparrow_m^k A_m^+}\ \uparrow R \quad \mathcal{M} \quad \overbrace{\Gamma, \Gamma' \vdash E}^{\mathcal{E}}}{\Gamma, d_m : D_m^+ \vdash \mathsf{que}(c_k, \mathsf{shift}(b_m), d_m), \mathsf{exec}_{c_k}(\mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ c_k; P), E}\ \mathsf{WF}_\leftarrow$$

which we replace with:

$$\dfrac{\mathcal{P}[b_m/a_m] \quad \dfrac{}{\emptyset \vdash \cdot : D_m^+ \leftsquigarrow D_m^+}\ \emptyset_{\mathsf{q}} \quad \mathcal{E}}{\Gamma, d_m : D_m^+ \vdash \mathsf{que}(d_m, \cdot, b_m), \mathsf{exec}_{b_m}(P), E}\ \mathsf{WF}_\leftarrow$$

For $\downarrow L$ we are given, after uses of Lemma 5 (showing only the initially

$\mathsf{WF}_{\rightarrow}$ case):

$$\mathcal{M} = \Gamma_3 \vdash M : A_r^+ \leftrightsquigarrow C_r^+$$

$$\mathcal{E} = \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash E$$

$$\underbrace{\dfrac{\overbrace{\emptyset;\emptyset;\Gamma_4 \vdash Q :: f_m{:}B_m^-}^{\mathcal{Q}} \quad \dfrac{}{\emptyset \vdash \mathsf{shift}(d_m){:}{\downarrow_k^m} B_m^- \leftrightsquigarrow B_m^-}{\downarrow_{\mathsf{q}}} \quad \mathcal{E}}{\Gamma_1,\Gamma_2,\Gamma_3,c_k : {\downarrow_k^m} B_m^- \vdash \mathsf{que}(c_k,\mathsf{shift}(d_m),f_m),\mathsf{exec}_{f_m}(Q),E}\,\mathsf{WF}_{\rightarrow}}_{\mathcal{Z}}$$

$$\dfrac{\dfrac{\overbrace{\emptyset;\emptyset;\Gamma_2,b_m : B_m^- \vdash P :: a'_r : C_r^+}^{\mathcal{P}}}{\emptyset;\emptyset;\Gamma_2,c_k : {\downarrow_k^m} B_m^- \vdash \mathsf{shift}\ b_m \leftarrow \mathsf{recv}\ c_k; P :: a'_r : C_r^+}{\uparrow}L \quad \mathcal{M} \quad \mathcal{Z}}{\Gamma_1, a_r : A_r^+ \vdash \begin{array}{c}\mathsf{que}(a_r,M,a'_r),\mathsf{exec}_{a'_r}(\mathsf{shift}\ b_m \leftarrow \mathsf{recv}\ c_k;P),\\ \mathsf{que}(c_k,\mathsf{shift}(d_m),f_m),\mathsf{exec}_{f_m}(Q),E\end{array}}\,\mathsf{WF}_{\rightarrow}$$

which we replace with:

$$\dfrac{\mathcal{P}[d_m/b_m] \quad \mathcal{M} \quad \dfrac{\mathcal{Q} \quad \dfrac{}{\emptyset \vdash \cdot B_m^- \leftrightsquigarrow B_m^-}\emptyset_{\mathsf{q}} \quad \mathcal{E}}{\Gamma_1;\Gamma_2;\Gamma_3,d_m : B_m^- \vdash \mathsf{que}(f_m,\cdot,d_m),\mathsf{exec}_{f_m}(Q),E}\,\mathsf{WF}_{\leftarrow}}{\Gamma_1, a_r : A_r^+ \vdash \begin{array}{c}\mathsf{que}(a_r,M,a'_r),\mathsf{exec}_{a'_r}(P[d_m/b_m]),\\ \mathsf{que}(f_m,\cdot,d_m),\mathsf{exec}_{f_m}(Q),E\end{array}}\,\mathsf{WF}_{\rightarrow}$$

□

Unfortunately, unlike some presentations [75], this version does not make it easy to see when persistence is used: none of the four cases examined in the previous proof can directly can directly contracted. However, because there are no propositions at mode U other than ${\downarrow_m^{\textsc{u}}} A_m^-$, we know that after using $\textsc{Recv}_{\mathsf{shift}}$ where the new channel is persistent, the channel involved must be of the form $\mathsf{que}(a_{\textsc{u}},\cdot,b_{\textsc{u}})$ for some $a_{\textsc{u}}$ and $b_{\textsc{u}}$ (i.e., by using inversion on $\leftrightsquigarrow$ in the second set of cases in the above proof). At that point, both the queue and the persistent process can be safely contracted (i.e., copied).

A proof of $\Gamma \vdash E$ is called *garbage free* if its use of $\mathsf{WF}_{\bullet}$ is used with $\cdot$.

Lemma 29 (Progress)**.** *If $\Gamma \vdash E$ and has a garbage free proof, then either $E \rightarrow E'$ or E is reactive.*

Proof. Reuse the proof from Theorem 27. □

Notice that the requirement to be garbage free is important here. Without it we run the risk of some garbage processes either making progress while our program make no meaningful progress or some garbage processes that are stuck without being reactive. Additionally, by preservation, garbage generated by a well-typed configuration cannot do anything particularly interesting to the non-garbage portions of the configuration.

4.6.5 Sequent Calculus for Polarized Adjoint Logic

We summarize the sequent calculus rules for polarized adjoint logic in Figure 4.15, omitting the uninteresting rules for existential and universal quantification. However, we have removed the stipulation that the only unrestricted propositions are $\uparrow^{\text{U}}_{m} A^{+}_{m}$, thereby making our theorem slightly more general at the expense of a nonstandard notation for intuitionistic connectives such as $A_{\text{U}} \multimap_{\text{U}} B_{\text{U}}$ for $A \supset B$. Additionally, we restrict our attention to the system without recursive types or $\{\}I/\{\}E$ (i.e., we use a traditional notion of cut in this section).

We have the following theorem.

Theorem 30.

1. *Cut is admissible in the system without cut.*
2. *Identity is admissible for arbitrary propositions in the system with the identity restricted to atomic propositions and without cut.*

Proof. The admissibility of cut follows by a nested structural induction, first on the cut formula A, second simultaneously on the proofs of the left and right premise. We liberally use a lemma which states that we can weaken a proof with affine and unrestricted hypotheses without changing its structure and we exploit the transitivity of $\geq$. See [22, 77] for analogous proofs.

The admissibility of identity at A follows by a simple structural induction on the proposition A, exploiting the reflexivity of $\geq$ in one critical case. □

A simple corollary is *cut elimination*, stating that every provable sequent has a cut-free proof. Cut elimination of the logic is the central reason why the session-typed processes assigned to these rules satisfy the by now expected properties of *session fidelity* (processes are guaranteed to follow the behavior prescribed by the session type) and *global progress* (a closed process network of type $c_0 : 1$ can either take a step will send end along c_0). In addition, we also have *productivity* (processes will eventually perform the action prescribed by the session type) and *termination* if recursive processes are appropriately restricted. The proofs of these properties closely follow those in the literature for related systems [16, 89, 91], so we do not formally state or prove them here.

4.6.6 Garbage Collection

In our definition of garbage free proofs, we implicitly declared that processes and queues left over after forming a well-typed proof were garbage. To rephrase this in more traditional garbage collection terms, the domain of Γ, in $\Gamma \vdash E$, is the root set for any collector we might build. However, within this very general scheme we have a lot of flexibility. For example, we could, essentially, do no collection at all, choosing only to kill all processes after the initial process has terminated or we could garbage collect after every transition. In addition to

$$\begin{array}{llll}
m, k, r & ::= & \text{U} \mid \text{F} \mid \text{L} \quad \text{with U} > \text{F} > \text{L} & \\
A^+_m, B^+_m & ::= & \alpha^+_m \mid 1_m \mid A^+_m \otimes_m B^+_m \mid A^+_m \oplus_m B^+_m \mid {\downarrow}^r_m A^-_r & (r \geq m) \\
A^-_m, B^-_m & ::= & \alpha^-_m \mid A^+_m \multimap_m B^-_m \mid A^-_m \&_m B^-_m \mid {\uparrow}^m_k A^+_k & (m \geq k) \\
A_m, B_m, C_m & ::= & A^+_m \mid A^-_m &
\end{array}$$

$$\frac{\Gamma \geq \text{F}}{\Gamma, A_m \vdash A_m}\ \textsc{Id} \qquad \frac{\Gamma \geq m \geq r \quad \Gamma \vdash A_m \quad \Gamma', A_m \vdash C_r}{\Gamma, \Gamma' \vdash C_r}\ \textsc{Cut}$$

$$\frac{\Gamma \vdash A^+_k}{\Gamma \vdash {\uparrow}^m_k A^+_k}\ {\uparrow}R \qquad \frac{k \geq r \quad \Gamma, A^+_k \vdash C_r}{\Gamma, {\uparrow}^m_k A^+_k \vdash C_r}\ {\uparrow}L$$

$$\frac{\Gamma_{\geq m} \vdash A^-_m}{\Gamma \vdash {\downarrow}^m_k A^-_m}\ {\downarrow}R \qquad \frac{\Gamma, A^-_m \vdash C_r}{\Gamma, {\downarrow}^m_k A^-_m \vdash C_r}\ {\downarrow}L$$

$$\frac{\Gamma \geq \text{F}}{\Gamma \vdash 1_m}\ 1R \qquad \frac{\Gamma \vdash C_r}{\Gamma, 1_m \vdash C_r}\ 1L$$

$$\frac{\Gamma \vdash A^+_m \quad \Gamma' \vdash B^+_m}{\Gamma, \Gamma' \vdash A^+_m \otimes_m B^+_m}\ {\otimes}R \qquad \frac{\Gamma, A^+_m, B^+_m \vdash C_r}{\Gamma, A^+_m \otimes_m B^+_m \vdash C_r}\ {\otimes}L$$

$$\frac{\Gamma, A^+_m \vdash B^-_m}{\Gamma \vdash A^+_m \multimap_m B^-_m}\ {\multimap}R \qquad \frac{\Gamma \geq m \quad \Gamma \vdash A^+_m \quad \Gamma', B^-_m \vdash C_r}{\Gamma, \Gamma', A^+_m \multimap_m B^-_m \vdash C_r}\ {\multimap}L$$

$$\frac{\Gamma \vdash A^-_m \quad \Gamma \vdash B^-_m}{\Gamma \vdash A^-_m \&_m B^-_m}\ \&R$$

$$\frac{\Gamma, A^-_m \vdash C_r}{\Gamma, A^-_m \&_m B^-_m \vdash C_r}\ \&L_1 \quad \frac{\Gamma, B^-_m \vdash C_r}{\Gamma, A^-_m \&_m B^-_m \vdash C_r}\ \&L_2$$

$$\frac{\Gamma \vdash A^+_m}{\Gamma \vdash A^+_m \oplus_m B^+_m}\ {\oplus}R_1 \quad \frac{\Gamma \vdash B^+_m}{\Gamma \vdash A^+_m \oplus_m B^+_m}\ {\oplus}R_2$$

$$\frac{\Gamma, A^+_m \vdash C_r \quad \Gamma, B^+_m \vdash C_r}{\Gamma, A^+_m \oplus_m B^+_m \vdash C_r}\ {\oplus}L$$

All judgments $\Gamma \vdash A_m$ presuppose $\Gamma \geq m$.
Γ, Γ' allows contraction of unrestricted A_{U} shared between Γ and Γ'

Figure 4.15: Polarized Adjoint Logic

applying traditional techniques, we can, as other languages incorporating affine types [25, 54] do, also perform collection in a logically motivated way by figuring out where use weakening are permitted (e.g., by examining the calculations of section 5.1).

Another logically motivated way to account for affine process garbage collection is by the following transformation [40] (showing only the affine cases), where we use $A^+ \& B^+$ to abbreviate $\&\{\texttt{inl} : A^+; \texttt{inr} : B^+\}$:

$$\begin{aligned}
&[\![1_{\mathrm{F}}]\!] = 1_{\mathrm{L}} && [\![\tau \wedge_{\mathrm{F}} A^+_{\mathrm{F}}]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} {\downarrow}^{\mathrm{L}}_{\mathrm{L}}(\tau \wedge_{\mathrm{L}} [\![A^+_{\mathrm{F}}]\!]) \\
&[\![\tau \supset_{\mathrm{F}} A^-_{\mathrm{F}}]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} (\tau \supset_{\mathrm{L}} [\![A^-_{\mathrm{F}}]\!]) && [\![A^+_{\mathrm{F}} \otimes_{\mathrm{F}} B^+_{\mathrm{F}}]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} ([\![A^+_{\mathrm{F}}]\!] \&_{\mathrm{L}} [\![B^+_{\mathrm{F}}]\!]) \\
&[\![A^+_{\mathrm{F}} \multimap_{\mathrm{F}} B^-_{\mathrm{F}}]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} ([\![A^+_{\mathrm{F}}]\!] \multimap_{\mathrm{L}} [\![B^-_{\mathrm{F}}]\!])
\end{aligned}$$

$$[\![\oplus_{\mathrm{F}}\{L_i : (A^+_{\mathrm{F}})_i\}_I]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} {\downarrow}^{\mathrm{L}}_{\mathrm{L}}(\oplus_{\mathrm{L}}\{L_i : [\![(A^+_{\mathrm{F}})_i]\!]\}_I)$$

$$[\![\&_{\mathrm{F}}\{L_i : (A^-_{\mathrm{F}})_i\}_I]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} (\&_{\mathrm{L}}\{L_i : [\![(A^-_{\mathrm{F}})_i]\!]\}_I)$$

$$[\![{\downarrow}^{\mathrm{U}}_{\mathrm{F}} A^+_{\mathrm{U}}]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} {\downarrow}^{\mathrm{U}}_{\mathrm{L}} [\![A^+_{\mathrm{U}}]\!] \qquad [\![{\uparrow}^{\mathrm{F}}_{\mathrm{L}} A^-_{\mathrm{L}}]\!] = 1_{\mathrm{L}} \&_{\mathrm{L}} {\downarrow}^{\mathrm{L}}_{\mathrm{L}} {\uparrow}^{\mathrm{L}}_{\mathrm{L}} [\![A^-_{\mathrm{L}}]\!]$$

Additionally, the transformation leaves linear and persistent types untouched other than to transform any affine subexpressions they may contain. This transformation eliminates all occurrences of affine types at the cost of forcing us to constantly answer the question "Should this channel be discarded now?" In addition to the extra instructions needed to answer this question (which could be implicit), this transformation has one unavoidable misfeature: it can force communication to nearly be synchronous. Consider what happens to a positive type. After the transformation, it would have a queue size bound of 1, exactly the same constraint we enforced when forcing channels to be synchronous at the type level (Definition 9). One way to sidestep this problem is to notice that we do not need precise garbage collection, an affine process executing for a few more instructions, while unfortunate, is unlikely to be dangerous. Thus, we could implement these garbage collection messages via special asynchronous interrupts and leave our main channel untouched.

Allowing some delays in garbage collection also suggests that we could insert external choices only on negative types. This bounds our garbage collection imprecision to the next time the process to be collected provides a negative type (i.e., either immediately or after sending a shift), but means that we do not artificially bound our queue sizes. To maintain garbage collecting completeness in the face of unbounded positive types, we need either some backup garbage collection strategy or to require boundedness for all affine types. Regardless of the choice on where to insert garbage collection messages, this will potentially double message traffic, suggesting that a bundled messaging semantics (section 4.2) may be worth using here.

4.6.7 Related Work

Linear Logic with Subexponentials [65] is similar to the Adjoint Logic approach to presenting substructural logics. For our purposes, it could have reasonably formed a basis for implementing SILL, though it is unclear how it would interact with polarization. While they examine computation enabled by the logic, it is done by proof search and not with a Curry-Howard style connection. Additionally, mediation through subexponentials does not produce the nice property that all connectives are either a send or receive, rather they look closer to the combined rule for $!A$.

Another recent attempt at integrating affine style weakening into session types is that of Affine Sessions [61]. They lack a strategy for implementation or the ability to specify linear types for sessions. The first is briefly mentioned as future work, and the latter is likely easy to rectify. Since they take a more classical view of Session Types, they naturally focus on channels and not processes as the fundamental unit of garbage collection. Perhaps the most important result of this difference is that processes can react to the channel being released by the other process using a particular channel through a sort of error handling mechanism.

In the broader field of using affine types to represent garbage collection concerns SILL follows in a rich tradition of languages [25, 54, 88] that utilize substructural logics to perform automatic garbage collection. These languages all worry about precisely tracking the usage of resources, not just channels, and commonly settle on affine types as the correct substructural type for doing so.

4.7 Subtyping

Choice, both internal and external, as we have seen it has a few unpleasant limitations. For example, it is only binary forcing us to represent larger choices via a tree of choices and an associated increase in message traffic. Since both choice operators are associative and commutative, this tree structure is unimportant, and we should use a variable width choice. To do this we fix a set of labels L_i and add two new propositistions that map some subset of these labels (specified by an indexing set I) to types of the appropriate modality and polarity:

$$\begin{array}{lcll} L & ::= & \ldots & \text{(Labels)} \\ A_m^+ & ::= & \ldots & \\ & | & \oplus_m\{L_i : (A_m^+)_i\}_I & \text{(Send Choice)} \\ A_m^- & ::= & \ldots & \\ & | & \&_m\{L_i : (A_m^-)_i\}_I & \text{(Receive Choice)} \end{array}$$

Given the risk of confusing index and modalitity subscripts we will always show both, even when they should be discoverable from context. When the exact indexing set is clear from context, e.g., when all cases are explity shown, we will omit it.

Once we have rephrased choice in this manner we can add session subtyping in the style of Gay and Hole [35]. Following their lead we first define a coinductive subtyping relationship.

Definition 31 (Coninductive Subtyping Relation). *A relation on session types, $\sim$, is a coninductive subtyping relation if for $A \sim B$:*

- *If $A = 1_m$, then $B = 1_m$*
- *If $A = \downarrow_k^m C_m^-$, then $B = \downarrow_k^m D_m^-$ and $C_m^- \sim D_m^-$*
- *If $A = \uparrow_k^m C_k^+$, then $B = \uparrow_k^m D_k^+$ and $C_k^+ \sim D_k^+$*
- *If $A = \tau \wedge_m C_m^+$, then $B = \tau \wedge_m D_m^+$ and $C_m^+ \sim D_m^+$*
- *If $A = \tau \supset_m C_m^-$, then $B = \tau \supset_m D_m^-$ and $C_m^- \sim D_m^-$*
- *If $A = C_m^+ \otimes_m \hat{C}_m^+$, then $B = D_m^+ \otimes_m \hat{D}_m^+$ and $C_m^+ \sim D_m^+$ and $\hat{C}_m^+ \sim \hat{D}_m^+$*
- *If $A = C_m^+ \multimap_m \hat{C}_m^-$, then $B = C_m^+ \multimap_m \hat{D}_m^-$ and $D_m^+ \sim C_m^+$ and $\hat{C}_m^- \sim \hat{D}_m^-$*
- *If $A = \oplus_m\{L_i : (C_m^+)_i\}_I$, then $B = \oplus_m\{L_j : (D_m^+)_j\}_J$ and $I \subseteq J$ and for $i \in I$, $(C_m^+)_i \sim (D_m^+)_i$*
- *If $A = \&_m\{L_i : (C_m^-)_i\}_I$, then $B = \&_m\{L_j : (D_m^-)_j\}_J$ and $J \subseteq I$ and for $j \in J$, $(C_m^-)_j \sim (D_m^-)_j$*

For two types, A and B, if there exists a coniductive subtyping relation $\sim$ such that $A \sim B$ then we call A a subtype *of B, denoted $A \sqsubseteq B$. For two lists of propositions, $\vec{A}$ and $\vec{B}$, we write $\vec{A} \sqsubseteq \vec{B}$ if they are subtypes pointwise.*

While coinductive subtyping subtyping gives us a fair amount of flexibility, it does not allow anything too crazy.

Lemma 32. *If $A \sqsubseteq B$, then A and B have the same polarity and modality.*

Proof. By cases. □

Subtyping forms a preorder. It would form a partial order if we had the appropriate coinductive notion of equality. Since we never need anti-symmetry, we will omit that proof.

Lemma 33. *$\sqsubseteq$ is reflexive.*

Proof. The identity relationship is a coninductive subtyping relationship and reflexive. □

Lemma 34. *$\sqsubseteq$ is transitive.*

Proof. Let $A \sqsubseteq B$ and $B \sqsubseteq C$. These have witnessing two witnessing relations $\sim_1$ and $\sim_2$. The following relation witnesses $A \sqsubseteq C$:

$$\begin{aligned} &\{(A_1, A_2) | \text{ for some } D\text{: } A_1 \sim_1 D \text{ and } D \sim_2 A_2\} \\ \cup&\{(A_2, A_1) | \text{ for some } D\text{: } A_2 \sim_2 D \text{ and } D \sim_1 A_1\} \end{aligned}$$

□

With our subtyping relation we only need to alter a few judgmental rules:

$$\frac{\Gamma \geq \textsc{f} \quad A \sqsubseteq B}{\Delta; \Psi; \Gamma, A \vdash B}\ \textsc{Id} \qquad \frac{\vec{A} \geq m \geq r \quad \Delta; \Psi \vdash \{C_m \leftarrow \vec{D}\} \quad \vec{A} \sqsubseteq \vec{D} \quad \Delta; \Psi; \Gamma, C_m \vdash B_r}{\Delta; \Psi; \vec{A}, \Gamma \vdash B_r}\ \{\}E$$

$$\frac{k \in I \quad \Delta; \Psi; \Gamma \vdash (A^+_m)_k}{\Delta; \Psi; \Gamma \vdash \oplus_m\{L_i : (A^+_m)_i\}_I}\ \oplus R_{L_k} \qquad \frac{\text{for all } i \in I\text{: } \Delta; \Psi; \Gamma, (A^+_m)_i \vdash B}{\Delta; \Psi; \Gamma, \oplus_m\{L_i : (A^+_m)_i\}_I \vdash B}\ \oplus L$$

$$\frac{\text{for all } i \in I\text{: } \Delta; \Psi; \Gamma \vdash (A^-_m)_i}{\Delta; \Psi; \Gamma \vdash \&_m\{L_i : (A^-_m)_i\}_I}\ \& R \qquad \frac{k \in I \quad \Delta; \Psi; \Gamma, (A^-_m)_k \vdash B}{\Delta; \Psi; \Gamma, \&_m\{L_i : (A^-_m)_i\}_I \vdash B}\ \& L_{L_k}$$

All judgments $\Delta; \Psi; \Gamma \vdash A_m$ presuppose $\Gamma \geq m$.

Since we cannot implicitly denote subtyping constraints, as we do with equality via pattern matching, the two rules that perform non-local checks on types (Id and $\{\}E$) need to be updated to use subtyping in their checks. Notice that Lemma 33 ensures that this is a generalization of their prior behavior.

In the following we let $\Gamma \sqsubseteq \Gamma'$ denote the pointwise lifting of subtyping to multisets of assumptions. This lets us show that the informal notion of a subtype as something that can be used wherever its supertype is acceptable is actually true.

Lemma 35. *If* $\Delta; \Psi; \Gamma \vdash A$ *and* $A \sqsubseteq B$ *and* $\Gamma' \sqsubseteq \Gamma$*, then* $\Delta; \Psi; \Gamma' \vdash B$*.*

Proof. By induction on $\Delta; \Psi; \Gamma \vdash A$ and casing on the last rule used. Since most cases are similar we will show only the relatively interesting ones.

