Thesis Proposal
Denotational Semantics for Session-Typed Languages

Ryan Kavanagh

Spring 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Stephen Brookes, co-chair
Frank Pfenning, co-chair
Jan Hoffmann
Luís Caires, Universidade Nova de Lisboa
Gordon Plotkin, University of Edinburgh

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

Key words and phrases. Denotational semantics, session types, general recursion, adjoint logic, dependent type theory

Abstract. Session types specify communications protocols for languages with message-passing concurrency. I claim that denotational semantics elucidate the structure of session-typed languages and allow us to reason about programs written in these languages in ways that are complementary to existing approaches. To support this thesis statement, I have developed a domain-theoretic semantics for polarized SILL, a session-typed language that cohesively integrates functional computation and message-passing concurrent computation. I propose expanding this semantics to support the features of dependently-session-typed languages and of computational interpretations of adjoint logic. To establish the complementary nature of these semantics, I propose showing that they are sound and adequate relative to existing substructural operational semantics for these languages. To show that these denotational semantics can tractably be used to reason about programs, I propose using my semantics to validate various non-trivial process equivalences and to verify process correctness.

Contents

List of Figures

CHAPTER 1

Introduction

The proofs-as-programs correspondence between linear logic and the session-typed π-calculus is the foundation of many programming languages [TCP13; Wad14; CP10; TY18a] for message-passing concurrency. Many techniques exist for reasoning about these languages. For example, Pérez et al. [Pér+12; Pér+14] and Toninho [Ton15] introduced logical-relations-based approaches. Gommerstadt, Jia, and Pfenning [GJP18] introduced "monitors" to ensure correctness at runtime. Game semantics [CY19] and denotational semantics [Atk17] have further enriched our understanding of session-typed languages. Very few approaches treat inductive and co-inductive session types [LM16; DP19] or general recursive types. Recent work [Kav20] gave the first denotational semantics to a higher-order session-typed language with general recursion.

Denotational semantics have many applications. At a fundamental level, they help us understand a language and its computational phenomena. More practically, denotational semantics are a tool that lets us reason modularly about programs. This is because denotational semantics are automatically *compositional*. Practical applications of denotational semantics range from compiler correctness [Wan95] to program verification [Pol81]. Denotational semantics also provide a notion of program equivalence.

Given these applications, it seems profitable to explore denotational semantics for languages with session types. These languages include computational interpretations of adjoint logic and session-typed languages with dependent types. Adjoint logic [Ben95; BW96; Ree09; Pru+18] is a framework in which we can uniformly combine multiple logical systems. It has a rich computational interpretation that coherently combines message-passing, shared-memory, and sequential computation [PP19b]. The rich types of dependently-session-typed languages [TCP11; TY18a; TY18b] allow programmers to precisely specify very complex protocols.

These concerns motivate the following thesis statement:

> *Denotational semantics elucidate the structure of session-typed languages and allow us to reason about programs written in these languages in ways that are complementary to existing approaches.*

Below, I further motivate this thesis statement and propose research to defend it. In chapter 2, I give an overview of session types and fix the notation used in the remainder of this proposal. I also introduce concepts and terminology that will be used throughout. Chapters 3 to 6 propose work that should form the body of my dissertation. I motivate each unit of proposed work and give a survey of existing related work. I then flesh out details of my proposed contribution. Chapter 7 gives a tentative timeline for completing each unit of proposed work.

CHAPTER 2

Background and Notation

A process is a computing agent that interacts with its environment through communication. Communication happens over named channels that can be thought of as wires that carry bidirectional communication. To ensure correctness, it is useful to specify the communication that are permitted on a channel. Session types specify these communication protocols [Hon93; THK94].

Session types and linear logic are intimately related [CP10; Wad14]. Indeed, we can give a proofs-as-programs interpretations to sequent calculi for linear logic, where proofs correspond to processes and propositions correspond to session types. Process reductions correspond to cut elimination.

We briefly illustrate this relationship. Consider the intuitionistic linear logic proposition $\mathbf{1}$. It has the following right and left rules:

$$\frac{}{\cdot \vdash \mathbf{1}}\ (\mathbf{1}R) \qquad \frac{\Delta \vdash C}{\Delta, \mathbf{1} \vdash C}\ (\mathbf{1}L)$$

In the proofs-as-processes interpretation, process $a_1 : A_1, \ldots, a_n : A_n \vdash P :: a_o : A_o$ communicates over channels a_i satisfying protocols A_i. We give the above rules the following proofs-as-processes interpretation:

$$\frac{}{\cdot \vdash \mathsf{close}\ a :: a : \mathbf{1}}\ (\mathbf{1}R) \qquad \frac{\Delta \vdash \mathsf{wait}\ a; P :: c : C}{\Delta, a : \mathbf{1} \vdash \mathsf{wait}\ a; P :: c : C}\ (\mathbf{1}L)$$

We interpret the process $\mathsf{close}\ a$ as sending a close message on the channel a and terminating, while the process $\mathsf{wait}\ a; P$ waits for the channel a to close and continues as P. The only possible communication satisfying the session type $\mathbf{1}$ is the close message.