Case Id: We are given:

$$\frac{\Gamma \geq \textsc{f} \quad A_m \sqsubseteq B_m}{\Delta; \Psi; \Gamma, A_m \vdash B_m}\ \textsc{Id}$$

and show for $C_m \sqsubseteq A_m$, $\Gamma' \sqsubseteq \Gamma$ and $B_m \sqsubseteq D_m$:

$$\frac{\Gamma' \geq \textsc{f} \quad C_m \sqsubseteq D_m}{\Delta; \Psi; \Gamma', C_m \vdash D_m}\ \textsc{Id}$$

which holds from Lemma 34.

Case $\{\}E$: This is similar to the case for Id but matching lists of subtypes rather than a single subtype.

Case $\multimap R$: We are given:

$$\frac{\Delta; \Psi; \Gamma, A_m^+ \vdash B_m^-}{\Delta; \Psi; \Gamma \vdash A_m^+ \multimap_m B_m^-} \multimap R$$

and show for $\Gamma' \sqsubseteq \Gamma$, $C_m^+ \sqsubseteq A_m^+$ and $B_m^- \sqsubseteq D_m^-$:

$$\frac{\Delta; \Psi; \Gamma', C_m^+ \vdash D_m^-}{\Delta; \Psi; \Gamma' \vdash C_m^+ \multimap_m D_m^-} \multimap R$$

which holds by the inductive hypothesis.

Case $\multimap L$: We are given:

$$\frac{\Gamma_1 \geq m \quad \Delta; \Psi; \Gamma_1 \vdash A_m^+ \quad \Delta; \Psi; \Gamma_2, B_m^- \vdash C_r}{\Delta; \Psi; \Gamma_1, \Gamma_2, A_m^+ \multimap_m B_m^- \vdash C_r} \multimap L$$

and need to show for $\Gamma_1' \sqsubseteq \Gamma_1$, $\Gamma_2' \sqsubseteq \Gamma_2$, $\hat{B}_m^- \sqsubseteq B_m^-$, $A_m^+ \sqsubseteq \hat{A}_m^+$, and $C_r \sqsubseteq \hat{C}_r$:

$$\frac{\Gamma_1' \geq m \quad \Delta; \Psi; \Gamma_1' \vdash \hat{A}_m^+ \quad \Delta; \Psi; \Gamma_2', \hat{B}_m^- \vdash \hat{C}_r}{\Delta; \Psi; \Gamma_1', \Gamma_2', \hat{A}_m^+ \multimap_m \hat{B}_m^- \vdash \hat{C}_r} \multimap L$$

which holds after two uses of the inductive hypothesis and the pointwise use of Lemma 32.

Case $\oplus R_{L_k}$: We are given:

$$\frac{k \in I \quad \Delta; \Psi; \Gamma \vdash (A_m^+)_k}{\Delta; \Psi; \Gamma \vdash \oplus_m\{L_i : (A_m^+)_i\}_I} \oplus R_{L_k}$$

and show for $\Gamma' \sqsubseteq \Gamma$ and $\oplus_m\{L_j : (B_m^+)_j\}_J$, where $I \subseteq J$ and for all $j \in I$ we have $(A_m^+)_j \sqsubseteq (B_m^+)_j$:

$$\frac{k \in J \quad \Delta; \Psi; \Gamma' \vdash (B_m^+)_k}{\Delta; \Psi; \Gamma \vdash \oplus\{L_j : (B_m^+)_j\}_J} \oplus R_{L_k}$$

which holds by inductive hypothesis and $I \subseteq J$.

Case $\oplus L$: We are given:

$$\frac{\text{for all } i \in I\text{: } \Delta; \Psi; (A_m^+)_i \vdash C_r}{\Delta; \Psi; \Gamma, \oplus_m\{L_i : (A_m^+)_i\}_I \vdash C_r} \oplus L$$

and we show for $\Gamma' \sqsubseteq \Gamma$, $C \sqsubseteq D$, and $\oplus_m\{L_j : (B_m^+)_j\}_J$, where $J \subseteq I$ and for all $j \in J$ we have $(B_m^+)_j \sqsubseteq (A_m^+)_j$:

$$\frac{\text{for all } j \in J\text{: } \Delta; \Psi; \Gamma, (B_m^+)_j \vdash C_r}{\Delta; \Psi; \Gamma, \oplus_m\{L_j : (B_m^+)_j\}_J \vdash C_r} \oplus L$$

which holds by repeated uses of the inductive hypothesis.

□

This large result can be simplified into a pair of rules that are easier to apply.

Corollary 36. *The following rules are admissible:*

$$\frac{B \sqsubseteq A \quad \Delta;\Psi;\Gamma \vdash B}{\Delta;\Psi;\Gamma \vdash A}\ \textsc{Sub}_R \qquad \frac{A \sqsubseteq C \quad \Delta;\Psi;\Gamma, C \vdash B}{\Delta;\Psi;\Gamma, A \vdash B}\ \textsc{Sub}_L$$

Since we have new logical rules we need to introduce new syntax. Forwarding and binding new processes both can reuse existing syntax, but the two choice constructs need to be expanded to allow for more labels than just inl and inr. As with the propositions we will add an (omittable) index set to the case contrusct.

$$P ::= \ldots \mid \mathsf{send}\ c\ L \mid \mathsf{case}\ c\ \mathsf{of}\ \{L_j \to P_j\}_J$$

As typing rules:

$$\frac{A_m \sqsubseteq C_m}{\Delta;\Psi; a : A_m \vdash c \leftarrow a :: c : C_m}\ \textsc{Id}$$

$$\frac{\vec{A} \geq m \quad \Delta;\Psi \vdash e : \{B_m \leftarrow \vec{D}\} \quad \vec{A} \sqsubseteq \vec{D} \quad \Delta;\Psi;\Gamma, b : B_m \vdash P :: c : C_r}{\Delta;\Psi;\Gamma, \vec{a} : \vec{A} \vdash b \leftarrow e \multimap\!\!< \vec{a}; P :: c : C_r}\ \{\}E$$

$$\frac{k \in I \quad \Delta;\Psi;\Gamma \vdash P :: c : (A_m^+)_k}{\Delta;\Psi;\Gamma \vdash \mathsf{send}\ c\ L_k; P :: c : \oplus_m\{L_i : (A_m^+)_i\}_I}\ \oplus R_{L_k}$$

$$\frac{I \subseteq J \quad \text{for all } k \in I : \ \Delta;\Psi;\Gamma, a : (A_m^+)_k \vdash P_k :: c : C_r}{\Delta;\Psi;\Gamma, a : \oplus_m\{L_i : (A_m^+)_i\}_I \vdash \begin{pmatrix}\mathsf{case}\ a\ \mathsf{of} \\ \{L_j \to P_j\}_J\end{pmatrix} :: c : C_r}\ \oplus L$$

$$\frac{I \subseteq J \quad \text{for all } k \in I : \ \Delta;\Psi;\Gamma \vdash P_k :: c : (C_m^-)_k}{\Delta;\Psi;\Gamma \vdash \begin{pmatrix}\mathsf{case}\ a\ \mathsf{of} \\ \{L_j \to P_j\}_J\end{pmatrix} :: c : \&_m\{c_i : (C_m^-)_i\}_I}\ \& R$$

$$\frac{k \in I \quad \Delta;\Psi;\Gamma, a : (A_m^-)_k \vdash P :: c : C_r}{\Delta;\Psi;\Gamma, a : \&\{L_i : (A_m^-)_i\}_I \vdash \mathsf{send}\ c\ L_k; P :: c : C_r}\ \& L_{L_k}$$

Operationally, we replace the inl and inr messages with:

$$M ::= \ldots \mid L$$

and rules to send and receive them:

$$\textsc{Send}_{\mathsf{choice}} : \mathsf{exec}_c(\mathsf{send}\ b\ L_k; P) \otimes \mathsf{que}(a, M, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M\ L_k, b)$$

$$\textsc{Recv}_{\mathsf{choice}} : !(k \in J) \otimes \mathsf{exec}_c\begin{pmatrix}\mathsf{case}\ a\ \mathsf{of} \\ \{L_j \to P_j\}_J\end{pmatrix} \otimes \mathsf{que}(a, L_k\ M, b)$$
$$\multimap \mathsf{exec}_c(P_k) \otimes \mathsf{que}(a, M, b)$$

4.7.1 Example: Permissions

Subtyping gives us an ability to present processes at multiple types for different users. Statically assigned permissions similarly require presenting resources differently to different users. For example, if we have a simple centralized automated grading system, each user, instead of being able to access grades or submit work for all users, should only be able to interact with his or her own work. To start with, consider only the case where students can retrieve previously assigned grades. We can type the grading database with the following type:

$$\&\{\texttt{AliceHW1} : \texttt{Int} \wedge 1; \texttt{AliceHW2} : \texttt{Int} \wedge 1; \dots; \texttt{BobHW1} : \texttt{Int} \wedge 1; \dots\}$$

where each label corresponds requesting to see the grade for a given user's particular homework assignment. If we had a module system we could give each user only a personalized subtype. For example, Alice should only be able to access the database at its supertype:

$$\&\{\texttt{AliceHW1} : \texttt{Int} \wedge 1; \texttt{AliceHW2} : \texttt{Int} \wedge 1; \dots\}$$

and similarly for Bob. On the other hand, course administrators would have access to the database at its full type (assuming there is only one course).

To allow each user to access the database independently we can allow each user to have its own persistent copy of the database. For example, the type signature for Alice's process might look like this:

$$\{1 \leftarrow {\downarrow}^{\mathrm{U}}_{\mathrm{L}}{\uparrow}^{\mathrm{U}}_{\mathrm{L}}{\downarrow}^{\mathrm{L}}_{\mathrm{L}}\&\{\texttt{AliceHW1} : \texttt{Int} \wedge 1; \texttt{AliceHW2} : \texttt{Int} \wedge 1; \dots\}$$

allowing her to query her grades as much, or as little, as she wished.

Trying to extend this to allowing for new home work submissions is much harder. In a traditional session typed language (section 2.2 and chapter 3) we could use accept/request to initiate connections between user processes and a central database process. Unfortunately, SILL does not have these constructs[1] (and has simpler progress theorems as a result), so we will need to explicitly model this matching construct ourselves. For example, we can write a general purpose server type as one that takes some sort of client and, after interacting with it, returns a server (with updated state) and the client. The linear version of a server and client would have the following types:

$$\begin{aligned}\texttt{Client}\ \alpha^+_{\mathrm{L}} &= \oplus\{\texttt{done} : 1; \texttt{server} : \alpha^+_{\mathrm{L}}\}\\ \texttt{Server}_{\mathrm{F}}\ \alpha^+_{\mathrm{L}} &= {\uparrow}^{\mathrm{F}}_{\mathrm{L}}{\downarrow}^{\mathrm{L}}_{\mathrm{L}}(\alpha^+_{\mathrm{L}} \multimap {\uparrow}^{\mathrm{L}}_{\mathrm{L}}((\texttt{Client}\ \alpha^+_{\mathrm{L}}) \otimes {\downarrow}^{\mathrm{F}}_{\mathrm{L}}(\texttt{Server}_{\mathrm{F}}\ \alpha^+_{\mathrm{L}})))\end{aligned}$$

And then we write a process to match clients and servers (explicitly performing the

[1]Adding them to a logically based session typing language is a pressing open problem.

matching implicit in similar languages' $\mathsf{accept}/\mathsf{request}$), here shown as appropriate for a single client and server:

```
match : ∀ $\alpha^+_L$.{1 ← Client $\alpha^+_L$; Server$_F$ $\alpha^+_L$}
$g_L$ ← match ⤙ $a_L$ $b_F$ =
  case $a_L$ of
    done → wait $a_L$;
            close $g_L$
    server → shift $c_L$ ← send $b_F$;
              shift $c_L$ ← recv $c_L$;
              send $c_L$ $a_L$;
              shift $c_L$ ← send $c_L$;
              $d_L$ ← recv $c_L$;
              shift $f_F$ ← recv $c_L$;
              $g_L$ ← match ⤙ $d_L$ $f_F$
```

which we can expand to include multiple servers and clients, if needed. Similarly, we can expand the kinds of servers usable by clients, or provide special `match`-level functions, by expanding the choice in the definition of `Client`. Once we have fixed concrete servers types, subtyping can allow clients to be restricted to only interacting with subsets of them, or subtypes of the servers. There is some tension between the most general form of this example and our integration of polymorphism and subtyping. Specifically, without bounded polymorphism, we cannot write a version of `match` that takes a variety of clients that are subtypes of an unspecified collection of servers. In practice, this does not seem to be too burdensome, we need to write a custom match for any set of server types anyway, to handle routing between the client choices and the appropriate server, and any concrete `match` has no need for bounded polymorphism.

To see this in action, lets explore what `match` for a variant of the automatic grader would look like for just Alice, Bob, and an instructor. For simplicity, we assume there is only one assignment. First, the types:

$$\begin{aligned}
\mathtt{All} &= \oplus\left\{\begin{matrix}\mathsf{done} : 1;\ \mathtt{submitA} : \mathtt{Int} \wedge \mathtt{All};\ \mathtt{submitB} : \mathtt{Int} \wedge \mathtt{All}; \\ \mathtt{grade} : {\downarrow}^{L}_{L}(\mathtt{Int} \supset \mathtt{Int} \supset {\uparrow}^{L}_{L}\mathtt{All})\end{matrix}\right\} \\
\mathtt{Alice} &= \oplus\{\mathtt{done} : 1;\ \mathtt{submitA} : \mathtt{Int} \wedge \mathtt{Alice}\} \\
\mathtt{Bob} &= \oplus\{\mathtt{done} : 1;\ \mathtt{submitB} : \mathtt{Int} \wedge \mathtt{Bob}\} \\
\mathtt{Elsa} &= \oplus\{\mathtt{done} : 1;\ \mathtt{grade} : {\downarrow}^{L}_{L}(\mathtt{Int} \supset \mathtt{Int} \supset {\uparrow}^{L}_{L}\mathtt{Elsa})\} \\
\mathtt{Server} &= {\uparrow}^{F}_{L}{\downarrow}^{L}_{L}\&\left\{\begin{matrix}\mathtt{submitA} : (\mathtt{Int} \wedge \mathtt{Alice}) \multimap {\uparrow}^{L}_{L}(\mathtt{Alice} \otimes {\downarrow}^{F}_{L}\mathtt{Server}); \\ \mathtt{submitB} : (\mathtt{Int} \wedge \mathtt{Bob}) \multimap {\uparrow}^{L}_{L}(\mathtt{Bob} \otimes {\downarrow}^{F}_{L}\mathtt{Server}); \\ \mathtt{grade} : ({\downarrow}^{L}_{L}(\mathtt{Int} \supset \mathtt{Int} \supset {\uparrow}^{L}_{L}\mathtt{Elsa})) \multimap {\uparrow}^{L}_{L}(\mathtt{Elsa} \otimes {\downarrow}^{F}_{L}\mathtt{Server})\end{matrix}\right\}
\end{aligned}$$

Notice that `Alice`, `Bob`, and `Elsa` are all subtypes of `Both`. So we can implement `matchABE` relatively easily:

```
matchABE : {1 ← All; All; All; Server}
$c_L$ ← matchABE —≺ $a_L$ $b_L$ $e_L$ $s_F$ =
 case $a_L$ of
   done → wait $a_L$; $c_L$ ← matchBE —≺ $b_L$ $e_L$ $s_F$
   submitA → shift $g_L$ ← send $s_F$;
             shift $g_L$ ← recv $g_L$;
             send $g_L$ submitA;
             send $g_L$ $a_L$;
             shift $g_L$ ← send $g_L$;
             $a_L$ ← recv $g_L$;
             shift $s_F$ ← recv $g_L$;
             $c_L$ ← matchABE —≺ $b_L$ $e_L$ $a_L$ $s_F$
   submitB → ...
   grade → ...
```

where `matchBE` is a matching process for only Bob and the instructor and we omit the other branches due to similarity. We rotate the client channels between recursive calls to provide some semblance of fairness.

While this example accomplishes its nominal objectives, it demonstrates a number of weaknesses in using SILL's subtyping to accomplish basic security objectives. First, a more general language concern, is that this is fairly verbose. This suggests that we should either integrate a higher level language for expressing permissions (e.g., an authorization logic [34,84]) or some sort of template or macro language. Second, while our servers are required to hand back a client, there is no guarantee that the client returned is the one `matchABE` sent. A well-typed program cannot discard any clients, so this risk might be ignorable depending how willing we are to trust the server. Third, SILL has no *productivity* theorem so nearly every step of `matchABE` can fail if the process' counterparty does not interact with it, either intentionally or through unfair scheduling. Extending the language with select (section 4.3) and using it heavily can avoid this problem at the cost of even more code. Lastly, our permissions are too static. We have no means to either revoke permissions for students or add new students. Both can be fixed if we have a finite number of potential students, but, again, at an extreme cost of code size. This suggests we need a more dependent notion of choice, allowing us to dynamically generate branches based on permissions (e.g., by performing a lookup in some permission database).

4.7.2 Theorems

The main subtyping result from the logic remains in the type system.

Corollary 37. *If* $\Delta; \Psi; \Gamma \vdash P :: a : A$ *and* $A \sqsubseteq B$ *and* $\Gamma' \sqsubseteq \Gamma$*, then* $\Delta; \Psi; \Gamma' \vdash P :: a : B$.

Proof. This is the Curry-Howard version of Lemma 35. □

A new message type means we need to augment our notion of a well-typed queue:

$$\frac{k \in I \quad \Gamma \vdash \mathsf{que}(a, M, b) :: (A_m^+)_k \leftsquigarrow B}{\Gamma \vdash \mathsf{que}(a, L_k\ M, b) :: \oplus_m\{L_i : (A_m^+)_i\}_I \leftsquigarrow B} \oplus_{\mathsf{q}}$$

$$\frac{k \in I \quad \Gamma \vdash \mathsf{que}(a, M, b) :: (A_m^-)_k \leftsquigarrow B}{\Gamma \vdash \mathsf{que}(a, L_k\ M, b) \&_m\{L_i : (A_m^-)_i\}_I \leftsquigarrow B} \&_{\mathsf{q}}$$

However, we do not need to change our notion of a well-typed configuration.

Theorem 38 (Preservation). *If $\Gamma \vdash E$ and $E \to E'$, then $\Gamma \vdash E'$.*

Proof. Most cases can follow the proof of Lemma 28 with uses of Lemma 35 to adjust types. We will confirm that the new rules works.

Case Send$_{\mathsf{choice}}$: There are two cases, when the process is typed with $\oplus R_{L_k}$ and one for $\& L_{L_k}$. We show only the first due to similarity. We are given:

$$\mathcal{E} = \Gamma, \Gamma', \Gamma'' \vdash E$$

$$\mathcal{M} = \Gamma'' \vdash M : A_m^+ \leftsquigarrow \oplus_m \{L_i : (C_m^+)_i\}_I$$

$$\frac{\dfrac{\overbrace{k \in I}^{\mathcal{K}} \quad \overbrace{\emptyset; \emptyset; \Gamma' \vdash P :: c : (C_m^+)_{L_k}}^{\mathcal{P}}}{\emptyset; \emptyset; \Gamma' \vdash \mathsf{send}\ c\ L_k; P :: c : \oplus\{L_i : (C_m^+)_i\}_I} \oplus R_{L_k} \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, a : A^+ \vdash \mathsf{exec}_c(\mathsf{send}\ c\ L_k; P), \mathsf{que}(a, M, c), E} \mathsf{WF}_{\to}$$

and replace this with:

$$\frac{\mathcal{P} \quad \dfrac{\mathcal{M} \quad \dfrac{\mathcal{K} \quad \dfrac{}{\emptyset \vdash \cdot : C_k^+ \leftsquigarrow (C_m^+)_k} \emptyset_{\mathsf{q}}}{\emptyset \vdash L_k : \oplus_m\{L_i : (C_m^+)_i\}_I \leftsquigarrow (C_m^+)_k} \oplus_{\mathsf{q}}}{\Gamma'' \vdash M\ L_k : (A_m^+) \leftsquigarrow (C_m^+)_k} \mathsf{trans}^+ \quad \mathcal{E}}{\Gamma, a : A^+ \vdash \mathsf{exec}_c(P), \mathsf{que}(a, M\ L_k, c), E} \mathsf{WF}_{\to}$$

Case Recv$_{\mathsf{choice}}$: As normal, there are two cases, one for $\& R$ and $\oplus L$. We only show the first. We are given:

$$\mathcal{E} = \Gamma, \Gamma', \Gamma' \vdash E$$

$$\mathcal{Z} = \dfrac{k \in I \quad \overbrace{\Gamma'' \vdash M : (C_m^-)_k \rightsquigarrow B}^{\mathcal{M}}}{\Gamma'' \vdash L_k\ M : \&\{L_i : (C_m^-)_i\}_I \rightsquigarrow B}\ \&_{\mathsf{q}}$$

$$\dfrac{\dfrac{I \subseteq J \quad \overbrace{\text{for all } i \in I\text{: } \emptyset; \emptyset; \Gamma' \vdash P_i :: c : (C_m^-)_i}^{\mathcal{P}}}{\emptyset; \emptyset; \Gamma' \vdash \begin{pmatrix}\mathsf{case}_J\ c\ \mathsf{of} \\ L_j \to P_j\end{pmatrix} :: c : \&\{L_i : (C_m^-)_i\}_I}\ \&R \qquad \mathcal{Z} \quad \mathcal{E}}{\Gamma, b : B \vdash \mathsf{exec}_c \begin{pmatrix}\mathsf{case}_J\ c\ \mathsf{of} \\ L_j \to P_j\end{pmatrix}, \mathsf{que}(c, L_k\ M, b), E}\ \mathsf{WF}_{\leftarrow}$$

which we replace with:

$$\dfrac{(\mathcal{P} \text{ for } i = k) \quad \mathcal{M} \quad \mathcal{E}}{\Gamma, b : B \vdash \mathsf{exec}_c(P_k), \mathsf{que}(c, M, b), E}\ \mathsf{WF}_{\leftarrow}$$

□

We no extra changes before proving progress.

Theorem 39 (Progress)**.** *If* $\Gamma \vdash E$ *and has a garbage free proof, then either* $E \to E'$ *or* E *is reactive.*

Proof. Follow the proof of Lemma 29. The slightly more complicated choice labels do not fundamentally change most cases, however two cases are worth discussing in a bit more detail. The expanded case construct allows a process to offer a choice over no cases, which can make no progress. There are two ways for a process to use this construct while being well typed, as a use of $\&R$ or as a use of $\oplus L$. In the first case, the channel provided by the process must be empty to be well-typed, so the process and its channel are reactive. In the $\oplus L$ case, the inductive hypothesis gives us that the channel used by the process is reactive and thus non-empty, a contradiction with it being well-typed. □

Related Work

Perhaps the closest work on integrating subtyping for session types and a functional language comes from Gay and Vasconcelos [36]. They investigate both synchronous channels, implementable in but not directly present in SILL, and asynchronous channels, which they feel presents a more realistic implementation of real world systems. They do not tightly associate each channel to a providing process and, as a result, both users of a channel give it a different (but compatible) type. SILL provides both affine and linear types whereas they only provide linear ones. The type system of SILL provides a strong stratification between functional level computation and side effecting statements, whereas their work intermingles the two. Due to this, they also assume more from their functional language and get richer subtyping system as a result. Finally, SILL focuses more on implementability.

4.8 Forwarding

While logically motivated, implementing forwarding is difficult. Toninho's semantics for SILL [90] implemented forwarding as a special metaoperation: a global unification of the two channel names involved. That is, its operational rule looked something like this:

$$\mathsf{exec}_c(c \leftarrow d) \multimap [c = d]$$

where $[c = d]$ performs global unification. This is problematic, the rule is non-local rule: in principal we need the entire execution context (omitted in the above rule) to be able to perform a forwarding operation. In a distributed system this clearly problematic to directly implement, but even in a shared memory system this approach can be unsatisfactory (imagine taking $O(n)$ time and a global lock to execute each forward, where n is the number of active processes).

The easiest and most general forwarding technique relies on residual processes that blindly passes messages between the forwarded channel and the channel provided by the forwarding process. This is particularly easy to implement when channels do not need special treatment of each type of message: we do not need to generate a type directed forwarding process. Additionally, this means that nothing other than the forwarding process itself needs to know how to perform forwarding. Instead we can work directly at the operational level by introducing a new operational proposition, $\mathsf{fwd}(c, d)$ which denotes that c and d are to be connected. with the following simple operational rules:

$$\begin{array}{ll}
\textsc{Fwd} : & \mathsf{exec}_c(c \leftarrow d) \multimap \mathsf{fwd}(c, d) \\
\textsc{Fwd}_{\rightarrow} : & \mathsf{que}(a, M, c) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(d, K\ M', b) \otimes !(K \notin \{\mathsf{end}, \mathsf{shift}\}) \\
& \quad \multimap \{\mathsf{que}(a, M\ K, c) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(d, M', b)\} \\
\textsc{Fwd}_{\leftarrow} : & \mathsf{que}(b, M, d) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(c, K\ M', a) \otimes !(K \notin \{\mathsf{end}, \mathsf{shift}\}) \\
& \quad \multimap \{\mathsf{que}(b, M\ K, d) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(c, M', a)\} \\
\textsc{Fwd}^{\mathsf{shift}}_{\rightarrow} : & \mathsf{que}(a, M, c) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(d, \mathsf{shift}, b) \\
& \quad \multimap \{\mathsf{que}(a, M\ \mathsf{shift}, c) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(b, \cdot, d)\} \\
\textsc{Fwd}^{\mathsf{shift}}_{\leftarrow} : & \mathsf{que}(b, M, d) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(c, \mathsf{shift}, a) \\
& \quad \multimap \{\mathsf{que}(b, M\ \mathsf{shift}, d) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(a, \cdot, c)\} \\
\textsc{Fwd}^{\mathsf{end}}_{\rightarrow} : & \mathsf{que}(a, M, c) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(d, \mathsf{end}, b) \multimap \mathsf{que}(a, M\ \mathsf{end}, c) \\
\textsc{Fwd}^{\mathsf{end}}_{\leftarrow} : & \mathsf{que}(b, M, d) \otimes \mathsf{fwd}(c, d) \otimes \mathsf{que}(c, \mathsf{end}, a) \multimap \mathsf{que}(b, M\ \mathsf{end}, d)
\end{array}$$

The main complication here is the need to treat shift messages carefully so that the ques always point in the correct direction. Since we have introduced a new construct into the operational semantics, we need a new rule for well-typed contexts before we can show that these rules are compatible with our progress and preservation results. We replace the prior well-typed rules for forwarding with the following:

$$\frac{\Gamma' \vdash D \leftsquigarrow A^{+} \quad \Gamma, \Gamma', d : D \vdash E}{\Gamma, a : A^{+} \vdash \mathsf{que}(a, M, c), \mathsf{fwd}(c, d), E}\ \mathsf{WF}^{\mathsf{fwd}}_{\rightarrow}$$

$$\frac{\Gamma' \vdash D^{-} \leftsquigarrow A \quad \Gamma, \Gamma', d : D^{-} \vdash E}{\Gamma, a : A \vdash \mathsf{que}(c, M, a), \mathsf{fwd}(c, d), E}\ \mathsf{WF}^{\mathsf{fwd}}_{\leftarrow}$$

Preservation is realtively straightforward.

Theorem 40 (Preservation)**.** *If* $\Gamma \vdash E$ *and* $E \to E'$*, then* $\Gamma \vdash E'$*.*

Proof. Most cases proceed as in Theorem 38. Let us then consider the new forwarding transitions.

Case Fwd: There are two cases, one for when the inital rule is $\mathsf{WF}_{\leftarrow}$ and one for $\mathsf{WF}_{\rightarrow}$. Starting with $\mathsf{WF}_{\leftarrow}$ we have:

$$\frac{\dfrac{}{\emptyset; d : C^{-} \vdash c \leftarrow d \vdash c :: C^{-}}\ \textsc{Id} \quad \overbrace{\Gamma' \vdash M : C^{-} \leftsquigarrow A}^{\mathcal{M}} \quad \overbrace{\Gamma, \Gamma', d : C^{-} \vdash E}^{\mathcal{E}}}{\Gamma, a : A \vdash \mathsf{que}(c, M, a), \mathsf{exec}_{c}(c \leftarrow d), E}\ \mathsf{WF}_{\leftarrow}$$

Which we replace with:

$$\frac{\mathcal{M} \quad \mathcal{E}}{\Gamma, a : A \vdash \mathsf{que}(c, M, a), \mathsf{fwd}(c, d), E}\ \mathsf{WF}^{\mathsf{fwd}}_{\leftarrow}$$

The version for $\mathsf{WF}_{\rightarrow}$ is similar.

Case Fwd$_{\rightarrow}$: There are two cases, one where the inner proof is $\mathsf{WF}_{\rightarrow}$ and one for $\mathsf{WF}^{\mathsf{fwd}}_{\rightarrow}$. We start with the first of these. Each possible kind of message for K generates its own case, we only show the for $\wedge_{\mathsf{q}}$ due to their similarity. We are given, after uses of Lemma 5:

$$\mathcal{Z} = \frac{\overbrace{\emptyset \vdash v : \tau}^{\mathcal{T}} \quad \overbrace{\Gamma''' \vdash M' : C^{+} \leftsquigarrow B}^{\mathcal{M}'}}{\Gamma''' \vdash v\ M' : \tau \wedge C^{+} \leftsquigarrow B}\ \wedge_{\mathsf{q}}$$

$$\frac{\overbrace{\Gamma' \vdash M : A^{+} \leftsquigarrow \tau \wedge C^{+}}^{\mathcal{M}} \quad \dfrac{\overbrace{\emptyset; \Gamma'' \vdash P :: b : B}^{\mathcal{P}} \quad \mathcal{Z} \quad \overbrace{\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash E}^{\mathcal{E}}}{\Gamma, \Gamma', d : \tau \wedge C^{+} \vdash \mathsf{que}(d, v\ M', b), \mathsf{exec}_{b}(P), E}\ \mathsf{WF}_{\rightarrow}}{\Gamma, a : A^{+} \vdash \mathsf{que}(a, M, c), \mathsf{fwd}(c, d), \mathsf{que}(d, v\ M', b), \mathsf{exec}_{b}(P), E}\ \mathsf{WF}^{\mathsf{fwd}}_{\rightarrow}$$

Which we replace with:

$$\mathcal{Z} = \dfrac{\mathcal{P} \quad \mathcal{M}' \quad \mathcal{E}}{\Gamma, \Gamma' d : C^+ \vdash \mathsf{que}(d, M', b), \mathsf{exec}_b(P), E}\ \mathsf{WF}_{\rightarrow}$$

$$\dfrac{\dfrac{\mathcal{M} \quad \dfrac{\mathcal{T} \quad \dfrac{}{\emptyset \vdash \cdot : C^+ \leftsquigarrow C^+}\ \emptyset_{\mathsf{q}}}{\Gamma' \vdash M : A^+ \leftsquigarrow \tau \wedge C^+}\ \wedge_{\mathsf{q}}}{\Gamma' \vdash M\ v : A^+ \leftsquigarrow C^+}\ \mathsf{trans}^+ \qquad \mathcal{Z}}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M\ v, c), \mathsf{fwd}(c, d), \mathsf{que}(d, M', b), \mathsf{exec}_b(P), E}\ \mathsf{WF}^{\mathsf{fwd}}_{\rightarrow}$$

The variant with $\mathsf{WF}^{\mathsf{fwd}}_{\rightarrow}$ is similar.

Case Fwd$_{\leftarrow}$: As the case for Fwd$_{\rightarrow}$ but reversing queues.

Case Fwd$^{\mathsf{shift}}_{\rightarrow}$: As the case for Fwd$_{\rightarrow}$ but reversing the queue that shift was read from.

Case Fwd$^{\mathsf{shift}}_{\leftarrow}$: As the case for Fwd$^{\mathsf{shift}}_{\rightarrow}$ but reversing the queues involved.

Case Fwd$^{\mathsf{end}}_{\rightarrow}$: As the case for Fwd$_{\rightarrow}$ but deleting the queue that end was read from.

Case Fwd$^{\mathsf{end}}_{\leftarrow}$: As the case for Fwd$^{\mathsf{end}}_{\rightarrow}$ but reversing the queues involved.

□

Before proving progress for this version, we need to update our notion of a reactive configuration to account for $\mathsf{fwd}(c, d)$. We treat these identically to a process providing c that receives when c is negative for the definition of reactive.

Theorem 41 (Progress)**.** *If $\Gamma \vdash E$, then either $E \to E'$ or E is reactive.*

Proof. As in Theorem 39. With the altered definition of reactive, the forwarding cases are straightforward. □

While this approach is easier to prove progress and preservation for than the version used in the rest of this thesis, it requires keeping the residual forwarding processes around until the combined channel is closed, potentially forever. In an early version of the OCaml implementation of SILL (section 5.5), this caused resource exhaustion on some examples. Additionally, every message takes an extra step for every forwarding join that it must cross, a potentially much larger cost than in the message based forwarding.

An alternate means of implementing this approach is to replace fwd with a specially crafted process that realizes this repeating version of forwarding. This corresponds, roughly, the a general result that reduces general Id to a version that works only on atomic propositions. Unfortunately, we cannot quite do this in a statically, due to our quantifiers. At run-time, as we saw in subsection 4.5.3, there are no unknown types whenever a process forwards, however statically we must deal with typing judgments like this: $\alpha^+_{\mathrm{L}}; \emptyset; d : \alpha^+_{\mathrm{L}} \vdash c \leftarrow d :: c : \alpha^+_{\mathrm{L}}$. If we were willing to keep run-time typing information around we could generate a forwarding process from the type of c (in $c \leftarrow d$) at run-time using the replacing the forwarding instruction with the following definition (showing only the linear cases and ignoring infinite types):

$$
\begin{aligned}
[\![1_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= \mathsf{wait}\ d; \mathsf{close}\ c \qquad [\![\tau \wedge A^+_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} = x \leftarrow \mathsf{recv}\ d; \mathsf{send}\ c\ x; [\![A^+_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![\tau \supset A^-_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= x \leftarrow \mathsf{recv}\ c; \mathsf{send}\ d\ x; [\![A^-_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![A^+_{\mathrm{L}} \otimes B^+_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= a \leftarrow \mathsf{recv}\ d; \mathsf{send}\ c\ a; [\![B^+_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![A^+_{\mathrm{L}} \multimap B^-_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= a \leftarrow \mathsf{recv}\ c; \mathsf{send}\ d\ a; [\![B^-_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![\oplus\{L_i : (A^+_{\mathrm{L}})_i\}_I]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= \mathsf{case}_I\ d\ \mathsf{of} \\
&\qquad\qquad L_i \to \mathsf{send}\ c\ L_i; [\![(A^+_{\mathrm{L}})_i]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![\&\{L_i : (A^-_{\mathrm{L}})_i\}_I]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= \mathsf{case}_I\ c\ \mathsf{of} \\
&\qquad\qquad L_i \to \mathsf{send}\ d\ L_i; [\![(A^-_{\mathrm{L}})_i]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![\exists \alpha. A^+_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= \alpha \leftarrow \mathsf{recv}\ d; \mathsf{send}\ c\ \alpha; [\![A^+_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![\forall \alpha. A^-_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= \alpha \leftarrow \mathsf{recv}\ c; \mathsf{send}\ d\ \alpha; [\![A^-_{\mathrm{L}}]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} \\
[\![\downarrow^m_{\mathrm{L}} A^-_m]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= \mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ d_{\mathrm{L}}; \mathsf{shift}\ b_m \leftarrow \mathsf{send} c_{\mathrm{L}}; [\![A^-_m]\!]^{a_m}_{b_m} \\
[\![\uparrow^{\mathrm{L}}_m A^+_m]\!]^{d_{\mathrm{L}}}_{c_{\mathrm{L}}} &= \mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ c_{\mathrm{L}}; \mathsf{shift}\ b_m \leftarrow \mathsf{send} d_{\mathrm{L}}; [\![A^+_m]\!]^{b_m}_{a_m}
\end{aligned}
$$

The second approach to doing forwarding is to immediately join the two channels involved. After removing the forwarding messages and their rules from our standard semantics, we add the following rules:

$$
\begin{aligned}
\textsc{Fwd}^+: &\quad \mathsf{que}(a_m, M, c_m) \otimes \mathsf{exec}_{c_m}(c_m \leftarrow d_m) \otimes \mathsf{que}(d_m, M', b_m) \\
&\quad \multimap \mathsf{que}(a_m, M\ M', b_m) \\
\textsc{Fwd}^-: &\quad \mathsf{que}(c_m, M, a_m) \otimes \mathsf{exec}_{c_m}(c_m \leftarrow d_m) \otimes \mathsf{que}(b_m, M', d_m) \\
&\quad \multimap \mathsf{que}(b_m, M'\ M, a_m)
\end{aligned}
$$

Theorem 42 (Preservation)**.** *If $\Gamma \vdash E$ and $E \to E'$ then $\Gamma \vdash E'$.*

Proof. Otherwise following the proof of Theorem 38 we show the two new cases.

Case $\textsc{Fwd}^+$: There are two cases, one each for when the secondary queue is used in $\mathsf{WF}_\to$ and one for $\mathsf{WF}_{\mathsf{end}}$. In the $\mathsf{WF}_\to$ case, we are given, after use of Lemma 5:

$$
\mathcal{Z} = \dfrac{\overbrace{\emptyset; \Gamma'' \vdash P :: b : B}^{\mathcal{P}} \quad \overbrace{\Gamma''' \vdash M' : C^+ \leftsquigarrow B}^{\mathcal{M}'} \quad \overbrace{\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash E}^{\mathcal{E}}}{\Gamma, \Gamma', d : C^+ \vdash \mathsf{que}(d, M', b), \mathsf{exec}_b(P), E}\ \mathsf{WF}_\to
$$

$$
\dfrac{\dfrac{}{\emptyset; d : C^+ \vdash c \leftarrow d :: c : C^+}\ \textsc{Id} \quad \overbrace{\Gamma' \vdash M : A^+ \leftsquigarrow C^+}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M, c), \mathsf{exec}_c(c \leftarrow d), \mathsf{que}(d, M', b), \mathsf{exec}_b(P), E}\ \mathsf{WF}_\to
$$

which we replace with:

$$\dfrac{\mathcal{P} \quad \dfrac{\mathcal{M} \quad \mathcal{M}'}{\Gamma', \Gamma''' \vdash M\ M' : A^+ \leftrightsquigarrow B}\,\mathsf{trans}^+ \quad \mathcal{E}}{\Gamma, a : A^+ \vdash \mathsf{que}(a, M\ M', b), \mathsf{exec}_b(P), E}\,\mathsf{WF}_\to$$

The case when the nested subproof uses $\mathsf{WF}_{\mathsf{end}}$ is similar.

Case $\textsc{Fwd}^-$: Since there is no type to represent "receive end" we only have on case here. We are given, after use of Lemma 5:

$$\mathcal{Z} = \dfrac{\overbrace{\emptyset; \Gamma'' \vdash P :: b : B^-}^{\mathcal{P}} \quad \overbrace{\Gamma''' \vdash M' : B^- \leftrightsquigarrow C^-}^{\mathcal{M}'} \quad \overbrace{\Gamma, \Gamma', \Gamma'', \Gamma''' \vdash E}^{\mathcal{E}}}{\Gamma, \Gamma', d : C^- \vdash \mathsf{que}(b, M', d), \mathsf{exec}_b(P), E}\,\mathsf{WF}_\leftarrow$$

$$\dfrac{\dfrac{}{\emptyset; d : C^- \vdash c \leftarrow d :: c : C^-}\,\textsc{Id} \quad \overbrace{\Gamma' \vdash M : C^- \leftrightsquigarrow A}^{\mathcal{M}} \quad \mathcal{Z}}{\Gamma, a : A \vdash \mathsf{que}(c, M, a), \mathsf{exec}_c(c \leftarrow d), \mathsf{que}(b, M', d), \mathsf{exec}_b(P), E}\,\mathsf{WF}_\leftarrow$$

which we replace with:

$$\dfrac{\mathcal{P} \quad \dfrac{\mathcal{M}' \quad \mathcal{M}}{\Gamma', \Gamma''' \vdash M'\ M : B^- \leftrightsquigarrow A}\,\mathsf{trans}^- \quad \mathcal{E}}{\Gamma, a : A \vdash \mathsf{que}(b, M'\ M, a), \mathsf{exec}_b(P), E}\,\mathsf{WF}_\leftarrow$$

□

Theorem 43 (Progress). *If $\Gamma \vdash E$, then either $E \to E'$ or E is reactive.*

Proof. We can mostly reuse the proof for Theorem 39, but we need to consider what happens with the new forwarding rules. There are two cases, depending on whether we use $\mathsf{WF}_\to$ or $\mathsf{WF}_\leftarrow$ on the forwarding process.

Case $\mathsf{WF}_\to$: There are two cases to consider, when the queue provided by the forwaring process is empty or non-empty. In the non-empty case this is reactive, so the inductive hypothesis finishes this case. Otherwise, the queue that we are forwarding from must be non-empty (it is positive and, by the inductive hypothesis, reactive) so we can transition.

Case $\mathsf{WF}_\leftarrow$: By the inductive hypothesis the queue to be forwarded to must be pointing in the correct direction to enable us to transition.

□

Unfortunately, in addition to being tricky to write a correct version of this operation, this approach, depending on how redirction can be performed, may have the major disadvantage of requiring the process performing the forward operation be able to directly alter the state of other processes. In a shared memory system, this might merely be expensive (e.g., if every channel use requires taking an extra lock), but for a distributed system this cost can be prohibitive.

Chapter 5

Implementation

In this chapter we cover a number of more practical considerations needed to actually implement a language such as SILL. The sections of this chapter can be broadly classified those concerned with the theoretical treatment of practical concerns and those that provide overviews of the three (partial) implementations of SILL itself. The rest of this chapter is structured as follows: first (section 5.1), a discussion of how to resolve some difficulties due to underspecification of the proofs that correspond to SILL's concrete syntax; second (section 5.2), we describe the type checking strategy employed by SILL's OCaml implementation; next (sections 5.3 and 5.4), we review some material needed to work with regular types in practice; lastly, we finish off with a trio of sections covering the three implementations of SILL.

5.1 Resource Management

Recall the type rules for $\otimes R$ and $\multimap L$:

$$\frac{\Delta; \Psi; \Gamma \vdash P :: a : A_m^+ \quad \Delta; \Psi; \Gamma' \vdash Q :: c : C_m^+}{\Delta; \Psi; \Gamma, \Gamma' \vdash \mathsf{send}\ c\ (a \leftarrow P); Q :: c : A_m^+ \otimes C_m^+} \otimes R$$

$$\frac{\Delta; \Psi; \Gamma \vdash P :: a : A_m^+ \quad \Delta; \Psi; \Gamma', b : B_m^- \vdash Q :: c : C}{\Delta; \Psi; \Gamma, \Gamma', b : A_m^+ \multimap B_m^- \vdash \mathsf{send}\ b\ (a \leftarrow P); Q :: c : C} \multimap L$$

Both of these rules require splitting the substructural environment into Γ and Γ' during type checking. Since we do not allow for duplicatation of resources, except for persistent resources, there is actually interesting information contained in how we choose to split the environment. There are a few ways we could fix this problem. Perhaps the easiest would be to notice that $\{\}E$ has no problems splitting its environment because the newly spawned process and we could fix our problems by having the syntax for $\otimes R$ and $\multimap L$ contain a list of channels. Alternatively, we could embrace the better behaved $\{\}E$ and change $\otimes R$ and

$\multimap L$ so send only existing channels:

$$\frac{\Delta; \Psi; \Gamma \vdash P :: c : C^+_m}{\Delta; \Psi; \Gamma, a : A^+_m \vdash \mathsf{send}\ c\ a; P :: c : A^+_m \otimes C^+_m} \otimes R$$

$$\frac{\Delta; \Psi; \Gamma, b : B^-_m \vdash P :: c : C}{\Delta; \Psi; \Gamma, a : A^+, b : A^+_m \multimap B^-_m \vdash \mathsf{send}\ b\ a; P :: c : C} \multimap L$$

Both of these choices are a bit undesirable: the first is much more verbose; the second makes $\otimes$ asymmetric.

Instead we explore an idea for coping with this problem developed in the context of proof search for linear logics: Resource Management [18, 97]. This family of approaches avoids the need to guess how to split environments by tracking, for a given subproof, which resources are actually used in the subproof and then confirming later that all resources are consumed correctly. Informally, this could be viewed as computing environment splits lazily. After implementing approaches based on both a naive system that might require backtracking and those from Cervesato *et al.* [18], we found the naive system to be sufficient for our needs. To demonstrate the difference, consider trying to prove the goal $A^+_m \otimes B^+_m$ from a collection of resources Γ. We might choose to try to find a proof starting from a use of $\otimes R$. This gives us two subgoals to prove A^+_m and B^+_m. In the naive system we independently try to prove both, discovering that we needed to use subsets, Γ_1 and Γ_2 of our initial resources to do so. However, due to linearity, we need to further ensure Γ_1 and Γ_2 are a partition of Γ. If they do not form a partition, we need to backtrack and find new proofs for the subgoals. This is, of course, undesirable when performing proof search. Instead RM1, the simplest system of Cervesato *et al.* [18], finds the used resources of one subproof and then allows the other subproof to be formed only from the resources left after performing the first subproof. This ensures that any proof found will not use too many resources. After implementing a type checking algorithm that tries to resolve enviromental splitting in the fasion of RM1 we learned two things: first, since SILL programs correspond to proofs there is much less to be gained from pruning environments between checking subproofs; second, providing high quality error messages under RM1 seemed to produce something equivalent to the naive system but with more complexity.

Another issue highlighted by the resource management is the notion of slack consumption of resources. As mentioned in the proof of Theorem 39, $\&\{\}_I$ and $\oplus\{\}_I$ are both perfectly legal types. The specialization of $\& R$ to this case is:

$$\frac{}{\Delta; \Psi; \Gamma \vdash \mathsf{case}_{\emptyset}\ c\ \mathsf{of} :: c : \&_m\{\}} \& R$$

When computing what subset of a given set of resources is needed to use this rule, a legal answer is anything between “none of them” or “all of them”. Instead of only considering a resource to be consumed or unconsumed, we add a third classification, *slackly consumed*, for resources denoting that they could be, but do not have to be, consumed by this proof. Confirming that all resources are used correctly, then becomes simple local consistency checks when otherwise type checking each instruction along with a final top-level check to confirm that all linear resources are either consumed or slackly consumed.