The (Cut) rule of intuitionistic linear logic is interpreted as process composition, where we compose two processes P and Q to communicate on a common channel a:

$$\frac{\Delta_1 \vdash P :: a : A \quad a : A, \Delta_2 \vdash Q :: c : C}{\Delta_1, \Delta_2 \vdash a \leftarrow P;\ Q :: c : C}\ (\textsc{Cut})$$

In a synchronous setting, we can capture the operational intuitions for closing channels using the reduction rule:

$$(a \leftarrow \mathsf{close}\ a;\ (\mathsf{wait}\ a; P)) \longrightarrow P \tag{1}$$

To illustrate how cut-elimination corresponds to process reduction, consider the process:

$$\dfrac{\dfrac{}{\cdot \vdash \mathsf{close}\ a :: a : \mathbf{1}}\ (\mathbf{1}R) \quad \dfrac{\dfrac{}{\cdot \vdash \mathsf{close}\ c :: c : \mathbf{1}}\ (\mathbf{1}R)}{a : \mathbf{1} \vdash \mathsf{wait}\ a; \mathsf{close}\ c :: c : \mathbf{1}}\ (\mathbf{1}L)}{\cdot \vdash a \leftarrow \mathsf{close}\ a;\ \mathsf{wait}\ a; \mathsf{close}\ c :: c : \mathbf{1}}\ (\textsc{Cut})$$

The cut-elimination algorithm [Pfe00] converts this proof to the proof:

$$\frac{}{\cdot \vdash \mathsf{close}\ c :: c : \mathbf{1}}\ (\mathbf{1}R)$$

This exactly mirrors the reduction (1).

We can give proofs-as-processes interpretations to the remainder of intuitionistic and classical linear logic. These interpretations have been extended to support various programming constructs. For example, SILL [TCP13] includes a functional programming layer and general recursion. It will be the subject of section 3.1.

2.1. Categorical and Domain-Theoretic Background and Notation

We frequently use indexed products. We may make their indices explicit to reduce ambiguity. We write $(d_1 : D_1) \times \cdots \times (d_n : D_n)$ for the product of the D_i indexed by the d_i. Given $\delta_i \in D_i$, we write $(d_1 : \delta_1, \ldots, d_n : \delta_n)$ for an element of this product. Given an indexed product $\prod_{i\in I} D_i$ and a subset $J \subseteq I$, we write π^I_J or π_J for the projection $\prod_{i\in I} D_i \to \prod_{j\in J} D_j$.

We write ω-**aBC** for the category of **Scott domains** (ω-algebraic bounded-complete dcpos) and continuous functions. We write ω-$\mathbf{aBC}_{\perp!}$ for the subcategory of strict functions.

The category ω-**aBC** has a continuous **least-fixed-point operator** $\mathsf{fix} : [D \to D] \to D$ for each object D. It also has a continuous fixed-point operator $(\cdot)^\dagger : [A \times X \to X] \to [A \to X]$ given by $f^\dagger(a) = \mathsf{fix}(\lambda x.f(a, x))$. It satisfies the fixed-point identity of [BÉ96]: $f^\dagger = f \circ \langle \mathsf{id}, f^\dagger \rangle$.

The category ω-**aBC** is also equipped with a **trace operator** [AHS02; CŞ90; JSV96]. It fixes the X component of a morphism $f : A \times X \to B \times X$ to produce a morphism $\mathsf{Tr}^X(f) : A \to B$. It is given by $\mathsf{Tr}^X(f) = \pi_B^{B\times X} \circ f \circ \langle \mathsf{id}_A, (\pi_X^{B\times X} \circ f)^\dagger \rangle$. It has the following Knaster-Tarski-style formulation: $\mathsf{Tr}^X(f)(a) = \pi_B(\bigsqcap\{(b, x) \mid f(a, x) \sqsubseteq (b, x)\})$.