To make this formal we introduce an ordered set of consumption annotations to denote if a channel is guaranteed to be consumed in a proof ($\textsc{n}$), slackly consumed ($\textsc{s}$), or guaranteed to not be consumed ($\textsc{c}$).

$$\chi ::= \textsc{n} < \textsc{s} < \textsc{c}$$

And denote mappings from channels to these consumption annotations by Ω. Before we introduce the system that uses these annotations, we need two more auxilary notions. First we define, pointwise, a way to combine two consumption mappings over the same domain (n.b., the output depends on the channel modality):

$$\begin{array}{cc}
(a_{\textsc{l}} : \textsc{c}) \bowtie (a_{\textsc{l}} : \textsc{s}) = a_{\textsc{l}} : \textsc{c} & (a_{\textsc{l}} : \textsc{c}) \bowtie (a_{\textsc{l}} : \textsc{n}) = a_{\textsc{l}} : \textsc{c} \\
(a_{\textsc{l}} : \textsc{s}) \bowtie (a_{\textsc{l}} : \textsc{c}) = a_{\textsc{l}} : \textsc{c} & (a_{\textsc{l}} : \textsc{n}) \bowtie (a_{\textsc{l}} : \textsc{s}) = a_{\textsc{l}} : \textsc{c} \\
(a_{\textsc{l}} : \textsc{s}) \bowtie (a_{\textsc{l}} : \textsc{s}) = a_{\textsc{l}} : \textsc{s} & (a_{\textsc{l}} : \textsc{n}) \bowtie (a_{\textsc{l}} : \textsc{n}) = a_{\textsc{l}} : \textsc{n} \\
(a_{\textsc{f}} : \textsc{c}) \bowtie (a_{\textsc{f}} : \textsc{s}) = a_{\textsc{f}} : \textsc{c} & (a_{\textsc{f}} : \textsc{s}) \bowtie (a_{\textsc{f}} : \textsc{c}) = a_{\textsc{f}} : \textsc{c} \\
(a_{\textsc{f}} : \textsc{s}) \bowtie (a_{\textsc{f}} : \textsc{s}) = a_{\textsc{f}} : \textsc{s} & (a_{\textsc{u}} : \textsc{c}) \bowtie (a_{\textsc{u}} : \textsc{s}) = a_{\textsc{u}} : \textsc{c} \\
(a_{\textsc{u}} : \textsc{s}) \bowtie (a_{\textsc{u}} : \textsc{c}) = a_{\textsc{u}} : \textsc{c} & (a_{\textsc{u}} : \textsc{s}) \bowtie (a_{\textsc{u}} : \textsc{s}) = a_{\textsc{u}} : \textsc{s} \\
(a_{\textsc{u}} : \textsc{c}) \bowtie (a_{\textsc{u}} : \textsc{c}) = a_{\textsc{u}} : \textsc{c} &
\end{array}$$

and a notion of when one consumption could can be used in place of another:

$$\textsc{n} \sqsubseteq \textsc{n} \qquad \textsc{s} \sqsubseteq \textsc{n} \qquad \textsc{s} \sqsubseteq \textsc{s} \qquad \textsc{s} \sqsubseteq \textsc{c} \qquad \textsc{c} \sqsubseteq \textsc{c}$$

We then introduce a new judgment, $\Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m$, saying that using the type variables in Δ, type environment for the underlying language Ψ, and resources of Γ, the process P provides a channel c_m with type C_m. Additionally, the resources of Γ are consumed as specified in Ω. Consequentally, we presuppose that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Omega)$ and $\Gamma \geq m$. Lastly, we $\Omega \geq \chi$ to denote that all bindings are $\textsc{s}$ or $\textsc{c}$ and $b_m : \star$ to match when b_m maps to either $\textsc{s}$ or $\textsc{c}$. The system itself is shown in Figure 5.1 and Figure 5.1.

We can also show that this new system is appropriately related to the system of section 4.7.

Theorem 44 (Soundness)**.** *If $\Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m$ then for any $\Gamma' \subseteq \Gamma$, such that $\Omega(a_r) = \textsc{c}$ implies $\Gamma'(a_r)$ is defined and $\Omega(a_r) = \textsc{n}$ implies that $\Gamma'(a_r)$ is undefined, then $\Delta; \Psi; \Gamma' \vdash P :: c_m : C_m$.*

Proof. By induction on $\Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m$ and then by cases on the last

$$\dfrac{\Delta;\Psi;\vec{a}:\vec{A}\rhd\Omega\vdash P::a:A\quad \Omega\geq\mathrm{S}}{\Delta;\Psi\vdash a\leftarrow\{P\}\mathrel{-\!\!\prec}\vec{a}:A\leftarrow\vec{A}}\;\{\}I$$

$$\dfrac{\begin{array}{c}\vec{A}\geq m\geq r\quad \Delta;\Psi\vdash e:\{B_m\leftarrow\vec{D}\}\\ \vec{A}\sqsubseteq\vec{D}\quad \Delta;\Psi;\Gamma,b:B_m\rhd\Omega,b_m:\star\vdash P::c_r:C_r\end{array}}{\Delta;\Psi;\Gamma,\vec{a}:\vec{A}\rhd\Omega,\vec{a}:\vec{\mathrm{C}}\vdash b_m\leftarrow e\mathrel{-\!\!\prec}\vec{a};P::c_r:C_r}\;\{\}E$$

$$\dfrac{A_m\sqsubseteq C_m}{\Delta;\Psi;\Gamma,a_m:A_m\rhd a_m:\mathrm{C},(\mathrm{dom}(\Gamma|_{\mathrm{L}}):\vec{\mathrm{N}}),(\mathrm{dom}(\Gamma|_{\geq\mathrm{F}}):\vec{\mathrm{S}})\vdash c_m\leftarrow a_m::c_m:C_m}\;\textsc{Id}$$

$$\dfrac{}{\Delta;\Psi;\Gamma\rhd(\mathrm{dom}(\Gamma|_{\mathrm{L}}):\vec{\mathrm{N}}),(\mathrm{dom}(\Gamma|_{\geq\mathrm{F}}):\vec{\mathrm{S}})\vdash \textsf{close}\ c_m::c_m:1_m}\;1R$$

$$\dfrac{\Delta;\Psi;\Gamma\rhd\Omega\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma\rhd\Omega,a_k:\mathrm{C}\vdash\textsf{wait}\ a_k;P::c_m:C_m}\;1L$$

$$\dfrac{\Delta;\Psi\vdash e:\tau\quad \Delta;\Psi;\Gamma\rhd\Omega\vdash P::c_m:C_m^+}{\Delta;\Psi;\Gamma\rhd\Omega\vdash\textsf{send}\ c_m\ e;P::c_m:\tau\wedge C_m^+}\;\wedge R$$

$$\dfrac{\Delta;\Psi,x:\tau;\Gamma,a_k:A_k^+\rhd\Omega,a_k:\star\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma,a_k:\tau\wedge A_k^+\rhd\Omega,a_k:\mathrm{C}\vdash x\leftarrow\textsf{recv}\ a_k;P::c_m:C_m}\;\wedge L$$

$$\dfrac{\Delta;\Psi,x:\tau;\Gamma\rhd\Omega\vdash P::c_m:\tau\supset C_m^-}{\Delta;\Psi;\Gamma\rhd\Omega\vdash x\leftarrow\textsf{recv}\ c_m;P::c_m:\tau\supset C_m^-}\;\supset R$$

$$\dfrac{\Delta;\Psi\vdash e:\tau\quad \Delta;\Psi;\Gamma,b_k:B_k^-\rhd\Omega,b_k:\star\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma,b_k:\tau\supset B_k^-\rhd\Omega,b_k:\mathrm{C}\vdash\textsf{send}\ b_k\ e;P::c_m:C_m}\;\supset L$$

$$\dfrac{\Delta;\Psi;\Gamma\rhd\Omega_1\vdash P::a_m:A_m^+\quad \Delta;\Psi;\Gamma\rhd\Omega_2\vdash Q::c_m:C_m^+}{\Delta;\Psi;\Gamma\rhd\Omega_1\bowtie\Omega_2\vdash\textsf{send}\ c_m\ (a_m\leftarrow P);Q::c_m:A_m^+\otimes C_m^+}\;\otimes R$$

$$\dfrac{\Delta;\Psi;\Gamma,a_k:A_k^+,b_k:B_k^+\rhd\Omega,a_k:\star,b_k:\star\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma,b_k:A_k^+\otimes B_k^+\rhd\Omega,b_k:\mathrm{C}\vdash a_k\leftarrow\textsf{recv}\ b_k;P::c_m:C_m}\;\otimes L$$

$$\dfrac{k\in I\quad \Delta;\Psi;\Gamma\rhd\Omega\vdash P::c:(A_m^+)_k}{\Delta;\Psi;\Gamma\rhd\Omega\vdash\textsf{send}\ c\ L_k;P::c:\oplus_m\{L_i:(A_m^+)_i\}_I}\;\oplus R_{L_k}$$

$$\dfrac{I\subseteq J\quad \text{for all } k\in I:\ \Delta;\Psi;\Gamma,a_m:(A_m^+)_k\rhd\Omega_k,a_m:\star\vdash P_k::c:C_r \text{ and } \Omega_k\sqsubseteq\Omega}{\Delta;\Psi;\Gamma,a_m:\oplus_m\{L_i:(A_m^+)_i\}_I\rhd\Omega,a_m:\mathrm{C}\vdash\begin{pmatrix}\textsf{case}_J\ a_m\ \textsf{of}\\ L_j\to P_j\end{pmatrix}::c_r:C_r}\;\oplus L$$

$$\dfrac{I\subseteq J\quad \text{for all } k\in I:\ \Delta;\Psi;\Gamma\rhd\Omega_k\vdash P_k::c:(C_m^-)_k \text{ and } \Omega_k\sqsubseteq\Omega}{\Delta;\Psi;\Gamma\rhd\Omega\vdash\begin{pmatrix}\textsf{case}_J\ c_m\ \textsf{of}\\ L_j\to P_j\end{pmatrix}::c_m:\&_m\{c_i:(C_m^-)_i\}_I}\;\& R$$

$$\dfrac{k\in I\quad \Delta;\Psi;\Gamma,a_m:(A_m^-)_k\rhd\Omega,a_m:\star\vdash P::c_r:C_r}{\Delta;\Psi;\Gamma,a_m:\&\{L_i:(A_m^-)_i\}_I\rhd\Omega,a_m:\mathrm{C}\vdash\textsf{send}\ c_r\ L_k;P::c_r:C_r}\;\& L_{L_k}$$

Presuppposes for $\Delta;\Psi;\Gamma\rhd\Omega\vdash P::c_m:C_m$ that $\mathrm{dom}(\Gamma)=\mathrm{dom}(\Omega)$ and $\Gamma\geq m$

Figure 5.1: Naive Resource Tracking

$$\frac{\Delta \vdash A \quad \Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m^+[A/\alpha]}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \mathsf{send}\ c_m\ A; P :: c_m : \exists\alpha.C_m^+}\ \exists R$$

$$\frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma, a_k : A_k^+[\beta/\alpha] \triangleright \Omega, a_k : \textsc{c} \vdash P :: c_m : C_m}{\Delta; \Psi; \Gamma, a_k : \exists\alpha.A_k^+ \triangleright \Omega, a_k : \star \vdash \alpha \leftarrow \mathsf{recv}\ a_k; P :: c_m : C_m}\ \exists L$$

$$\frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m^-[\beta/\alpha]}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \alpha \leftarrow \mathsf{recv}\ c_m; P :: c_m : \forall\alpha.C_m^-}\ \forall R$$

$$\frac{\Delta \vdash A \quad \Delta; \Psi; \Gamma, b_k : B_k^- \triangleright \Omega, b_k : \star \vdash P :: c_m : C_m}{\Delta; \Psi; \Gamma, b_k : \forall\alpha.B_k^- \triangleright \Omega, b_k : \textsc{c} \vdash \mathsf{send}\ b_k\ A; P :: c_m : C_m}\ \forall L$$

$$\frac{\Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: a_k : A_k^+}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \mathsf{shift}\ a_k \leftarrow \mathsf{recv}\ b_m; P :: b_m : \uparrow_k^m A_k^+}\ \uparrow R$$

$$\frac{k \geq r \quad \Delta; \Psi; \Gamma, a_k : A_k^+ \triangleright \Omega, a_k : \star \vdash Q :: c_r : C_r}{\Delta; \Psi; \Gamma, b_m : \uparrow_k^m A_k^+ \triangleright \Omega, b_m : \textsc{c} \vdash \mathsf{shift}\ a_k \leftarrow \mathsf{send}\ b_m; Q :: c_r : C_r}\ \uparrow L$$

$$\frac{\Gamma \geq m \quad \Delta; \Psi; \Gamma \triangleright \Omega \vdash Q :: a_m : A_m^-}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \mathsf{shift}\ a_m \leftarrow \mathsf{send}\ b_k; Q :: b_k : \downarrow_k^m A_m^-}\ \downarrow R$$

$$\frac{\Delta; \Psi; \Gamma, a_m : A_m^- \triangleright \Omega, a_m : \star \vdash P :: c_r : C_r}{\Delta; \Psi; \Gamma, b_k : \downarrow_k^m A_m^- \triangleright \Omega, b_k : \textsc{c} \vdash \mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ b_k; P :: c_r : C_r}\ \downarrow L$$

Presuppposes for $\Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m$ that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Omega)$ and $\Gamma \geq m$

Figure 5.1: Naive Resource Tracking (cont.)

rule used in the proof. Most cases are straightforward, so we only show a few here.

Case Id: The channel we forward from is marked as consumed, so Γ' must include it. Any channel that is marked S must be either F or U. Together these mean we can use Id to complete this case.

Case $\otimes R$: We are given Γ' and the following:

$$\frac{\overbrace{\Delta; \Psi; \Gamma \triangleright \Omega_1 \vdash P :: a_m : A_m^+}^{\mathcal{P}} \quad \overbrace{\Delta; \Psi; \Gamma \triangleright \Omega_2 \vdash Q :: c_m : C_m^+}^{\mathcal{Q}}}{\Delta; \Psi; \Gamma \triangleright \Omega_1 \bowtie \Omega_2 \vdash \mathsf{send}\ c_m\ (a_m \leftarrow P); Q :: c_m : A_m^+ \otimes C_m^+}\ \otimes R$$

and show:

$$\frac{\overbrace{\Delta; \Psi; \Gamma_1 \vdash P :: a_m : A_m^+}^{\text{Inductive Hyp. w/ } \mathcal{P} \text{ and } \Gamma_1} \quad \overbrace{\Delta; \Psi; \Gamma_2 \vdash Q :: c_m : C_m^+}^{\text{Inductive Hyp. w/ } \mathcal{Q} \text{ and } \Gamma_2}}{\Delta; \Psi; \Gamma_1, \Gamma_2 \vdash \mathsf{send}\ c_m\ (a_m \leftarrow P); Q :: c_m : A_m^+ \otimes C_m^+}\ \otimes R$$

where $\Gamma_1 \subseteq \Gamma'$ and $\Gamma_2 \subseteq \Gamma'$ are constructed follows:

$$\Gamma_1(b_r) = \begin{cases} \Gamma(b_r) & \text{if } b_r \in \mathrm{dom}(\Gamma') \text{ and } \Omega_1(b_r) = \textsc{c} \\ \Gamma(b_r) & \text{if } b_r \in \mathrm{dom}(\Gamma') \text{ and } \Omega_1(b_r) = \textsc{s} \text{ and } \Omega_2(b_r) \neq \textsc{c} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and

$$\Gamma_2(b_r) = \begin{cases} \Gamma(b_r) & \text{if } b_r \in \text{dom}(\Gamma') \text{ and } \Omega_2(b_r) = \text{C} \\ \Gamma(b_r) & \text{if } b_r \in \text{dom}(\Gamma') \text{ and } \Omega_2(b_r) = \text{S} \text{ and } \Omega_1(b_r) \neq \text{N} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Case $\&R$**:** We are given Γ' and the following:

$$\frac{I \subseteq J \quad \text{for all } k \in I : \ \Delta; \Psi; \Gamma \rhd \Omega_k \vdash P_k :: c : (C_m^-)_k \text{ and } \Omega_k \sqsubseteq \Omega}{\Delta; \Psi; \Gamma \rhd \Omega \vdash \begin{pmatrix} \mathsf{case}_J \ c_m \ \mathsf{of} \\ L_j \to P_j \end{pmatrix} :: c_m : \&_m\{c_i : (C_m^-)_i\}_I} \&R$$

and show the following by repeated use of the inductive hypothesis:

$$\frac{I \subseteq J \quad \text{for all } k \in I : \ \Delta; \Psi; \Gamma' \vdash P_k :: c : (C_m^-)_k}{\Delta; \Psi; \Gamma' \vdash \begin{pmatrix} \mathsf{case}_J \ c_m \ \mathsf{of} \\ L_j \to P_j \end{pmatrix} :: c_m : \&_m\{c_i : (C_m^-)_i\}_I} \&R$$

□

Before proving completeness, we need an ability to weaken our environment.

Lemma 45. *If $\Delta; \Psi; \Gamma \rhd \Omega \vdash P :: c_m : C_m$, then for a_r with $r \geq m$ we have $\Delta; \Psi; \Gamma, a_r : A_r \rhd \Omega, a_r : \chi \vdash P :: c_m : C_m$ where $\chi = \text{N}$ if $r = \text{L}$ and $\chi = \text{S}$ otherwise.*

Proof. By induction on $\Delta; \Psi; \Gamma \rhd \Omega \vdash P :: c_m : C_m$. □

Theorem 46 (Completeness)**.** *If $\Delta; \Psi; \Gamma \vdash P :: c_m : C_m$, then there exists Ω such that $\Delta; \Psi; \Gamma \rhd \Omega \vdash P :: c_m : C_m$ and N is not in the range of Ω.*

Proof. By induction on $\Delta; \Psi; \Gamma \vdash P :: c_m : C_m$. Use Lemma 45 to accommodate the splitting in $\otimes R$ and $\multimap L$. □

5.2 Bidirectional Checking

Recall from section 2.6 that creating a bidirectional version of a type system, hopefully suitable for mechanization, involves defining two different judgments, one for types we can *synthesize* and one for which we can merely *check*. While it appears that it is possible [5,17,44,85] to infer session types, we use a bidirectional system where we only check the types of process expressions. Since most rules only require switching a use of "::" for "$\Leftarrow$", lets look at the more interesting ones in a bit more detail.

First, the rule for $\{\}I$ assumes that we have a bidirectional system for our underlying language as well and then merely initiates checking the process expression.

$$\frac{\Delta; \Psi; \vec{a} : \vec{A} \triangleright \Omega \vdash P \Leftarrow a : A \quad \Omega \geq \mathrm{S}}{\Delta; \Psi \vdash a \leftarrow \{P\} \multimap \vec{a} \Leftarrow A \leftarrow \vec{A}} \ \{\}I$$

For $\{\}E$, we are forced to synthesize the type of the bound process expression since the existing type does not give us any information about it.

$$\frac{\begin{array}{c}\vec{A} \geq m \geq r \quad \Delta; \Psi \vdash e \Rightarrow \{B_m \leftarrow \vec{D}\} \\ \vec{A} \sqsubseteq \vec{D} \quad \Delta; \Psi; \Gamma, b : B_m \triangleright \Omega, b_m : \star \vdash P \Leftarrow c_r : C_r\end{array}}{\Delta; \Psi; \Gamma, \vec{a} : \vec{A} \triangleright \Omega, \vec{a} : \vec{\mathrm{C}} \vdash b_m \leftarrow e \multimap \vec{a}; P \Leftarrow c_r : C_r} \ \{\}E$$

The two remaining rules that interact with the underlying functional language can both check their subexpressions.

$$\frac{\Delta; \Psi \vdash e \Leftarrow \tau \quad \Delta; \Psi; \Gamma \triangleright \Omega \vdash P \Leftarrow c_m : C_m^+}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \mathsf{send}\ c_m\ e; P \Leftarrow c_m : \tau \wedge C_m^+} \ \wedge R$$

$$\frac{\Delta; \Psi \vdash e \Leftarrow \tau \quad \Delta; \Psi; \Gamma, b_k : B_k^- \triangleright \Omega, b_k : \star \vdash P \Leftarrow c_m : C_m}{\Delta; \Psi; \Gamma, b_k : \tau \supset B_k^- \triangleright \Omega, b_k : \mathrm{C} \vdash \mathsf{send}\ b_k\ e; P \Leftarrow c_m : C_m} \supset L$$

This gives any easy accounting of where we might need type annotations, whenever we try to synthesize the type of a process expression. Given the size of process expressions, in practice, explicit ones are top-level definitions and, thus, writing a type annotation is non-intrusive. However, there is one way even fairly boring programs might try to synthesize the type of a process expression: syntactic sugar (subsection 4.1.6). Specifically, SILL's syntactic sugar generates large amounts of process expressions utilizing tail-bindings. Corollary 37 suggests suggests a solution: introduce a custom tail-bind rule that fuses $\{\}E$ and Id by checking the bound expression at its "most general" type:

$$\frac{\Delta; \Psi \vdash e \Leftarrow \{C_r \leftarrow \vec{A}\}}{\Delta; \Psi; \Gamma, \vec{a} : \vec{A} \triangleright \Omega, \vec{a} : \vec{\mathrm{C}} \vdash c_r \leftarrow e \multimap \vec{a} \Leftarrow c_r : C_r} \ \textsc{Tail}$$

This almost works. Unfortunately, due to the lack of subtyping at the underlying functional level following example would fail if we tried to use Tail:

$$\emptyset; x : \&\{\mathtt{inl} : 1\}; \emptyset \triangleright \emptyset \vdash c_{\mathrm{L}} \leftarrow x \Leftarrow c_{\mathrm{L}} : \&\{\}$$

but its desugaring is typeable (by using $\{\}E$ and Id):

$$\emptyset; x : \&\{\mathtt{inl} : 1\}; \emptyset \triangleright \emptyset \vdash a_{\mathrm{L}} \leftarrow x; c_{\mathrm{L}} \leftarrow a_{\mathrm{L}} \Leftarrow c_{\mathrm{L}} : \&\{\}$$

One reaction to this would be to force subtyping on the underlying functional language (which would be challenging for the Haskell version of SILL). Instead we pursue the simpler route of saying "If the underlying language can synthesize a type, let it," i.e., only try to provide extra guidance when it is truly needed.

Formally, this is accomplished by using the judgment $\Delta; \Psi \vdash e \not\Rightarrow$ to denote that there is no proof of $\Delta; \Psi \vdash e \Rightarrow \tau$ for any τ and slightly altering Tail:

$$\frac{\Delta; \Psi \vdash e \not\Rightarrow \quad \vec{A} \geq r \quad \Delta; \Psi \vdash e \Leftarrow \{C_r \leftarrow \vec{A}\}}{\Delta; \Psi; \Gamma, \vec{a} : \vec{A} \triangleright \Omega, \vec{a} : \vec{c} \vdash c_r \leftarrow e \multimap\!\!\prec \vec{a} \Leftarrow c_r : C_r} \textsc{Tail}$$

If this rule is inapplicable, we fall back to checking on the desugared syntax.

Due to the prevalence of tail-binds we generally want to execute in a system that operationally optimizes them:

$$\begin{aligned}
\textsc{Tail}_{\mathsf{step}} &: \mathsf{exec}_c(c \leftarrow e \multimap\!\!\prec \vec{a}) \otimes !(e \rightarrow e') \multimap \mathsf{exec}_c(c \leftarrow e' \multimap\!\!\prec \vec{a}) \\
\text{TB}\textsc{ind} &: \mathsf{exec}_c(c \leftarrow (b \leftarrow \{P\} \multimap\!\!\prec \vec{b}) \multimap\!\!\prec \vec{a}) \multimap \mathsf{exec}_c(P[c, \vec{a}/b, \vec{b}])
\end{aligned}$$

which have no impact on progress or preservation.

5.3 Coinductive Subtyping Algorithm

In this section, we follow the presentation of Gay and Hole [35] to produce an algorithm for checking the coinductive notion of subtyping (Definition 31). Nothing in this section is innovative, but it is needed for a complete picture of how the various SILL implementations work. To see why this can be challenging in our situation, consider the following example. Suppose, we wished to show that $\mu x.\texttt{int} \wedge x$ is a subtype of $\mu y.\texttt{int} \wedge \texttt{int} \wedge y$. In some sense, this should be easy because these two types are equal (both are an infinite stream of `int`), however since the types are not syntactically indentical, this equality can be tricky to discover algorithmically. Specifically, we will see in the remainder of the section how to unfold these μ-presented regular types in the right way to allow fully algorithmic checking.

We define a proof system that gives us an obvious algorithm. The system will consist of a set of pairs on which the relation is assumed to hold and a goal pair. These pairs will be suggestively written as $(A \sqsubseteq B)$, and the judgment is $\Xi \vdash A \sqsubseteq B$, where Ξ is a set of assumptions. The rules are presented in Figure 5.2. These rules can be split into the recursion enabling Rec rule and the remaining rules which enforce the local requirements of coinductive subtyping relations. The initial goal for $A \sqsubseteq B$ is $\emptyset \vdash A \sqsubseteq B$. To get an algorithm from this system we preferentially use Rec.

To ensure termination of our algorithm we restrict to working with regular types. Additionally, we will handle μs implicitly throughout the rest of this section.

Definition 47 (Regular Type)**.** *The* subterms *of a type A, denoted* Sub(A)*, are given by the following inductively defined set:*

- $A \in \textsc{Sub}(A)$
- *If* $\tau \wedge B \in \textsc{Sub}(A)$*, then* $B \in \textsc{Sub}(A)$

$$\frac{}{(A_m \sqsubseteq B_m), \Xi \vdash A_m \sqsubseteq B_m}\,\textsc{Rec} \qquad \frac{}{\Xi \vdash 1_m \sqsubseteq 1_m}\,1 \qquad \frac{}{\Xi \vdash \alpha_m \sqsubseteq \alpha_m}\,\textsc{Atom}$$

$$\frac{(\tau \wedge A_m \sqsubseteq \tau \wedge B_m), \Xi \vdash A_m \sqsubseteq B_m}{\Xi \vdash \tau \wedge A_m \sqsubseteq \tau \wedge B_m}\,\wedge \qquad \frac{(\tau \supset A_m \sqsubseteq \tau \supset B_m), \Xi \vdash A_m \sqsubseteq B_m}{\Xi \vdash \tau \sqsubset A_m \sqsubseteq \tau \supset B_m}\,\supset$$

$$\frac{(A \otimes B_m \sqsubseteq C_m \otimes D_m), \Xi \vdash A_m \sqsubseteq C_m \quad (A \otimes B_m \sqsubseteq C_m \otimes D_m), \Xi \vdash B_m \sqsubseteq D_m}{\Xi \vdash A_m \otimes B_m \sqsubseteq C_m \otimes D_m}\,\otimes$$

$$\frac{\begin{array}{c}(A_m \multimap B_m \sqsubseteq C_m \multimap D_m), \Xi \vdash C_m \sqsubseteq A_m \\ (A_m \multimap B_m \sqsubseteq C_m \multimap D_m), \Xi \vdash B_m \sqsubseteq D_m\end{array}}{\Xi \vdash A_m \multimap B_m \sqsubseteq C_m \multimap D_m}\,\multimap$$

$$\frac{I \subseteq J \quad \text{for all } k \in I\text{: } \oplus\{L_i : (A_m)_i\}_I \sqsubseteq \oplus\{L_j : (B_m)_j\}_J, \Xi \vdash (A_m)_k \sqsubseteq (B_m)_k}{\Xi \vdash \oplus\{L_i : (A_m)_i\}_I \sqsubseteq \oplus\{L_j : (B_m)_j\}_J}\,\oplus$$

$$\frac{J \subseteq I \quad \text{for all } k \in J\text{: } \&\{L_i : (A_m)_i\}_I \sqsubseteq \&\{L_j : (B_m)_j\}_J, \Xi \vdash (A_m)_k \sqsubseteq (B_m)_k}{\Xi \vdash \&\{L_i : (A_m)_i\}_I \sqsubseteq \&\{l_j : (B_m)_j\}_J}\,\&$$

$$\frac{(\uparrow_m^k A_m \sqsubseteq \uparrow_m^k B_m), \Xi \vdash A_m \sqsubseteq B_m}{\Xi \vdash \uparrow_m^k A_m \sqsubseteq \uparrow_m^k B_m}\,\uparrow \qquad \frac{(\downarrow_m^k A_m \sqsubseteq \downarrow_m^k B_m), \Xi \vdash A_m \sqsubseteq B_m}{\Xi \vdash \downarrow_m^k A_m \sqsubseteq \downarrow_m^k B_m}\,\downarrow$$

Figure 5.2: Coinduction for Subtyping

- *If* $\tau \supset A \in \textsc{Sub}(A)$*, then* $B \in \textsc{Sub}(A)$
- *If* $B \otimes C \in \textsc{Sub}(A)$*, then* $B \in \textsc{Sub}(A)$ *and* $C \in \textsc{Sub}(A)$
- *If* $B \multimap C \in \textsc{Sub}(A)$*, then* $B \in \textsc{Sub}(A)$ *and* $C \in \textsc{Sub}(A)$
- *If* $\oplus\{L_i : (C_m)_i\}_I \in \textsc{Sub}(A)$*, then for all* $i \in I$ *we have* $(C_m)_i \in \textsc{Sub}(A)$
- *If* $\&\{L_i : (C_m)_i\}_I \in \textsc{Sub}(A)$*, then for all* $i \in I$ *we have* $(C_m)_i \in \textsc{Sub}(A)$
- *If* $\uparrow_m^k B_m \in \textsc{Sub}(A)$*, then* $B_m \in \textsc{Sub}(A)$
- *If* $\downarrow_k^m B_m \in \textsc{Sub}(A)$*, then* $B_m \in \textsc{Sub}(A)$

A type A *is called* regular *if* $\textsc{Sub}(A)$ *is finite.*

From here on we will assume that all our types are regular. Notice, that for the types we μs we have used throughout this thesis we can only ever generate regular types. We mention this quirk for two reasons, to be explicit in a secion that deals more with the quirks of regular types and to minimize confusing with our implicit treatment of μs. Returning to the example at the start of this section, we can see that the subterms of both types is, after α-conversion, the singleton set $\{\mu x.\texttt{int} \wedge x\}$.

Lemma 48. *If* A *and* B *are regular, then the algorithm terminates for* $A \sqsubseteq B$*.*

Proof. Notice that when starting from $\emptyset \vdash A \sqsubseteq B$, the intermediate Ξs used by the algorithm must be a subset of $\textsc{Sub}(A) \times \textsc{Sub}(B)$, thus Ξ is always finite. At every rule, we either terminate that branch of the proof search or recurse on a strictly larger Ξ. This means that we will eventually either find a mismatched node (i.e., we can report that $A \neq B$) or be able to complete our proof with one of Rec, 1, or Atom. □

We use a notion of a sound goal to denote one which should have a proof.

Definition 49 (Sound Circular Subtyping Goal)**.** *A goal $\Xi \vdash A \sqsubseteq B$ is sound if $A \sqsubseteq B$ and for all $(C \sqsubseteq D) \in \Xi$, $C \sqsubseteq D$*

Lemma 50. *If a subtyping goal is sound, then its immediate subgoals are sound and a rule is applicable.*

Proof. Let $\Xi \vdash A \sqsubseteq B$ be a sound goal. If Rec is applicable, the lemma holds. For the other rules, there are either no subgoals, or soundness follows from the applicable rule and the definitions of subterms and equality. If no rule is applicable, then there is a labeling mismatch at the root and thus $A \not\sqsubseteq B$, a violation of our assumption that $\Xi \vdash A \sqsubseteq B$ is sound. □

Lemma 51. *If the algorithm returns a proof starting from $\Xi \vdash A \sqsubseteq B$, then it will return a proof starting from $\Xi' \vdash A \sqsubseteq B$, where $\Xi \subseteq \Xi'$.*

Proof. A larger set of assumptions can only allow us to use Rec more readily, it will not prohibit any of the rules used in the originally returned proof. □

Lemma 52. *If $A \sqsubseteq B$, then the algorithm returns a proof of $\Xi \vdash A \sqsubseteq B$.*

Proof. By Lemma 51 it is sufficient to consider the case of $\emptyset \vdash A \sqsubseteq B$, which is sound. Thus, by Lemma 50 we have that a rule is applicable and that all the subgoals are sound, thus the algorithm can proceed. Since the algorithm terminates (Lemma 48) and we cannot get stuck we must return a proof of $\emptyset \vdash A \sqsubseteq B$, which by Lemma 51 means we can return a proof of $\Xi \vdash A \sqsubseteq B$. □

Corollary 53 (Completeness)**.** *If $A \sqsubseteq B$, then $\emptyset \vdash A \sqsubseteq B$.*

Lemma 54. *If $\emptyset \vdash A \sqsubseteq B$ and $(A \sqsubseteq B) \vdash C \sqsubseteq D$ is an immediate subgoal of $\emptyset \vdash A \sqsubseteq B$, then $\emptyset \vdash C \sqsubseteq D$.*

Proof. If the proof of $(A \sqsubseteq B) \vdash C \sqsubseteq D$ does not use Rec, we are done. Otherwise, find the use of Rec for $(A \sqsubseteq B)$ and replace it with the proof of $\emptyset \vdash A \sqsubseteq B$, adjusting via Lemma 51 as needed. □

Theorem 55 (Soundness)**.** *If $\emptyset \vdash A \sqsubseteq B$, then $A \sqsubseteq B$.*

Proof. The relation given by $R = \{(A, B) | \emptyset \vdash A \sqsubseteq B\}$ is a coinductive subtyping relation. The proof by cases of A is are all similar, so we show only the case for $A = \uparrow^k_m C_m$. Consider a proof of $\emptyset \vdash \uparrow^k_m C_m = B$. Since Rec is inapplicable, the proof must start with $\uparrow$. Thus $B = \uparrow^k_m D_m$ as needed. To see that $(C_m, D_m) \in R$, notice that it is a subgoal of $\emptyset \vdash A = B$ and then apply Lemma 54. □

5.4 Working With Infinite Equirecursive Types

Throughout this thesis we have assumed that types are equirecursive, i.e., implicitly allowing them to be infinite (if regular). An implementation cannot work with infinite types directly, so we will need some finitary representation. Two that have been used in the various implementations are pointer based graphs (section 5.5) and μ-based explicit fixed points (section 5.6 and section 5.7).

Notice that the proofs of section 5.3 work even if we have multiple distinguishable copies of types in $\textsc{Sub}(A)$, so long as we maintain finiteness. As an example, imagine a slightly odd version of $\textsc{Sub}$ that had the returned a finite multiset instead: $\textsc{Sub}(\texttt{Int} \wedge 1) = \{\texttt{Int} \wedge 1, 1, 1'\}$ where 1 and $1'$ are only accidentally distinguished (e.g., because an implementation might wish to use cheap pointer equality to approximately distinguish terms). When proving equality using a multiset of subterms, we may require a larger proof (i.e., $\textsc{Rec}$ will be harder to apply) but will not give up termination. This observation will free us from needing to come up with either minimal or canonical representations of types, and, in particular, means that we can use a finer grained notion of equality on types to perform the search implicit in prioritizing the use of $\textsc{Rec}$. In particular, we wish to avoid requiring a fully functioning implementation of regular type equality to define regular type equality, but, rather, only something easy to provide (e.g., purely syntactic equality on terms).

The first mechanism for representing types is to store every subterm in a reference. Since pointers can form cycles in memory and our types are regular, this allows us to replace multiple occurrences of a subterm with a reference back to some previous definition of that subterm (possibly introducing a cycle). Since memory is finite, a value of that form must have a finite number of subterms. Since we do not need canonicity, we do not even need to take care to reduce this representation, we may have multiple instances of the same abstract subterm (like 1 and $1'$). The set of assumptions we search when trying to apply $\textsc{Rec}$ then contains references to subterms and we can use cheap pointer equality tests to determine whether a match as been found.

A purer alternative to pointer based representations is to leave μs explicit. This does not give up on equirecursivity, but will require us to potentially unfold our types to reach a non-μ connective. As with any μ-based representation this introduces the ability to have non-contractive types. We will assume, as usual, that such illformed types are prohibited. Syntactic equality on the μ-based representation will be taken as our fine grained notion equality. Preserving finiteness of sets of subterms is slightly trickier. To see one issue that arises, consider the type $\mu\, x.\texttt{Int} \wedge x$. By itself, this can be represented in an infinite number of ways, e.g., the μ can be unfolded an arbitrary number of times. This can be avoided by considering our coinductive proof search more carefully. There is nothing more to be gained from examining $\textsc{Rec}$, but consider trying to decide which of the remaining rules to apply to a pair of μ-based terms. If neither is of the form $\mu x.A$, then we can apply the appropriate rule directly. However, if either type starts with a μ, we must first unfold the type at least once. Fortunately, there is no need to unfold more than is needed to remove any μs at the top of the type or to unfold inside the topmost connective. Thus we can see that the circular coindutive results still hold when using the following slightly altered notion of $\textsc{Sub}$.

Definition 56 ($\textsc{Sub}_\mu$)**.** *The* μ-subterms *of a type A, denoted $\textsc{Sub}_\mu(A)$, are given by the following inductively defined set:*

- $A \in \textsc{Sub}_\mu(A)$
- *If* $\mu x.B \in \textsc{Sub}_\mu(A)$*, then* $B[\mu x.B/x] \in \textsc{Sub}_\mu(A)$
- *If* $\tau \wedge B \in \textsc{Sub}_\mu(A)$*, then* $B \in \textsc{Sub}_\mu(A)$
- *If* $\tau \supset A \in \textsc{Sub}_\mu(A)$*, then* $B \in \textsc{Sub}_\mu(A)$
- *If* $B \otimes C \in \textsc{Sub}_\mu(A)$*, then* $B \in \textsc{Sub}_\mu(A)$ *and* $C \in \textsc{Sub}_\mu(A)$
- *If* $B \multimap C \in \textsc{Sub}_\mu(A)$*, then* $B \in \textsc{Sub}_\mu(A)$ *and* $C \in \textsc{Sub}_\mu(A)$
- *If* $\oplus\{L_i : (B_m)_i\}_I \in \textsc{Sub}_\mu(A)$*, then for all* $k \in I$*,* $(B_m)_i \in \textsc{Sub}_\mu(A)$
- *If* $\&\{L_i : (B_m)_i\} \in \textsc{Sub}_\mu(A)$*, then for all* $k \in I$*,* $(B_m)_i \in \textsc{Sub}_\mu(A)$
- *If* $\uparrow_k^m B_m \in \textsc{Sub}_\mu(A)$*, then* $B_m \in \textsc{Sub}_\mu(A)$
- *If* $\downarrow_m^k B \in \textsc{Sub}_\mu(A)$*, then* $B_m \in \textsc{Sub}_\mu(A)$

5.5 OCaml

The main implementation of SILL is an OCaml interpreter supporting the features discussed in chapter 4, excluding bundled messaging (section 4.2), select (section 4.3), and asynchronous receiving (section 4.4). Since this is the main implementation of SILL, it follows the presentation of SILL in this thesis fairly closely (e.g., we use the bidirectional system of section 5.2), with a few syntactic quirks to ease parsing and fixing a concrete underlying functional language. The underlying language is an extremely basic functional language with algebraic datatypes, mutually recursive functions, the minimal required extras to support SILL, and like the one in section 2.6, requires more type signatures than might be strictly neccessary. Additionally, we provide an ability to ignore polarization, when not moving between modes, by default (automatically polarizing via Figure 4.6). This can be disabled with the `-strict-polarity` option to the interpreter.

The full syntax of this version of SILL is presented in section A.4. The primary changes from the syntax used through the rest of this thesis are: sending a choice is handled as though accessing a field record (e.g., `c.L; ...`) due to an overlap between our syntactic class of labels and that of data type constructors; no mode subscripts, instead session type declarations have a modality prefix (`ltype` for L; `atype` for F; `utype` for U) and subscripts on type variables are replaced with a prefix, `'` for L, `@` for F, and `!` for U; the two arrows ($\uparrow_m^k$ and $\downarrow_k^m$) are replaced with the modality tags for their target mode k (we can always determine m from context). Additionally, we provide an `abort` construct that halts all execution (if possible), typed with the following rule:

$$\frac{}{\Delta; \Psi; \Gamma \triangleright \mathrm{dom}(\Gamma) : \vec{\mathrm{s}} \vdash \texttt{abort} :: c_r : C_r} \textsc{Abort}$$

The OCaml implementation of SILL offers a pluggable framework for interpretation backends: by defining a small set of functions (roughly six of which are interesting) we can radically change our evaluation strategy. While not all of them are currently maintained, we have implemented backends that: implement the semantics as a shared memory system using OCaml's user level threads; explore the different styles of forwarding presented in section 4.8; implement synchronous communication in the style of Toninho *et al.* [90]; implement channels as `unix` pipes between separate processes; implement channels via `ssh` connections; and an backend based on `mpi`. Some technical limitations made the last two of these not as exciting as hoped, however they did reveal that SILL does not have a good accounting of how to distribute processes (e.g., we should study a hybrid version [14, 15] of polarized adjoint logic). In addition to being obviously profitable, the tail-bind optimization (section 5.2) turns out to be suprisingly important in some backends, critically helping to avoid the bumping into caps on the maximum number of simultaneously live processes.

To guide our explorations and help determine what language optimizations might be worth pursuing the SILL intepreter collects a variety of performance statistics. The simplest statistic gathered is the total number of processes created over the lifetime of the program (i.e. number of times $\textsc{Send}_{\mathsf{chan}}$, $\textsc{Bind}_+$, or $\textsc{Bind}_-$ execute). When using the tail-bind optimization we track the number of processes whose creation is avoided (i.e., the number of times $\textsc{TBind}$ executes). Similarly, we track the number of times forwarding is used. The last two statistics are less broadly applicable. First, we gathered the number of times a process sent two consecutive messages along the same channel occured without an intervening usage of another channel. The optimization we thought this showed would be interesting to investigate ultimately transformed into the bunlded semantics of section 4.2.

The last and most complicated statistic that SILL gathers, requires some extra background knowledge. Vector Clocks [32, 55] are a technique for tracking the progress of time in concurrent systems that allows for more accurate causal relations in asynchronous behavior than with purely global notions of time. Every process keeps an estimate of how much time has elapsed for each process (generally discretized to the natural numbers) and updates its estimates as it discovers more information. To make this more formal, we let Π_p represent the time estimate map for process p and order two such maps by the pointwise lifting of the ordering on time (i.e., $\Pi_p \leq \Pi_q$ if for all processes r, $\Pi_p(r) \leq \Pi_q(r)$). Whenever a message is sent between processes, the sender updates its time estimate for itself (generally by incrementing it) and sends its time estimates along with the real message. The receiving process, upon reception, updates its estimates to the pointwise maximum of the estimates passed with the message or its own and then increments its time estimate for itself. This allows us to guarantee that events in a single process are ordered and that time respects causal relationships between processes as well. In the context of SILL, the top level process's estimate for when it sends its `end` message provides an abstract notion of how long all processes took to execute. We hope that by contrasting this time to the overall number of operations used during an exeuction (i.e., its purely sequential execution time) we can develop a useful type-directed notion of how much parallelism is available in a program.

5.6 Haskell

Thanks to the relatively rich features of Haskell's type system we can reflect the typing rules of the fragment of SILL presented in section 4.1 in Haskell. The basic approach is to create an embedded domain specific language [43] that represents the constructs of SILL as an Abstract Syntax Tree (AST) with relatively complicated phantom typing constraints and then interpret these ASTs at run time to execute the desired communication side-effects. This approach will allow us to use Haskell as our underlying functional language, a much more expressive language than the toy system used in the OCaml interpreter (section 5.5). Since working directly with an AST will end up being relatively unpleasant, we also explore an indexed monadic approach that will allow us to use Haskell's `do`-notation. Lastly, we will see how we can perform type directed optimizations of our implementation using the notion of boundedness (Definition 9).

Generalized Algebraic Datatypes (GADTs) [45] are a type extension that allows for the resulting type of each constructor to vary (e.g., by utilizing different phantom types). GADTs also allow for easy existential quantification over types by mentioning type parameters which do not appear in the resulting type. Using GADTs we can create a representation of SILL's session types (Figure 5.3). Notice that we use existential quantification here: the `a` parameter of the `SendD` and `RecvD` constructors do not occur in the resulting parameterless Session type. Since we need a finitary representation of our types, we utilize an explicit μ and a variable (section 5.4), which treats each variable as bound by the most recent enclosing μ. This representation could be enriched by indexing each μ and Var to enable complicated nested bindings, but this limited representation will be sufficient for the examples in this section. The other constructors directly

```
data Session where
  Mu :: Session -> Session
  Var :: Session
  One :: Session
  SendD :: a -> Session -> Session
  RecvD :: a -> Session -> Session
  SendC :: Session -> Session -> Session
  RecvC :: Session -> Session -> Session
  External :: Session -> Session -> Session
  Internal :: Session -> Session -> Session
  SendShift :: Session -> Session
  RecvShift :: Session -> Session
```

Figure 5.3: GADT for SILL Types

```
subst :: Session -> Session -> Session
subst _ One = One
subst _ (Mu x) = Mu x
subst x Var = x
subst x (SendD a s) = SendD a (subst x s)
subst x (RecvD a s) = RecvD a (subst x s)
subst x (External s1 s2) = External (subst x s1) (subst x s2)
subst x (Internal s1 s2) = Internal (subst x s1) (subst x s2)
subst x (SendShift s) = SendShift (subst x s)
subst x (RecvShift s) = RecvShift (subst x s)

unfold :: Session -> Session
unfold (Mu x) = subst (Mu x) x
unfold s = s
```

Figure 5.4: Unfolding SILL Session Types

correspond to their logical connectives: `One` for 1; `SendD` for $\wedge$; `RecvD` for $\supset$; `SendC` for $\otimes$; `RecvC` for $\multimap$; `External` for $\&$; `Internal` for $\oplus$; `SendShift` for $\downarrow$; and `RecvShift` for $\uparrow$. Our finitary representation of SILL session types will require us to be able to unfold the μ-bindings in our types (for contractive types). To accomplish this we define a simple substitution function and then utilize it to perform unfolding (Figure 5.4).

While the type of Figure 5.3 is sufficient to represent types in a Haskell interpreter for SILL, we aim for a Haskell DSL for SILL in this section (to make this more than just a reimplementation of section 5.5). In particular, we want to be able to report type errors at compile time and not only when a particular DSL AST is interpreted. To enable this we will need one extra functional programming technique, phantom types, and a type extension, `DataKinds`.

Phantom Types [24] are a way to add additional type level information to algebraic data types by including a type parameter not actually constrained by any of the data types constructors. As a simple example, we might work with strings tagged as safe or tainted as seen in the following code listing:

```
data TagString a = TagString String
data Safe = Safe
data Tainted = Tainted
readInput :: IO (TagString Tainted)
validate :: TagString Tainted -> Maybe (TagString Safe)
```

Notice that the type of TagString does not utilize the type parameter `a` within its constructor, thus we are free to instantiate it with either of our two tag types. The SILL DSL will use phantom types to statically track the typing environment used in typechecking SILL programs. However, we still need a way to promote our session data types to be usable at the type level.

By default, Haskell only allows us to express kind constraints on type parameters from a relatively limited class of kinds generated by the following grammer:

$$k ::= \star \mid k \to k$$

where $\star$ is the kind of data types and $k \to k$ allows for type level functions. The `DataKinds` extension [98] enriches these constraints by allowing data types[1] to be promoted to kind constraints while treating their constructors as types that satisfy the constraints. To see this in action, notice that the tagged string example allows for nonsensical phantom types such as `TagString Int`. The `DataKinds` extension will allow us to create a type to hold both `Safe` and `Tainted` and then, after promotion, us this to constrain the phantom type to only legal values. The code is shown in the following listing:

```
data Tag = Safe | Tainted
data TagString (a::Tag) = TagString String
readInput :: IO (TagString Tainted)
validate :: TagString Tainted -> Maybe (TagString Safe)
```

Notice that the functions for working with our tagged strings have not changed their types.

Using `DataKinds` we now have the tools needed to represent the proofs in SILL's type system in Haskell's type system. Specifically, we will transform $\Psi; \Gamma \vdash P :: c : C$ into an AST where Γ and C are phantom types and c will be left implicit (i.e., left and right rules will be distinguished directly in the AST's constructors and not by channel names). To do this we need to answer two questions: What will our phantom typing environment look like (beyond involving promoted session types)? How do we decide session type equality at the type level? The first question is relatively easy to answer, we will use lists of `Maybe Session` to track the type of each channel, ordered by channel binding textual order (i.e., newer bindings are at the end of the list). This representation has two important features to constrast to the explicitly named representation

[1]Technically, only promotable data types. We will only promote promotable data types, so we will not discuss the restrictions.

utilized by SILL's typing rules: first, we do not need to figure out how to promote variable names to the type level; second, since channels can become unusable throughout a computation without becoming textually invisible (e.g., $1L$), we wrap our `Session` type in a `Maybe` to enable unusable channels to be marked with `Nothing`. The second question, deciding type-level session type equality, requires a new type extension: Type Families.

Type families [21] provide a mechanism for specifying type level functions directly in Haskell. Type families work well with `DataKinds`, allowing us to compute with phantom types , and when specifying closed type families,[2] even permit overlapping cases that are resolved by textual order. As an example consider the following type family that computes whether a type-level list of `Maybe`s is composed entirely of `Nothing`s, returning a promoted `Bool`. To distinguish between data-level and type-level lists Haskell uses `'[]` and `':` for the type-level version of nil and cons, respectively.

```
type family AllNothing (l::[Maybe a]) :: Bool where
  AllNothing '[] = True
  AllNothing (Nothing ': xs) = AllNothing xs
  AllNothing ys = False
```

In practice, we use the `Singletons` package of Template Haskell functions to automatically generate these type family definitions from data-level functions.

We now have the tools to examine some of the simple SILL DSL constructs in detail. Each type rule will correspond to a construct in a Process AST GADT. Due to the size of this GADT, we will examine its constructors incrementally. The top level declaration of the Process GADT takes in a list of `Maybe Session`, its typing environment, and a `Session` for the type of the channel it provides:

```
data Process :: [Maybe Session] -> Session -> * where
```

The easiest rule to implement, $1R$, will be named `Close` and takes no arguments. However, it comes with two side conditions. First, the type that the process provides, `t`, may include some amount of `Mu` constructors before the `One` constructor we need, thus, we use a type-family based type-level version of unfold, called `Unfold`, to assert that the unfolded version of this type is equal to `One`. Second, since we never shrink our environment but the typing rule requires an empty environment, we confirm that the environment has been entirely marked as unusable.

```
  Close :: (Unfold t ~ One
           ,AllNothing env ~ True) => Process env t
```

The other rules are similar in how they manipulate types (i.e., using `Unfold` to confirm the expected type is present and possibly using a type-level function to check some side condition), however many of them are complicated by a

[2]All of our uses of type families are closed.

need to refer to channels that are in the environment. To avoid the possibility duplication of channels by the DSL user, we will refer to channels by reference (which will be safe to duplicate) rather than directly. Since we have chosen to represent the environment as a list, we use natural number indices into this list as references. To avoid creating a specialized version of the left rules for each possible channel name (i.e., natural number) each might use, we need a way to link concrete runtime channel values to our phantom type. Happily, there is a relatively convenient way to do that via the `singletons` library.

The `singletons` library [31] is a template Haskell library that provides a limited form of dependent types by generating singleton versions of datatypes and mapping them to type level representatives. In more detail, every value of a data type is turned into its own singleton type with exactly one constructor, giving us a tight connection between data-level values and their type-level equivalents. This is somewhat inconvenient to work with since every value is its own type and functions over this type need to be just polymorphic enough to allow these values but no unindended arguments. With careful, i.e., via singletons generated automatically with Template Haskell, usage of type classes these singleton values can be uniformly treated as values whos type contains a phantom type that is guaranteed to be the promoted value that the singleton represents. As an example, consider trying to write the `Process` AST constructor for $1L$. This must take an argument specifying which channel to wait on and the subprocess to continue as. Notice that, if we knew the type of the argument channel, this would be no harder than the case for $1R$. The singleton version of the `Nat` datatype (`data Nat = Z | S Nat`) has two constructors, `SZ` and `SS`, one for each of the constructors of `Nat`. At the type level the `Nat` represented by this is bound as a type parameter of `SNat`, the type of singleton natural numbers. Generating this singleton data type is done via the following call to Template Haskell:

```
singletons [d| data Nat = Z | S Nat |]
```

The constructor for $1L$, called `Wait`, is given by the following listing, where `Inbounds`, `Index`, and `Update` are three utility functions that ensure our index is inbounds, returns the result of indexing into a list, and updates a list at a given index, respectively:

```
  Wait :: (Inbounds n env ~ True
          ,Index n env ~ Just t
  ,Unfold t ~ One)
=> SNat n
-> Process (Update n Nothing env) s
-> Process env s
```

Notice, that in the environment of the continuation process we have updated the channel that was waited on to be marked as unusable (i.e., `Nothing`). This ensures that we cannot use a channel after it is closed. To see this in action, the following example shows a process that waits on a terminated channel and then terminates itself.

```
foo :: Process '[ Just One ] One
foo = Wait SZ Close
```

As mentioned before, the remaining constructors are all straightforward adoptions of their respective typing rules, possibly with a few extra type-level utility functions (e.g., session equality via circular coinduction). While this gives us a technically working solution, the syntax is quite ugly: explicitly writing `SNat`s, even after providing a `Num` type class instance, is a far cry from the named channels seen in the formal syntax; and writing down an AST directly is much uglier than the `do`-notation-like used in the formal syntax. Both of these limitations are solvable, but will require a bit more infrastructure.

Monads have the kind $\star \to \star$, which means that the type of a monadic expression can only vary by changing the $\star$ kinded type parameter. If we are interested in tracking information via phantom types as our computation runs, this can be too limiting. One approach to deal with this problem is indexed monads [56]. This typeclass allows for instances that have an input and output (phantom) type index in addition to the normal return type of the expression. The class definition is given by the following listing and, with the `RebindableSyntax` extension, can be used directly with `do`-notation.

```
class IxMonad m where
  return :: a -> m k k a
  (>>=) :: m k1 k2 a -> (a -> m k2 k3 b) -> m k1 k3 b
  (>>) :: m k1 k2 a -> m k2 k3 b -> m k1 k3 b
  fail :: String -> m k1 k2 a

  m >> n = m >>= const n
  fail = error
```

With this we can, for straightline process code, use the `do`-notation to provide a clean interface to the DSL. Unfortunately, this requires us to always provide both an input and output phantom type for each process, which is ill-defined in the case of a completed process (i.e., one that ends in ID or $1R$). To work around this we define a new wrapper, `PState`, that indicates whether a process is terminated or running with a particular typing environment and type of provided channel:

```
data PState where
  Term :: PState
  Running :: [Maybe Session] -> Session -> PState
```

Before providing an indexed monad instance we also need to determine what information we need at run time to execute a process. We will defer this until subsection 5.6.1 and assume that it fits into a placeholder ExecState type. We then define the indexed state of a process, IState, as the following, where the prelude imported qualified as `P`:

```
newtype IState (k1::PState) (k2::PState) a
  = IState {runIState :: (ExecState -> IO (a,ExecState))}
instance IxMonad IState where
  return a = IState (\s -> P.return (a,s))
  v >>= f = IState (\i -> runIState v i P.>>=
        \ (a,m) -> runIState (f a) m)
```

The individual operations inside the `IState` indexed monad have very similar types to the previous DSL, however process continuations become output type indicies instead of arguments to an AST constructor. To see some simple examples consider the type declarations for the functions coresponding to $1R$ and $1L$:

```
close :: (AllNothing env ~ True
         ,Unfold s ~ One)
      => IState (Running env s) Term ()


wait :: (Inbounds n env ~ True
        ,Index n env ~ Just t
, Unfold t ~ One)
     => SNat n
     -> IState (Running env s)
               (Running (Update n Nothing env) s) ()
```

These have almost identical types. The two major differences is that `close` uses `Term` as its output type, indicating that a process ending in close is fully defined, and that `wait` has an output type instead of a continuation subprocess. The other definitions are straightforward adaptions of our AST to the indexed monadic presentation.

In the original AST, the channel reference `SNat`s were directly inserted into the AST. With our monadic presenation we can hide the details of `SNat` references via Haskell's existing monadic binders. As a result, rules that need to refer to channels can refer to them via some pleasant name (e.g., c) instead of a natural number. In the case of $\multimap R$ (and $\otimes L$ and $\{\}E$) this is accomplished by having the monadic expression return the appropriate `SNat`, which can then be bound as normal for a monadic return value. For example, the type of the operaton for $\multimap R$ is:

```
recvcr :: (Unfold u ~ (RecvC s1 s2))
       => IState (Running env u)
                 (Running (env :++ '[ Just s1]) s2)
 (SNat (NatLen env))
```

which in addition to doing the expected unfolding, ensures that the continuation process will have one more channel in its environment (`:++` is the type-level list append function) and returns the `SNat` that corresponds to the newly inserted environment binding.

The two rules that take two possible continuation proccesses, $\&R$ and $\oplus L$, do not cleanly fit into `do`-notation. Instead they directly take subprocess arguments (`extchoir` for EXTernal CHOIce Right and `exthoil` for EXTernal CHOIce Left):

```
extchoir :: (Unfold u ~ (External s1 s2))
  => IState (Running env s1) t a
  -> IState (Running env s2) t a
  -> IState (Running env u) t a

extchoil :: (Inbounds n env ~ True
            ,Index n env ~ Just u
    ,Unfold u ~ (Internal s1 s2))
  => SNat n
  -> IState (Running (Update n (Just s1) env) t) k a
  -> IState (Running (Update n (Just s2) env) t) k a
  -> IState (Running env t) k a
```

The last bit of syntactic clean up is to simplify the treatment of process declarations by creating a type family to expand a `Process` type expression into a function with an appropriate number of `SNat` arguments. This enables us to write top level bindings that directly use function arguments instead of needing initial `let` or `where` bindings to name the initial channels in a process's environment. As an example, consider this start of a declaration for process that has two argument processes:

```
binary :: Process '[Just (Bag Nat), Just (Bag Nat)]
  (Bag Nat)
binary l r = ...
```

Ignoring the definition of `Bag Nat`, this shows how easy working with environments can be with the `Process` type family. More formally, the `Process` type family takes in an initial session environment and session type of channel provided by that process and returns a function with a number of `SNat` arguments equal to the size of the environment and that results in an `IState` with a `Running` process with the given environment and provided channel type as its input type index and `Term` as its output type index. This is defined via two type families, `VarArgs`, which provides the function arrows, and `Process`, which initializes `VarArgs` and hides the `Running`/`Term` details.

```
type family VarArgs (n::Nat) (args1::[a]) (base::b) :: *
  where VarArgs n '[] base = base
```

```
        VarArgs n (x ': xs) base = SNat n
               -> VarArgs (S n) xs base

type Process env s = (AllWellformed env ~ True) =>
   VarArgs Z env (IState (Running env s) Term ())
```

5.6.1 Execution

As in the SILL semantics (subsection 4.1.5), we will implement the monadic operations in terms of directed, reversible when empty, queues. To do this we will utilize Haskell's Software Transactional Memory and perform type directed code generation to ensure that queues point in the correct direction. Before describing the details of `ExecState` and our channels' queues, we need to determine the messages that can be sent. We will take a messaging view of forwarding (section 4.8), so our messages can be defined, with one constructor per message kind, by the following GADT, where `ExtDiQueue` a polymorphic channel implementation we will describe later:

```
data Comms where
  COne :: Comms
  CShift :: Comms
  CData :: a -> Comms
  CChoice :: Bool -> Comms
  CChan :: (ExtDiQueue Comms) -> Comms
  CForward :: (ExtDiQueue Comms) -> Comms
```

During execution we will associate an `ExtDiQueue` with each channel used or provided by a process. To allow `CForwarding` messages to update channel bindings we store each of these queues in a mutable `IORef`. Thus, `ExecState` is defined as a list of queues for those channels in its environment and a single queue for the channel that it provides:

```
data ExecState = ExecState [IORef (ExtDiQueue Comms)]
                           (IORef (ExtDiQueue Comms))
```

Software transactional memory (STM) [29, 38, 86] allows for the easy design of race free programs by structuring potentially racy portions of programs as transactions. GHC provides an easy to use monadic interface to its STM implementation. Three different STM types will be of particular interest to us: `TVar`, `TQueue`, and `TMVar`. A `TVar` is a transactional variable and can be read or written like a normal reference. The interesting feature of a `TVar` is that reads or writes during a transaction occur atomically w.r.t. that transaction. Thus, even a program that accesses multiple `TVars` from multiple processes can ensure race freedom. The `TQueue` type provides a standard queue interface but with a transactional flavor. Lastly, the `TMVar` type provides a transactional combination of a mutex and a `TVar` (similar to a `TVar` holding a `Maybe`), where the `TMVar` is *empty* if some process has *taken* the `TMVar` and is *full* if after a process *puts* a value in it. Transactions can be aborted early by using the `retry` function and, when efficiently implemented, will wait until some used transactional variable has changed before rerunning (i.e., what one might do with `wait`/`notify`).

Using STM we define a `DiQueue` type class that provides a directed queue with two distinguished ends (`C` for client, `P` for provider) and the ability for the client or provider to read, write, wait to write, or swap the direction of the queue. Attempting to perform an operation incompatible with the current queue direction may temporarily block the process, but, if our type system is correct, this will never introduce deadlock. The signatures for the `DiQueue` operations are given by the following:

```
class DiQueue a where
  safeReadC :: a b -> STM b
  safeReadP :: a b -> STM b
  safeWriteC :: a b -> b -> STM ()
  safeWriteP :: a b -> b -> STM ()
  waitToC :: a b -> STM ()
  waitToP :: a b -> STM ()
  swapDir :: a b -> STM ()
```

The first, most general, instance of our `DiQueue` type class, unbounded `DiQueues` (`UDiQueue`) is is implemented in terms of a data-type for the queue direction, a `TQueue`, and a `TVar` to track the direction of the queue:

```
data Dir = ToC | ToP deriving (Eq, Show)
invertDir :: Dir -> Dir
invertDir ToC = ToP
invertDir ToP = ToC

data UDiQueue a = UDQ { dirU :: TVar Dir
                      , queU :: TQueue a }

-- Bool indicates to start pointing at Client or Provider
newUDiQueue :: Bool -> IO (UDiQueue a)
newUDiQueue b = do q <- newTQueueIO
                   d <- newTVarIO (if b then ToC
                                         else ToP)
                   return (UDQ d q)
```

The other operations are implemented in terms of a function that waits for the `UDiQueue` to point in the correct direction and then performs the operation:

```
-- Wait until the queue is pointing the specified
-- direction and then modify the TQueue
withDirU :: Dir -> (TQueue a -> STM b) -> UDiQueue a
```

```
         -> STM b
withDirU d f q = do qd <- readTVar (dirU q)
                    if d == qd
                    then f (queU q)
                    else retry

instance DiQueue UDiQueue where
  safeReadC = withDirU ToC readTQueue
  safeReadP = withDirU ToP readTQueue
  safeWriteC q x = withDirU ToP ((flip writeTQueue) x) q
  safeWriteP q x = withDirU ToC ((flip writeTQueue) x) q
  waitToC = withDirU ToC (\_ -> return ())
  waitToP = withDirU ToP (\_ -> return ())
  swapDir q = modifyTVar' (dirU q) invertDir
```

The second instance of the `DiQueue` type class, `BDiQueue`, encodes directed queues that know an upper bound on the number of elements they will store. This is done by replacing the `TQueue` of `UBiQueues` with an `Array` of `TMVars` and a pair of `Ints` to track the position of the next array location to read and write. These `TMVars` start empty and are filled in by the process writing to the queue. The reading process blocks whenever it encounters an empty `TMVar` and takes the `TMVar` when it finds a full `TMVar`. Because the reading process takes each `TMVar` as it reads it and the type system guarantees all written values will be read before a `swapDir` is needed, we do not need to walk the entire array and empty it during a `swapDir`. The complete definition of BDiQueue is the following:

```
data BDiQueue a = BDQ { dirB :: TVar Dir
                      , readPos :: TVar Int
                      , writePos :: TVar Int
                      , queB :: Array Int (TMVar a) }

-- Wait until the queue is pointing the specified
-- direction and then modify the queue
withDirB :: Dir -> (TVar Int -> TVar Int
                   -> Array Int (TMVar a) -> STM b)
         -> BDiQueue a -> STM b
withDirB d f q =
  do qd <- readTVar (dirB q)
     if d == qd
     then f (readPos q) (writePos q) (queB q)
     else retry

-- Bool indicates to start pointing at Client or Provider
```

```
newBDiQueue :: Bool -> Int -> IO (BDiQueue a)
newBDiQueue b i =
  do q <- mapM (\_ -> newEmptyTMVarIO) [0..i-1]
     rp <- newTVarIO 0
     wp <- newTVarIO 0
     d <- newTVarIO (if b then ToC else ToP)
     return (BDQ d rp wp (listArray (0,i-1) q))

instance DiQueue BDiQueue where
  safeReadC = withDirB ToC
    (\rp _ q -> do i <- readTVar rp
                   writeTVar rp (i+1)
                   takeTMVar (unsafeAt q i))
  safeReadP = withDirB ToP
    (\rp _ q -> do i <- readTVar rp
                   writeTVar rp (i+1)
                   takeTMVar (unsafeAt q i))
  safeWriteC qr x = withDirB ToP
    (\_ wp q -> do i <- readTVar wp
                   writeTVar wp (i+1)
                   putTMVar (unsafeAt q i) x) qr
  safeWriteP qr x = withDirB ToC
    (\_ wp q -> do i <- readTVar wp
                        writeTVar wp (i+1)
                        putTMVar (unsafeAt q i) x) qr
  waitToC = withDirB ToC (\_ _ _ -> return ())
  waitToP = withDirB ToP (\_ _ _ -> return ())
  swapDir qr = do modifyTVar' (dirB qr) invertDir
                  writeTVar (readPos qr) 0
                  writeTVar (writePos qr) 0
```

Lastly, we define a type class instance of `DiQueue`, `ExtDiQueue`, that existentially quantifies over both of our other instances. This allows us to store either type of `DiQueue` in an `ExecState`, which is needed since a process may make use of both bounded an unbounded channels. Of the three instances, this is the most boring:

```
data ExtDiQueue a where
  ExtDiQueue :: DiQueue q => q a -> ExtDiQueue a

instance DiQueue ExtDiQueue where
  safeReadC (ExtDiQueue q) = safeReadC q
  safeReadP (ExtDiQueue q) = safeReadP q
  safeWriteC (ExtDiQueue q) x = safeWriteC q x
```

```
forward n = IState (\ (ExecState env self) ->
  if polarity
  then myWaitToReadC (index (fromSing n) env) P.>>
       readIORef (index (fromSing n) env) P.>>= \ q ->
       myWriteP self (CForward q) P.>>
       P.return undefined
  else myWaitToReadP self P.>>
       readIORef self P.>>= \ q ->
       myWriteC (index (fromSing n) env) (CForward q)
       P.>> P.return undefined)
  where polarity :: Bool
        polarity = fromSing (sing :: SBool (IsPos s))
```

Figure 5.5: Indexed Monadic Forwarding

```
  safeWriteP (ExtDiQueue q) x = safeWriteP q x
  waitToC (ExtDiQueue q) = waitToC q
  waitToP (ExtDiQueue q) = waitToP q
  swapDir (ExtDiQueue q) = swapDir q
```

Since any newly created channel can either a positive or negative type we need some mechanism to allow us to pass information from the session types known at the type level to the data level. A similar issue arise when forwarding, we need to send the forwarding message in the correct direction. The `singletons` package provides a solution to this: it can generate singleton values at a specified type. To see this in action, let us consider the forwarding operation and, in particular, its `polarity` local binding. First we show its type signature:

```
forward :: forall env n s t.
           (Inbounds n env ~ True
           ,AllButNothing n env ~ True
           ,Index n env ~ Just t
           ,RTEq s t ~ True
           ,SingI (IsPos s))
        => SNat n -> IState (Running env s) Term ()
```

The type constraints on forwarding are fairly standard, `InBounds` checks whether the provided channel is defined, `AllButNothing` checks that only the forwarded from channel is usable, `Index` returns the type of the forwarded channel, `RTEq` performs a circular coinductive check for session type equality, and `SingI` is an ignorable quirk of the `singletons` library. Since forwarding is the last instruction in a process the output type index is `Term`.

The definition of `forward` (Figure 5.5) makes use of a few new functions:

- `myWaitToReadC`, `myWaitToReadP`, `myWriteC`, and `myWriteP` all wrap the `DiQueue` type class to preform an read of the `IORef` holding the relevant queue and then perform the transactional operation specified

- `index` takes a natural number and a list and returns the value in the specified position
- `fromSing` takes a singleton and converts it to its corresponding basic value

Focusing on the `polarity` binding, this says that if the type level `IsPos` function (which returns whether a given `Session` is positive) returns `true` on the channel the process is forwarding from, we create the singleton `SBool True` value and then immediately convert that to a normal `Bool`, `True`. Similarly, if we forward from a negative type, `polarity` will be `False`. In the remainder of the definition we then case on the `polarity` and wait for both queues to point in the same direction before sending a forwarding message to the queue waiting to be written to.

To actually make use of our, hopefully more efficient, bounded queues the definition for the $\{\}E$ operation has an additional local binding `bounds` that uses a type level implementation of Definition 9 to determine what if any bounds the type of channel guarantees:

```
where polarity :: Bool
      polarity = fromSing (sing :: SBool (IsPos t))
      bounds :: Maybe Nat
      bounds = fromSing (sing :: SMaybe (GlobalBounds t))
```

The rest of the operation defintion is a bit long, so we omit it, but this local binding demonstrates how easy the combination of type families and `singletons` make performing even relatively complicated type-directed optimizations.

5.6.2 Prime Sieve Example

Before consider an example we need the ASCII names of each of our type rules:

Rule	ASCII	Rule	ASCII
1R	`close`	1L	`wait`
$\wedge R$	`senddr`	$\wedge L$	`recvdl`
$\supset R$	`recvdr`	$\supset L$	`senddl`
$\otimes R$	`sendcr`	$\otimes L$	`recvcl`
$\multimap R$	`recvcr`	$\multimap L$	`sendcl`
$\oplus R_1$	`intchoir1`	$\oplus L$	`extchoil`
$\oplus R_2$	`intchoir2`		
$\& R$	`extchoir`	$\& L_1$	`intchoil1`
		$\& L_2$	`intchoil2`
$\uparrow R$	`recvsr`	$\uparrow L$	`sendsl`
$\downarrow R$	`sendsr`	$\downarrow L$	`recvsl`
$\{\}E$	`cut`	$\textsc{Id}$	`forward`
$\{\}E + \textsc{Id}$	`tailcut`		

The primes example (subsection 4.1.7) then is a straightforward translation into these indexed monadic operations (passing arguments to cut as a singleton list):

```
type Stream a = (Mu (External (RecvShift One)
     (RecvShift (SendD a (SendShift Var)))))

printstream :: (Show a) => Nat
            -> Process '[Just (Stream a)] One
printstream Z c = do intchoil1 c
                     sendsl c
                     wait c
                     close
printstream (S n) c =
  do intchoil2 c
     sendsl c
     x <- recvdl c
     recvsl c
     liftIO $ putStrLn ("Got "++show x)
     b <- cut (printstream n) (SCons c SNil)
     forward b


countup :: Nat -> Process '[] (Stream Nat)
countup n = extchoir
              (do recvsr
                  close)
              (do recvsr
                  senddr n
                  sendsr
                  a <- cut (countup (S n)) SNil
                  forward a)

silter :: (a -> Bool)
       -> Process '[Just (Stream a)] (Stream a)
silter f c = extchoir
               (do recvsr
                   intchoil1 c
                   sendsl c
                   wait c
                   close)
               (do recvsr
                   intchoil2 c
                   sendsl c
                   x <- recvdl c
                   recvsl c
                   b <- cut (silter f) (SCons c SNil)
                   case f x of
                     True -> do senddr x
                                sendsr
                                forward b
                     False -> do intchoil2 b
                                 sendsl b
                                 forward b)

natsub :: Nat -> Nat -> Nat
natsub n Z = n
natsub (S n) (S m) = natsub n m

divisible :: Nat -> Nat -> Bool
divisible n Z = True
divisible n m = (n <= m) && (divisible n (natsub m n))

sieve :: Process '[Just (Stream Nat)] (Stream Nat)
sieve c = extchoir
          (do intchoil1 c
              sendsl c
              wait c
              recvsr
              close)
          (do intchoil2 c
              sendsl c
              x <- recvdl c
              recvsl c
              recvsr
              senddr x
              sendsr
              b <- cut (silter (not . divisible x))
                       (SCons c SNil)
              tailcut sieve (SCons b SNil))

top :: Process '[] One
top = do a <- cut (countup (S (S Z))) SNil
         b <- cut sieve (SCons a SNil)
         c <- cut (printstream (S (S (S (S (S Z))))))
                  (SCons b SNil)
```

```
        wait c
        close
```

5.6.3 Related Work

The most closely related work is the, apparently unmaintained, `sessions` package.[3] This library implements a classical session typing system without the benefit of several more years of advanced type system feature development. In some cases, this means that the library replicates functionality we utilize by hand, e.g., manually creating singleton types. Session types are represented as type level programs, which are incrementally executed by the type checker. Our μ-based representation could be viewed similarly, but seems considerably cleaner. An interesting choice of the library's design is that it has less need for type annotations, but the ones it does use are supplied via ugly type annotations on `undefined` values.

5.7 Idris

Idris [11] is an eager dependently typed language (section 2.5) that attempts to focus on practical programmability, e.g., it features type classes, compilation, and a good type erasure algorithm. This version of SILL, the least feature complete, provided guidance in designing the Haskell implementation (section 5.6). Specifically, it allowed for experimentation in a Haskell-like language without any of the clunkiness present in the various advanced type features used in the Haskell implementation, while occasionally introducing some challenges of its own. Since we have already discussed how the Haskell, this section will primarily focus on the differences between the two implementations. Unfortunately, while the Idris version allowed for a cleaner system in some ways, some limitations of Idris presented the choice of investing much more time submitting patches[4] or abandoning this as a successful prototype that informed the Haskell SILL implementation.

The Idris version of SILL only aims to produce an abstract syntax tree rather than the more pleasant syntax allowed by the indexed monadic approach of section 5.6, but otherwise proceeds along broadly similar lines. This choice is not fundamental, Idris provides most of the functionality of indexed monads, with some generalization enabled by a fully dependently typed language, in its effect system [13]. Additionally, Idris provides a much more general syntactic extension system [12] beyond that provided by Haskell's `RebindableSyntax`, so we might be able to even encode the syntactic sugar for top-level SILL process defintions (subsection 4.1.6).

Session types are represented via regular trees labeled with elements of the

[3] `https://hackage.haskell.org/package/sessions-2008.7.18`
[4] Some of which would not be acceptable to the maintainers.

set $\{\tau\wedge, \tau\subset, \otimes, \multimap, \oplus, \&, \uparrow, \downarrow\}$. Notice that since τ is infinite, this set of letters is not finite, though the set of labels used in any particular regular tree will be. Since we are working in a dependently typed language, we can, and did, write a regular tree library and freely use it at the type level, whereas the Haskell version of type equality is specific to the session type type and only available at the type level. Unfortunately, circular coinduction (section 5.3) requires us to decide equality on all our letters. In Haskell, since we are stuck working at the type level, we can write a non-linear pattern as needed to check for equality of the two letters that use τ. Since Idris requires parametericity of polymorphic functions, enabling "Theorems for Free" [92], this means that we cannot directly check for type equality while performing circular coinduction. Instead, we rely on an approach inspired by DeBruijn indexing. By attaching a list of types as a phantom type parameter to each regular tree we can replace occurrences of types generated by τ with natural number indices, which feature a decidable equality relation, into this list.

Before looking at the code listing for labels we need two types from Idris's standard library: the type `Vect k t` is the type of lists with length `k` and elements of type `t` and the type `Fin k` is the type of natural numbers less than `k`. We define a type of labels using Idris's GADTs-like syntax, where `{v:Vect k Type}` indicates that the kind vector is passed implicitly (i.e., determined by unification):

```
data SType : (Vect k Type) -> Type where
  SOne      : SType v
  SExternal : SType v
  SInternal : SType v
  SSendC    : SType v
  SRecvC    : SType v
  SSendS    : SType v
  SRecvS    : SType v
  SSendD    : {v:Vect k Type} -> Fin k -> SType v
  SRecvD    : {v:Vect k Type} -> Fin k -> SType v
```

With the exception of needing the index into the kind list for `SSendD` and `SRecvD` these all merely state that they are labels and nothing more. To use these labels with the general regular tree library, we additionally provide an arity for each label (where `{v=v}` helps the unifier fill in the implicit argument of `STypeArities`):

```
STypeArities : {v:Vect k Type} -> (SType v) -> Nat
STypeArities (SSendD _) = 1
STypeArities (SRecvD _) = 1
STypeArities SOne = 0
STypeArities SExternal = 2
STypeArities SInternal = 2
STypeArities SSendC = 2
STypeArities SRecvC = 2

SessionType : Vect k Type -> Type
SessionType v = RegularTree (STypeArities {v=v})
```

With this we can define a session typing environment as a list of session types with some channels marked as unavaible. As before (section 5.6), this enables us to store channel references, which are safe to duplicate, in the AST rather than store the linear channels directly:

```
SesEnv : (k:Nat) -> (v:Vect m Type) -> Type
SesEnv k v = Vect k (Maybe (SessionType v))
```

We then can define an AST that takes session environments and a session type for the provided channel with one constructor per rule in our type system (a full listing is available in section B.3):

```
data Process : (SesEnv k v) -> (SessionType v) -> Type where
  Close : {k:Nat} -> Process (replicate k Nothing) One
  ...
```

Two constructors, coresponding to ID and $\{\}E$, are complicated by the need to use type equality and not just local checks (modulo unfolding). While this is easy enough to encode in the constructors' types via a feature that allows default values to be passed to implicit arguments (here, the default value is the proof generated by the type equality algorithm), a flaw in the way bound variables in default arguments are treated means that this proof will only be successfully generated if we manually fill in the algorithms arguments. This makes forwarding exremely verbose and is quite tedious. After encountering and confirming this bug with Idris's developers, we stopped work on this version of SILL and used the lessons learned for the Haskell implementation.

With our AST complete, we then implemented a simple sequential interpreter that manually multiplexed the processes' threads. This is, of course, not the end goal of such a project, but, since Idris's concurrency primitives do not include thread-safe queues, this was an easy first pass (which became the only pass after hitting the default argument bug).

Chapter 6

Conclusion

This thesis described an exploration in concurrent programming language design, focusing on session types. Our aim was to study logically motivated session typing systems which have received increasing research attention recently [16, 28, 72, 89, 94]. The main result of this investigation was SILL, a language that demonstrates the compatibility of a number of interesting and practically important language features with a logical basis.

The main claims of this thesis were the following:

- Polarization provided a natural way to describe a logically based session typing language with asynchronous communication while retaining a semantics that is reasonably implementable. Additionally, polarization gave us a way to smoothly integrate synchronous channels into SILL without needing a semantic extension.

- Polarization and Adjoint Logic combine very cleanly, giving SILL an ability to incorporate a variety of modalities with relatively little work. From a practical perspective, this gave SILL access to persistent processes and garbage collection for processes.

- We explored a trio of loosely related language extensions, and their logical connections, inspired by the above results: bundled message passing to reduce the number of communications performed by processes; racy programs, enabled by a select/epoll-like mechanism; and asynchronous receiving, a generalization of the basic asynchronous semantics.

- We descirbed three different implementations of SILL: a simple but relatively full featured interpreter written in OCaml; a fragment of SILL as an embedded domain specific language in Haskell; and a cleaner version of the same in Idris.

- We showed that Liquid Types and Session Types are compatible. This gives us one notion of a dependently session typed language.

Overall, we contend that our theory, implementation, and examples show that concurrent programming based on logical principles is a promising way to structure programs in the functional context. Compared to traditional languages such as Concurrent Haskell [73] and Concurrent ML [69] where channels are *not* linearly typed, many additional properties of communication can be statically expressed and enforced. Moreover, the presence of internal and external choice combined with forwarding gives rise to new programming patterns that are not easily supported in prior languages. This thesis demonstrates that these new forms of expressions and types are compatible with many features found in modern languages such as type inference, first-class functions, polymorphism, and data abstraction.

6.1 Future Work

Although this work has accomplished much of what it set out to do, there are a number of interesting avenues for further investigation.

Perhaps the most straightforward direction would be to integrate the LiquidPi work with SILL. As in chapter 3, we expect this to work cleanly at the most basic level but to run into difficulties with the ability to arbitrarily unfold types. Additionally, there is some tension between the type inference of dependent types but no inference of basic session types in SILL. Optimistically, this tension is the same as shows up in many dependently typed languages where some arguments to functions can be elided. An alternative approach is to embrace non-inference and incorporate some dependent types directly into the user visible type descriptions. The main attraction of this approach would be that it avoid the unfolding ambiguity problem by forcing users to deal with it.

With the integration of adjoint logic into SILL (section 4.6), a natural question to ask is, "Can we integrate the modal split between data and processes into the adjoint framework?" There are two obvious ways one might approach this problem (assuming they are not equivalent after a deeper investigation). The first is to note that process reification in SILL looks like staged computations, so trying to integrate a logical treatment [27] of that into SILL may be successful. Alternatively, one could try to directly account for data's behavior by creating a new mode above $\textsc{u}$, that allows the use of a logical accounting of the functional language. The rule $\{\}I$, should then become an instance of $\uparrow R$. Similarly, we should hope that $\{\}E$ becomes $\uparrow L$. Unfortunately, this naive arrangement presents some problems. Since $\uparrow R$ and $\uparrow L$ do not have a notion of argument channels, $\{\}E$ needs to be broken into multiple steps: first initializing the process and then passing its argument channels in one-by-one (essentially, all processes would be given in a Curried form). In theory, this is no worse than splitting $!$ (and can be hidden via syntactic sugar) but needs practical confirmation. In particular, this makes the tail-bind optimization less obvious. Optimistically, this might motivate a more general optimization for processes providing a channel expecting to receive multiple channels (e.g., a variant of the focusing of section 4.2).

While our existing test cases, and the ability to actual execute them, were critical for guiding development of various features, they are still not as extensive as one might hope. In particular, while we have demonstrated the utility of select, it is unclear if larger examples will require a more complete logical incorporation of accept/request and the ability to create cycles of processes. Similarly, subtyping is another language feature that needs more testing, given that our original use case, permissions, worked less well than hoped. Lastly, the theorems could be tested more thoroughly by incorporating them into a theorem prover, with some inspiration from the dependently typed Idris implementation.

References

[1] The coq proof assistant reference manual, 2009.

[2] Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory, 2008.

[3] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[5] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. Technical Report 62, Digital Equipment Corporation, Systems Research Centre, January 1990.

[6] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[7] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions.

[8] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In Prasad Naldurg and Nikhil Swamy, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, pages 15–26. ACM, 2013.

[9] David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4790 of *Lecture Notes in Computer Science*, pages 92–106. Springer Berlin Heidelberg, 2007.

[10] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, August 1977.

[11] Edwin Brady. Idris: general purpose programming with dependent types. In Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard, editors, *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 1–2. ACM, 2013.

[12] Edwin Brady. The idris programming language - implementing embedded domain specific languages with dependent types. In Viktória Zsók, Zoltán Horváth, and Lehel Csató, editors, *Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, volume 8606 of *Lecture Notes in Computer Science*, pages 115–186. Springer, 2013.

[13] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144. ACM, 2013.

[14] Torben Braüner and Valeria de Paiva. Intuitionistic hybrid logic. *J. Applied Logic*, 4(3):231–255, 2006.

[15] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Logic-based domain-aware session types. Technical report, Carnegie Mellon University, 2014.

[16] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2013. To appear. Special Issue on Behavioural Types.

[17] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Inf. Comput.*, 92(1):48–80, May 1991.

[18] Iliano Cervesato, Joshuas. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, pages 67–81. Springer-Verlag LNAI, 1996.

[19] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

[20] Francesco Cesarini and Simon Thompson. *ERLANG Programming.* O'Reilly Media, Inc., 1st edition, 2009.

[21] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. Associated types with class. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 1–13. ACM, 2005.

[22] Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. A judgmental analysis of linear logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University, Department of Computer Science, December 2003.

[23] Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *J. Autom. Reasoning*, 40(2-3):133–177, 2008.

[24] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[25] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Reference counting as a computational interpretation of linear logic. *J. Funct. Program.*, 6(2):195–244, 1996.

[26] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In Richard A. DeMillo, editor, *POPL*, pages 207–212. ACM Press, 1982.

[27] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[28] Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. Cut reduction in linear logic as asynchronous session-typed communication. In P. Cégielski and A. Durand, editors, *Proceedings of the 21st Conference on Computer Science Logic*, CSL 2012, pages 228–242, Fontainebleau, France, September 2012. Leibniz International Proceedings in Informatics.

[29] Anthony Discolo, Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. Lock free data structures using STM in haskell. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2006.

[30] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 429–442, New York, NY, USA, 2013. ACM.

[31] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM.

[32] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56âĂŞ66, 1988.

[33] Dov M Gabbay and Ruy JGB De Queiroz. Extending the curry-howard interpretation to linear, relevant and other resource logics. *Journal of Symbolic Logic*, pages 1319–1365, 1992.

[34] Deepak Garg and Frank Pfenning. Stateful authorization logic - proof theory and a case study. *Journal of Computer Security*, 20(4):353–391, 2012.

[35] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.

[36] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, January 2010.

[37] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[38] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[39] Matthew Hennessy. *A distributed Pi-calculus*. Cambridge University Press, 2007.

[40] Joshua Hodas. Logic programming with multiple context management schemes. In *Fourth International Workshop on Extensions of Logic Programming*, page 360. Springer-Verlag LNCS, 1993.

[41] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[42] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, 2008.

[43] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[44] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping, 1997.

[45] Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In John H. Reppy and Julia L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 50–61. ACM, 2006.

[46] Dimitrios Kouzapas, Ramūnas Gutkovas, and Simon J Gay. Session types for broadcasting. *arXiv preprint arXiv:1406.3481*, 2014.

[47] Joachim Lambek. The mathematics of sentence structure. *Americal Mathematical Monthly*, 65:154–170, 1958.

[48] Olivier Laurent. Polarized proof-nets: Proof-nets for LC. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999)*, pages 213–227, L'Aquila, Italy, April 1999. Springer LNCS 1581.

[49] Olivier Laurent. A proof of the focalization property of linear logic, 2004.

[50] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In Jacques Duparc and ThomasA. Henzinger, editors, *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 451–465. Springer Berlin Heidelberg, 2007.

[51] Jean marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347, 1992.

[52] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K Aswad, and Phil Trinder. Seq no more: better strategies for parallel haskell. In *ACM Sigplan Notices*, volume 45, pages 91–102. ACM, 2010.

[53] Per Martin-Lof. Intuitionistic type theory, 1984.

[54] Nicholas D. Matsakis and Felix S. Klock, II. The rust language. *Ada Lett.*, 34(3):103–104, October 2014.

[55] Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

[56] Conor Mcbride. Kleisli arrows of outrageous fortune, 2011.

[57] Daniel S. McFarlin, Charles Tucker, and Craig B. Zilles. Discerning the dominant out-of-order performance advantage: is it speculation or dynamism? In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 241–252. ACM, 2013.

[58] Sean McLaughlin and Frank Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 174–181. Springer, 2008.

[59] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical computer science*, 25(3):267–310, 1983.

[60] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[61] Dimitris Mostrous and VascoThudichum Vasconcelos. Affine sessions. In Eva KÃijhn and Rosario Pugliese, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 115–130. Springer Berlin Heidelberg, 2014.

[62] Dimitris Mostrous and Nobuko Yoshida. Two session typing systems for higher-order mobile processes. In SimonaRonchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 321–335. Springer Berlin Heidelberg, 2007.

[63] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[64] Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryftis. Safe parallel programming with session java. In *Proceedings of the 13th International Conference on Coordination Models and Languages*, COORDINATION'11, pages 110–126, Berlin, Heidelberg, 2011. Springer-Verlag.

[65] Vivek Nigam and Dale Miller. Algorithmic specifications in linear logic with subexponentials. In António Porto and Francisco Javier López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, pages 129–140. ACM, 2009.

[66] Ulf Norell. Dependently typed programming in agda. In *In Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[67] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[68] C-HL Ong and Charles A Stewart. A curry-howard foundation for functional computation with control. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 215–227. ACM, 1997.

[69] Prakash Panangaden and John Reppy. The essence of concurrent ml. In Flemming Nielson, editor, *ML with Concurrency*, chapter 1. Springer-Verlag, 1997.

[70] Francesco Paoli. *Substructural logics : a primer.* Trends in logic. Kluwer Academic, cop. 2002 (impr. aux Pays-Bas), Dordrecht, Boston (Mass.), 2002.

[71] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, LICS '98, pages 176–, Washington, DC, USA, 1998. IEEE Computer Society.

[72] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 539–558. Springer, 2012.

[73] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

[74] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, January 2007.

[75] Frank Pfenning and Dennis Griffith. Polarized substructural session types. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2015.

[76] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

[77] Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, 2009.

[78] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[79] John H Reppy. Concurrent ml: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. Springer, 1993.

[80] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[81] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.

[82] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 131–144. ACM, 2010.

[83] Andrew William Roscoe. *A mathematical theory of communicating processes*. PhD thesis, University of Oxford, 1982.

[84] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (NAL): design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 14(1):8, 2011.

[85] Tatsurou Sekiguchi and Akinori Yonezawa. A complete type inference system for subtyped recursive types. In Masami Hagiya and JohnC. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 667–686. Springer Berlin Heidelberg, 1994.

[86] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[87] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.

[88] Nikhil Swamy, Juan Chen, Cedric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. In *The 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*. ACM SIGPLAN, September 2011.

[89] Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and New University of Lisbon, 2015. In preparation.

[90] Bernardo Toninho, Luis Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 350–369, Berlin, Heidelberg, 2013. Springer-Verlag.

[91] Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *Proceedings of the 9th International Symposium on Trustworthy Global Computing (TGC 2014)*, Rome, Italy, September 2014. To appear.

[92] Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

[93] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[94] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014.

[95] Philip Wadler. Propositions as types, 2014.

[96] E.F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation execution in distributed systems. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 253–259, May 1990.

[97] Michael Winikoff and James Harland. Deterministic resource management for the linear logic programming language lygon. Technical report, The University of Melbourne, 1994.

[98] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In Benjamin C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 53–66. ACM, 2012.

[99] Nobuko Yoshida and Vasco Thudichum Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Appendix A

Supplementary Listings

A.1 SILL Logic

$$\frac{\Delta; \Psi; \vec{A} \vdash A}{\Delta; \Psi \vdash \{A \leftarrow \vec{A}\}}\ \{\}I \qquad \frac{\vec{A} \geq m \geq r \quad \Delta; \Psi \vdash \{C_m \leftarrow \vec{D}\} \quad \vec{A} \sqsubseteq \vec{D} \quad \Delta; \Psi; \Gamma, C_m \vdash B_r}{\Delta; \Psi; \vec{A}, \Gamma \vdash B_r}\ \{\}E$$

$$\frac{\Gamma \geq \textsc{f} \quad A \sqsubseteq B}{\Delta; \Psi; \Gamma, A \vdash B}\ \textsc{Id} \qquad \frac{\Gamma \geq \textsc{f}}{\Delta; \Psi; \Gamma \vdash 1}\ 1R \qquad \frac{\Delta; \Psi; \Gamma \vdash A}{\Delta; \Psi; \Gamma, 1 \vdash A}\ 1L$$

$$\frac{\Delta; \Psi; \Gamma \vdash A^+ \quad \Delta; \Psi; \Gamma' \vdash B^+}{\Delta; \Psi; \Gamma, \Gamma' \vdash A^+ \otimes B^+}\ \otimes R \qquad \frac{\Delta; \Psi; \Gamma, A^+, B^+ \vdash C}{\Delta; \Psi; \Gamma, A^+ \otimes B^+ \vdash C}\ \otimes L$$

$$\frac{\Delta; \Psi; \Gamma, A^+ \vdash B^-}{\Delta; \Psi; \Gamma \vdash A^+ \multimap B^-}\ \multimap R \qquad \frac{\Delta; \Psi; \Gamma \vdash A^+ \quad \Delta; \Psi; \Gamma', B^- \vdash C}{\Delta; \Psi; \Gamma, \Gamma', A^+ \multimap B^- \vdash C}\ \multimap L$$

$$\frac{k \in I \quad \Delta; \Psi; \Gamma \vdash (A^+_m)_k}{\Delta; \Psi; \Gamma \vdash \oplus_m\{L_i : (A^+_m)_i\}_I}\ \oplus R_{L_k} \qquad \frac{\text{for all } i \in I\text{: } \Delta; \Psi; \Gamma, (A^+_m)_i \vdash B}{\Delta; \Psi; \Gamma, \oplus_m\{L_i : (A^+_m)_i\}_I \vdash B}\ \oplus L$$

$$\frac{\text{for all } i \in I\text{: } \Delta; \Psi; \Gamma \vdash (A^-_m)_i}{\Delta; \Psi; \Gamma \vdash \&_m\{L_i : (A^-_m)_i\}_I}\ \& R \qquad \frac{k \in I \quad \Delta; \Psi; \Gamma, (A^-_m)_k \vdash B}{\Delta; \Psi; \Gamma, \&_m\{L_i : (A^-_m)_i\}_I \vdash B}\ \& L_{L_k}$$

$$\frac{\Delta; \Psi; \Gamma \vdash A^+_k}{\Delta; \Psi; \Gamma \vdash \uparrow^m_k A^+_k}\ \uparrow R \qquad \frac{k \geq r \quad \Delta; \Psi; \Gamma, A^+_k \vdash C_r}{\Delta; \Psi; \Gamma, \uparrow^m_k A^+_k \vdash C_r}\ \uparrow L$$

$$\frac{\Gamma \geq m \quad \Delta; \Psi; \Gamma \vdash A^-_m}{\Delta; \Psi; \Gamma \vdash \downarrow^m_k A^-_m}\ \downarrow R \qquad \frac{\Delta; \Psi; \Gamma, A^-_m \vdash C_r}{\Delta; \Psi; \Gamma, \downarrow^m_k A^-_m \vdash C_r}\ \downarrow L$$

$$\frac{\Delta; \Psi \vdash \tau \quad \Delta; \Psi, \Gamma \vdash A^+}{\Delta; \Psi, \Gamma \vdash \tau \wedge A^+}\ \wedge R \qquad \frac{\Delta; \Psi, \tau; \Gamma, A^+ \vdash B}{\Delta; \Psi; \Gamma, \tau \wedge A^+ \vdash B}\ \wedge L$$

$$\frac{\Delta; \Psi, \tau; \Gamma \vdash A^-}{\Delta; \Psi; \Gamma \vdash \tau \supset A^-}\ \supset R \qquad \frac{\Delta; \Psi \vdash \tau \quad \Delta; \Psi; \Gamma, A^- \vdash B}{\Delta; \Psi; \Gamma, \tau \supset A^- \vdash B}\ \supset L$$

$$\frac{\Delta \vdash C \quad \Delta; \Psi; \Gamma \vdash A^+[C/\alpha]}{\Delta; \Psi; \Gamma \vdash \exists \alpha . A^+}\ \exists R \qquad \frac{\beta \notin (\Delta, \alpha) \quad \Delta; \Psi; \Gamma, A^+[\beta/\alpha] \vdash B}{\Delta; \Psi; \Gamma, \exists \alpha . A^+ \vdash B}\ \exists L$$

$$\frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma \vdash A^-[\beta/\alpha]}{\Delta; \Psi; \Gamma \vdash \forall \alpha . A^-}\ \forall R \qquad \frac{\Delta \vdash C \quad \Delta; \Psi; \Gamma, A^-[C/\alpha] \vdash B}{\Delta; \Psi; \Gamma, \forall \alpha . A^- \vdash B}\ \forall L$$

All judgments $\Delta; \Psi; \Gamma \vdash A_m$ presuppose $\Gamma \geq m$.

A.2 SILL Type System

$$\frac{\Delta;\Psi;\vec{a}:\vec{A}\rhd\Omega\vdash P::a:A\quad \Omega\geq\textsc{s}}{\Delta;\Psi\vdash a\leftarrow\{P\}\mathrel{-\!\!\prec}\vec{a}:A\leftarrow\vec{A}}\;\{\}I$$

$$\frac{\begin{array}{c}\vec{A}\geq m\geq r\quad \Delta;\Psi\vdash e:\{B_m\leftarrow\vec{D}\}\\ \vec{A}\sqsubseteq\vec{D}\quad \Delta;\Psi;\Gamma,b:B_m\rhd\Omega,b_m:\star\vdash P::c_r:C_r\end{array}}{\Delta;\Psi;\Gamma,\vec{a}:\vec{A}\rhd\Omega,\vec{a}:\vec{\textsc{c}}\vdash b_m\leftarrow e\mathrel{-\!\!\prec}\vec{a};P::c_r:C_r}\;\{\}E$$

$$\frac{A_m\sqsubseteq C_m}{\Delta;\Psi;\Gamma,a_m:A_m\rhd a_m:\textsc{c},(\mathrm{dom}(\Gamma|_{\textsc{l}}):\vec{\textsc{n}}),(\mathrm{dom}(\Gamma|_{\geq\textsc{f}}):\vec{\textsc{s}})\vdash c_m\leftarrow a_m::c_m:C_m}\;\textsc{Id}$$

$$\frac{}{\Delta;\Psi;\Gamma\rhd(\mathrm{dom}(\Gamma|_{\textsc{l}}):\vec{\textsc{n}}),(\mathrm{dom}(\Gamma|_{\geq\textsc{f}}):\vec{\textsc{s}})\vdash\mathsf{close}\ c_m::c_m:1_m}\;1R$$

$$\frac{\Delta;\Psi;\Gamma\rhd\Omega P::c_m:C_m}{\Delta;\Psi;\Gamma\rhd\Omega,a_k:\textsc{c}\vdash\mathsf{wait}\ a_k;P::c_m:C_m}\;1L$$

$$\frac{\Delta;\Psi\vdash e:\tau\quad \Delta;\Psi;\Gamma\rhd\Omega\vdash P::c_m:C_m^+}{\Delta;\Psi;\Gamma\rhd\Omega\vdash\mathsf{send}\ c_m\ e;P::c_m:\tau\wedge C_m^+}\;\wedge R$$

$$\frac{\Delta;\Psi,x:\tau;\Gamma,a_k:A_k^+\rhd\Omega,a_k:\star\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma,a_k:\tau\wedge A_k^+\rhd\Omega,a_k:\textsc{c}\vdash x\leftarrow\mathsf{recv}\ a_k;P::c_m:C_m}\;\wedge L$$

$$\frac{\Delta;\Psi,x:\tau;\Gamma\rhd\Omega\vdash P::c_m:\tau\supset C_m^-}{\Delta;\Psi;\Gamma\rhd\Omega\vdash x\leftarrow\mathsf{recv}\ c_m;P::c_m:\tau\supset C_m^-}\;\supset R$$

$$\frac{\Delta;\Psi\vdash e:\tau\quad \Delta;\Psi;\Gamma,b_k:B_k^-\rhd\Omega,b_k:\star\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma,b_k:\tau\supset B_k^-\rhd\Omega,b_k:\textsc{c}\vdash\mathsf{send}\ b_k\ e;P::c_m:C_m}\;\supset L$$

$$\frac{\Delta;\Psi;\Gamma\rhd\Omega_1\vdash P::a_m:A_m^+\quad \Delta;\Psi;\Gamma\rhd\Omega_2\vdash Q::c_m:C_m^+}{\Delta;\Psi;\Gamma\rhd\Omega_1\bowtie\Omega_2\vdash\mathsf{send}\ c_m\ (a_m\leftarrow P);Q::c_m:A_m^+\otimes C_m^+}\;\otimes R$$

$$\frac{\Delta;\Psi;\Gamma,a_k:A_k^+,b_k:B_k^+\rhd\Omega,a_k:\star,b_k:\star\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma,b_k:A_k^+\otimes B_k^+\rhd\Omega,b_k:\textsc{c}\vdash a_k\leftarrow\mathsf{recv}\ b_k;P::c_m:C_m}\;\otimes L$$

$$\frac{k\in I\quad \Delta;\Psi;\Gamma\rhd\Omega\vdash P::c:(A_m^+)_k}{\Delta;\Psi;\Gamma\rhd\Omega\vdash\mathsf{send}\ c\ L_k;P::c:\oplus_m\{L_i:(A_m^+)_i\}_I}\;\oplus R_{L_k}$$

$$\frac{I\subseteq J\quad\text{for all }k\in I:\ \Delta;\Psi;\Gamma,a_m:(A_m^+)_k\rhd\Omega_k,a_m:\star\vdash P_k::c:C_r\text{ and }\Omega_k\sqsubseteq\Omega}{\Delta;\Psi;\Gamma,a_m:\oplus_m\{L_i:(A_m^+)_i\}_I\rhd\Omega,a_m:\textsc{c}\vdash\begin{pmatrix}\mathsf{case}_J\ a_m\ \mathsf{of}\\ L_j\to P_j\end{pmatrix}::c_r:C_r}\;\oplus L$$

$$\frac{I\subseteq J\quad\text{for all }k\in I:\ \Delta;\Psi;\Gamma\rhd\Omega_k\vdash P_k::c:(C_m^-)_k\text{ and }\Omega_k\sqsubseteq\Omega}{\Delta;\Psi;\Gamma\rhd\Omega\vdash\begin{pmatrix}\mathsf{case}_J\ c_m\ \mathsf{of}\\ L_j\to P_j\end{pmatrix}::c_m:\&_m\{c_i:(C_m^-)_i\}_I}\;\&R$$

$$\frac{k\in I\quad \Delta;\Psi;\Gamma,a_m:(A_m^-)_k\rhd\Omega,a_m:\star\vdash P::c_r:C_r}{\Delta;\Psi;\Gamma,a_m:\&\{L_i:(A_m^-)_i\}_I\rhd\Omega,a_m:\textsc{c}\vdash\mathsf{send}\ c_r\ L_k;P::c_r:C_r}\;\&L_{L_k}$$

$$\frac{\Delta\vdash A\quad \Delta;\Psi;\Gamma\rhd\Omega\vdash P::c_m:C_m^+[A/\alpha]}{\Delta;\Psi;\Gamma\rhd\Omega\vdash\mathsf{send}\ c_m\ A;P::c_m:\exists\alpha.C_m^+}\;\exists R$$

$$\frac{\beta\notin(\Delta,\alpha)\quad \Delta,\beta;\Psi;\Gamma,a_k:A_k^+[\beta/\alpha]\rhd\Omega,a_k:\textsc{c}\vdash P::c_m:C_m}{\Delta;\Psi;\Gamma,a_k:\exists\alpha.A_k^+\rhd\Omega,a_k:\star\vdash\alpha\leftarrow\mathsf{recv}\ a_k;P::c_m:C_m}\;\exists L$$

$$\frac{\beta \notin (\Delta, \alpha) \quad \Delta, \beta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m^-[\beta/\alpha]}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \alpha \leftarrow \mathsf{recv}\ c_m; P :: c_m : \forall \alpha. C_m^-} \ \forall R$$

$$\frac{\Delta \vdash A \quad \Delta; \Psi; \Gamma, b_k : B_k^- \triangleright \Omega, b_k : \star \vdash P :: c_m : C_m}{\Delta; \Psi; \Gamma, b_k : \forall \alpha. B_k^- \triangleright \Omega, b_k : \mathrm{C} \vdash \mathsf{send}\ b_k\ A; P :: c_m : C_m} \ \forall L$$

$$\frac{\Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: a_k : A_k^+}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \mathsf{shift}\ a_k \leftarrow \mathsf{recv}\ b_m; P :: b_m : \uparrow_k^m A_k^+} \ \uparrow R$$

$$\frac{k \geq r \quad \Delta; \Psi; \Gamma, a_k : A_k^+ \triangleright \Omega, a_k : \star \vdash Q :: c_r : C_r}{\Delta; \Psi; \Gamma, b_m : \uparrow_k^m A_k^+ \triangleright \Omega, b_m : \mathrm{C} \vdash \mathsf{shift}\ a_k \leftarrow \mathsf{send}\ b_m; Q :: c_r : C_r} \ \uparrow L$$

$$\frac{\Gamma \geq m \quad \Delta; \Psi; \Gamma \triangleright \Omega \vdash Q :: a_m : A_m^-}{\Delta; \Psi; \Gamma \triangleright \Omega \vdash \mathsf{shift}\ a_m \leftarrow \mathsf{send}\ b_k; Q :: b_k : \downarrow_k^m A_m^-} \ \downarrow R$$

$$\frac{\Delta; \Psi; \Gamma, a_m : A_m^- \triangleright \Omega, a_m : \star \vdash P :: c_r : C_r}{\Delta; \Psi; \Gamma, b_k : \downarrow_k^m A_m^- \triangleright \Omega, b_k : \mathrm{C} \vdash \mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ b_k; P :: c_r : C_r} \ \downarrow L$$

Presuppposes for $\Delta; \Psi; \Gamma \triangleright \Omega \vdash P :: c_m : C_m$ that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Omega)$ and $\Gamma \geq m$

A.3 SILL Semantics

BIND$_{\mathsf{step}}$: $\mathsf{exec}_c(a \leftarrow e \multimap \vec{a}; P) \otimes !(e \to e') \multimap \mathsf{exec}_c(a \leftarrow e' \multimap \vec{a}; P)$

DATA$_{\mathsf{step}}$: $\mathsf{exec}_c(\mathsf{send}\ b\ e; P) \otimes !(e \to e') \multimap \mathsf{exec}_c(\mathsf{send}\ b\ e'; P)$

SEND$_{\mathsf{data}}$: $\mathsf{exec}_c(\mathsf{send}\ b\ v;\ P) \otimes \mathsf{que}(a, M, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M\ v, b)$

RECV$_{\mathsf{data}}$: $\mathsf{exec}_c(x \leftarrow \mathsf{recv}\ a; P) \otimes \mathsf{que}(a, v\ M, b)$
$\multimap \mathsf{exec}_c(P[v/x]) \otimes \mathsf{que}(a, M, b)$

SEND$_{\mathsf{end}}$: $\mathsf{exec}_c(\mathsf{close}\ c) \otimes \mathsf{que}(a, M, c) \multimap \mathsf{que}(a, M\ \mathsf{end}, c)$

RECV$_{\mathsf{end}}$: $\mathsf{exec}_c(\mathsf{wait}\ a; P) \otimes \mathsf{que}(a, \mathsf{end}, b) \multimap \mathsf{exec}_c(P)$

SEND$_{\mathsf{chan}}$: $\mathsf{exec}_c(\mathsf{send}\ b\ (d \leftarrow P); Q) \otimes \mathsf{que}(a, M, b)$
$\multimap \exists f, g. \mathsf{exec}_c(Q) \otimes \mathsf{que}(a, M\ g, b) \otimes \mathsf{exec}_f(P[f/d]) \otimes \mathsf{que}(g, \cdot, f)$

RECV$_{\mathsf{chan}}$: $\mathsf{exec}_c(d \leftarrow \mathsf{recv}\ a; P) \otimes \mathsf{que}(a, f\ M, b)$
$\multimap \mathsf{exec}_c(P[f/d]) \otimes \mathsf{que}(a, M, b)$

SEND$_{\mathsf{choice}}$: $\mathsf{exec}_c(\mathsf{send}\ b\ L_k; P) \otimes \mathsf{que}(a, M, b) \multimap \mathsf{exec}_c(P) \otimes \mathsf{que}(a, M\ L_k, b)$

RECV$_{\mathsf{choice}}$: $\mathsf{exec}_c\begin{pmatrix}\mathsf{case}_J\ a\ \mathsf{of} \\ L_j \to P_j\end{pmatrix} \otimes \mathsf{que}(a, L_k\ M, b) \multimap \mathsf{exec}_c(P_k) \otimes \mathsf{que}(a, M, b)$

SEND$_{\mathsf{shift}}$: $\mathsf{que}(a_k, M, c_k) \otimes \mathsf{exec}_{c_k}(\mathsf{shift}\ b_m \leftarrow \mathsf{send}\ c_k; P)$
$\multimap \exists d_m, f_m. \mathsf{que}(a_k, M\ \mathsf{shift}(f_m), d_m) \otimes \mathsf{exec}(P[d_m/b_m])$

RECV$_{\mathsf{shift}}$: $\mathsf{exec}_{c_k}(\mathsf{shift}\ a_m \leftarrow \mathsf{recv}\ c_k; P) \otimes \mathsf{que}(c_k, \mathsf{shift}(b_m), d_m)$
$\multimap \mathsf{que}(d_m, \cdot, b_m) \otimes \mathsf{exec}_{b_m}(P[b_m/a_m])$

SEND$_{\mathsf{type}}$: $\mathsf{que}(a, M, c) \otimes \mathsf{exec}_c(\mathsf{send}\ c\ A; P) \multimap \mathsf{que}(a, M\ A, c) \otimes \mathsf{exec}_c(P)$

RECV$_{\mathsf{type}}$: $\mathsf{exec}_c(\alpha \leftarrow \mathsf{recv}\ c; P) \otimes \mathsf{que}(c, A\ M, d)$
$\multimap \mathsf{exec}_c(P[A/\alpha]) \otimes \mathsf{que}(c, M, d)$

A.4 OCaml SILL Syntax

```
lowcase ::= [a-z]([a-zA-Z0-9'_])*
uppcase ::= [A-Z]([a-zA-Z0-9'_])*
```

```
linchan ::= '<lowcase>
affchan ::= @<lowcase>
shrchan ::= !<lowcase>

subchan ::= <linchan> | <affchan>
anychan ::= <linchan> | <affchan> | <shrchan>

data_type
  ::= { <session_type> <- <';'-separated list of <session_type>> }
    | { <- <';'-separated list of <session_type>> }
    | <uppcase> <' '-separated list of <either_type>>
    | <lowcase>
    | ()
    | ( <data_type> )
    | ( <data_type> , <data_type> )
    | [ <data_type> ]
    | <data_type> -> <data_type>

session_type
  ::= ( <session_type> )
    | <uppcase> <' '-separated list of <either_type>>
    | <anychan>
    | 1
    | ' <session_type>
    | @ <session_type>
    | ! <session_type>
    | <data_type> /\ <session_type>
    | <data_type> => <session_type>
    | <session_type> *  <session_type>
    | <session_type> -o <session_type>
    | +{ <';'-separated list of <session_type_mapping>> }
    | &{ <';'-separated list of <session_type_mapping>> }
    | forall <anychan> . <session_type>
    | exists <anychan> . <session_type>

session_type_mapping ::= <lowcase> : <session_type>

either_type ::= <data_type> | <session_type>

any_type_var ::= <lowcase> | <anychan>

var_list ::= <' '-separated list of <any_type_var> >
```

```
ctor ::= uppcase <' '-separated list of <data_type>>

typedec
  ::= type <uppcase> <var_list> = <'|'-separated lists of <ctor>>
    | ltype <uppcase> <var_list> = <session_type>
    | atype <uppcase> <var_list> = <session_type>
    | utype <uppcase> <var_list> = <session_type>


pat_var ::= lowcase
              | _

exp
  ::= ( <exp> )
    | <integer literal>
    | <float literal>
    | <string literal>
    | <built_in>
    | ()
    | [ <';'-separated list of <exp>> ]
    | <lowcase>
    | <uppcase>
    | <exp> <exp>
    | ( <exp> , <exp> )
    | <exp> || <exp>
    | <exp> && <exp>
    | <exp> < <exp>
    | <exp> > <exp>
    | <exp> <= <exp>
    | <exp> >= <exp>
    | <exp> = <exp>
    | <exp> :: <exp>
    | <exp> + <exp>
    | <exp> - <exp>
    | <exp> +. <exp>
    | <exp> -. <exp>
    | <exp> * <exp>
    | <exp> / <exp>
    | <exp> *. <exp>
    | <exp> /. <exp>
    | <exp> ^ <exp>
    | <exp> ** <exp>
    | let <' '-separated list of <pat_var>> : <data_type>
      = <exp> in <exp>
    | fun <' '-separated list of <pat_var>> -> <exp>
    | if <exp> then <exp> else <exp>
    | <lowcase>'<'<','-separated list of <either_type>>'>'
    | case <exp> of <' '-separated list of <case_exp>>
    | <exp> : <data_type>
    | <anychan> <-{ <proc> }
    | <anychan> <-{ <proc> }-< <' '-separated list of <anychan>>
    | _ <-{ <proc> }
    | _ <-{ <proc> }-< <' '-separated list of <anychan>>

case_exp
  ::= '|' <uppcase> <' '-separated list of <pat_var>> -> <exp>
    | '|' [ ] -> <exp>
    | '|' <pat_var> :: <pat_var> -> <exp>
    | '|' ( <pat_var> , <pat_var> ) -> <exp>

case_proc
  ::= '|' <uppcase> <' '-separated list of <pat_var>> -> <proc>
    | '|' [ ] -> <proc>
    | '|' <pat_var> :: <pat_var> -> <proc>
    | '|' ( <pat_var> , <pat_var> ) -> <proc>

built_in ::= assert          /* Bool -> () */
           | sleep           /* Int -> () */
           | print           /* Int -> () */
           | print_str       /* String -> () */
           | flush           /* () -> () */
           | i2s             /* Int -> String */
           | sexp2s          /* a -> String */

branch ::= '|' <lowcase> -> <proc>

proc
 ::= ( <proc> )
   | abort
   | close <subchan>
   | wait <subchan> ; <proc>
   | <anychan> <- <exp; <proc>
   | <anychan> <- <exp -< <' '-separated list of <anychan>> ;
       <proc>
   | _ <- <exp>; <proc>
   | _ <- <exp> -< <' '-separated list of <anychan>>; <proc>
   | send <subchan> <exp> ; <proc>
   | <pat_var> <- recv <subchan> ; <proc>
   | send <subchan> <subchan> ; <proc>
   | send <subchan> ( <subchan> <- <proc> ) ; <proc>
   | <subchan> <- recv <subchan> ; <proc>
   | <subchan> <- send <anychan> ; <proc>
   | send <subchan> ( <anychan> <- <proc> )
   | <subchan> <- <subchan>
   | if <exp> then <proc> else <proc>
   | case <exp> of <' '-separated list of <case_proc>>
   | case <subchan> of <' '-separated list of <branch>>
   | <exp> ; <proc>
   | let <lowcase> <' '-separated list of <pat_var>> =
       <exp> ; <proc>
   | send <subchan> < <session_type> > ; <proc>
   | < <subchan> > <- recv <subchan> ; <proc>
   | send <subchan> shift; <proc>
   | shift <- recv <subchan>; <proc>

ses_quant_list ::= '<' <','-separated list of <anychan>> '>'

topsig ::= <lowcase> : <data_type> ;;
         | forall <ses_quant_list> . <data_type>;;

topdef
  ::= <topsig> <lowcase> <ses_quant_list>
        <' '-separated list of <pat_var>> = <exp>
    | <topsig> <subchan> <- <lowcase> <ses_quant_list>
        <' '-separated list of <pat_var>> = <proc>
    | <topsig> <subchan> <- <lowcase> <ses_quant_list>
        <' '-separated list of <pat_var>> -<
        <' '-separated list of <anychan>> = <proc>

topproc ::= <linchan> <- <proc>

code ::= <typedec> ;;
       | <'and'-separated list of <topproc>> ;;
```

Appendix B

Code Listings

This chapter contains a list of various supporting source code files. All are present in the accompanying tarballs. Since there are three different codebases included, we explicitly list which files go with which project.

B.1 OCaml

For the OCaml implementation of SILL see the following files in `ocaml.tar.gz`:

- `oasis`
- `base.ml`
- `bidir.ml`
- `catqueue.ml`
- `connection.ml`
- `coresyntax.ml`
- `desttypes.ml`
- `desugar.ml`
- `errortester.sh`
- `eval-abstract.ml`
- `eval-exp.ml`
- `eval-functor.ml`
- `eval-pipe.ml`
- `eval-ssh.ml`
- `eval-thread.ml`
- `fullsyntax.ml`
- `newparser.ml`

- `sill.ml`
- `sshstub.ml`
- `subtype.ml`
- `syntax.ml`
- `tester.sh`
- `types.ml`
- `unify.ml`
- `value.ml`
- `vars.ml`
- `vector-clock.ml`
- `async.sill`
- `async.sill.interp`
- `basic-proc.sill`
- `basic-proc.sill.interp`
- `basic-proc.sill.typecheck`
- `basic.sill`
- `basic.sill.interp`
- `basic.sill.typecheck`
- `counter.sill`
- `counter.sill.interp`
- `counter2.sill`
- `counter2.sill.interp`
- `counter2.sill.output`
- `fastfilter.sill`
- `fastfilter.sill.interp`
- `fastfilter.sill.output`
- `hash.sill`
- `hash.sill.interp`

- `mergesort.sill`
- `mergesort.sill.output`
- `oddevensort.sill`
- `oddevensort.sill.interp`
- `parsertests.sill`
- `parsertests.sill.parse`
- `pq.sill`
- `pq.sill.interp`
- `primes.sill`
- `primes.sill.interp`
- `primes.sill.output`
- `primes.sill.typecheck`
- `seg.sill`
- `seg.sill.interp`
- `seg.sill.output`
- `sessionpoly.sill`
- `sessionpoly.sill.interp`
- `sieve2-lazy.sill`
- `sieve2-lazy.sill.output`
- `stack.sill`
- `stack.sill.interp`
- `stack2.sill`
- `stack2.sill.interp`
- `stack2.sill.output`
- `subtyping.sill`
- `subtyping.sill.interp`
- `subtyping10.sill`
- `subtyping10.sill.interp`

- `subtyping2.sill`
- `subtyping2.sill.interp`
- `subtyping3.sill`
- `subtyping3.sill.interp`
- `subtyping4.sill`
- `subtyping4.sill.interp`
- `subtyping5.sill`
- `subtyping5.sill.interp`
- `subtyping6.sill`
- `subtyping6.sill.interp`
- `subtyping7.sill`
- `subtyping7.sill.interp`
- `subtyping8.sill`
- `subtyping8.sill.interp`
- `subtyping9.sill`
- `subtyping9.sill.interp`
- `sugar-subtyping.sill`
- `sugar-subtyping.sill.interp`
- `t1.sill`
- `t1.sill.interp`
- `t1.sill.typecheck`
- `t11.sill`
- `t11.sill.interp`
- `tree.sill`
- `tree.sill.interp`
- `tree.sill.output`

B.2 Haskell

For the Haskell implementation of SILL see the following files in `haskell.tar.gz`:

- `DiQueue.hs`
- `Sessions.hs`
- `Syntax.hs`

B.3 Idris

For the Idris implementation of SILL see the following files in `idris.tar.gz`:

- `RegTrees.idr`
- `RegularTrees.idr`
- `sessions.idr`