The **lifting functors** $(-)_\perp : \omega\text{-}\mathbf{aBC} \to \omega\text{-}\mathbf{aBC}$ and $(-)_\perp : \omega\text{-}\mathbf{aBC}_{\perp!} \to \omega\text{-}\mathbf{aBC}_{\perp!}$ are respectively left-adjoint to the identity functors $\mathsf{id}_{\omega\text{-}\mathbf{aBC}}$ and $\mathsf{id}_{\omega\text{-}\mathbf{aBC}_{\perp!}}$. The units $\mathsf{id} \to (-)_\perp$ are respectively called up and $\mathsf{up}_{\perp!}$. The counits $(-)_\perp \to \mathsf{id}$ are equal and are called down. Write $[d]$ for $\mathsf{up}_D(d) \in D_\perp$. The domain $D_\perp$ is obtained by adjoining a new bottom element to D. The morphism $f_\perp : D_\perp \to E_\perp$ is given by $f_\perp(\perp_{D_\perp}) = \perp_{E_\perp}$ and $f_\perp(d) = [f(d)]$ for $d \in D$. Given a functor F, we abbreviate the composition $(-)_\perp \circ F$ as $F_\perp$.

To make a function $f : \prod_{i\in I} A_i \to B$ strict in a component $j \in I$, we use the continuous function $\mathrm{strict}_j : [\prod_{i\in I} A_i \to B] \to [\prod_{i\in I} A_i \to B]$:

$$\mathrm{strict}_j(f)\left((a_i)_{i\in I}\right) = \begin{cases} \perp_B & \text{if } a_j = \perp_{A_j} \\ f\left((a_i)_{i\in I}\right) & \text{otherwise.} \end{cases} \tag{2}$$

An **O-category** [SP82, Definition 5; Gun92, p. 322] is a category such that

(1) every hom-set $[C \to D]$ is a dcpo,
(2) composition of morphisms is continuous with respect to the partial ordering on arrows.

Examples of **O**-categories include ω-**aBC**, ω-$\mathbf{aBC}_{\perp!}$, and functor categories $[\mathbf{C} \to \mathbf{D}]$ whenever $\mathbf{D}$ is an **O**-category. A functor $F : \mathbf{D} \to \mathbf{E}$ between **O**-categories is **locally continuous** if the maps $f \mapsto F(f) : [D_1 \to D_2] \to [F(D_1) \to F(D_2)]$ are continuous for all objects D_1, D_2 of $\mathbf{D}$. We write $[\mathbf{D} \xrightarrow{\text{l.c.}} \mathbf{E}]$ for the category of locally continuous functors from $\mathbf{D}$ to $\mathbf{E}$. Its morphisms are natural transformations.

A pair of morphisms $e : A \to B$ and $p : B \to A$ in an **O**-category $\mathbf{K}$ forms an embedding-projection pair or **e-p-pair** (e, p) if $p \circ e = \mathsf{id}_A$ and $e \circ p \sqsubseteq \mathsf{id}_B$. In this case, we say A is a **subdomain** of B. We say e is an **embedding** and p is a **projection**. The morphisms e and p uniquely determine each other and p is always a meet-homomorphism [AJ95, Propositions 3.1.10 and 3.1.13]. Given an e-p-pair (f, g), we may write g^e for f and f^p for g. We write $\mathbf{K}^e$ and $\mathbf{K}^p$ for the subcategories of $\mathbf{K}$ where all morphisms are embeddings and projections, respectively.

Given two functors $F, G : \mathbf{C} \to \mathbf{K}$ into an **O**-category $\mathbf{K}$, we say a natural transformation $\epsilon : F \Rightarrow G$ is a **natural pointwise-embedding** if each component $\epsilon_C : FC \to GC$ is an embedding. We say that it is a **natural embedding** if the family of projections $\{\epsilon^p_C : GC \to FC\}$ is a natural transformation $G \Rightarrow F$, i.e., if ϵ is an embedding in the functor category $[\mathbf{C} \to \mathbf{K}]$. The definitions of **natural pointwise-projection** and **natural projection** are analogous.

Let $F, H : \mathbf{D} \to \mathbf{E}$ and $G, I : \mathbf{C} \to \mathbf{D}$ be functors, and let $\eta : F \Rightarrow H$ and $\rho : G \Rightarrow I$ be natural transformations. They compose to form natural transformations $\eta G : FG \Rightarrow HG$ and $F\rho : FG \Rightarrow FI$ whose components at $C \in \mathbf{C}$ are $(\eta G)_C = \eta_{GC}$ and $(F\rho)_C = F(\rho_C)$. The **horizontal composition** $\eta * \rho : FG \Rightarrow HI$ is given by the equal natural transformations $\eta I \circ F\rho$ and $H\rho \circ \eta G$.

We write $\bot_{\mathbf{D}}$ and $\top_{\mathbf{D}}$ for the initial and terminal objects, respectively, of a category $\mathbf{D}$. When $\bot_{\mathbf{D}}$ and $\top_{\mathbf{D}}$ are isomorphic, we say that $\mathbf{D}$ has a **zero object** $0_{\mathbf{D}}$. We may drop subscripts when no ambiguity arises.

Given two functors $F : \mathbf{D} \to \mathbf{C}$ and $G : \mathbf{E} \to \mathbf{C}$, the **comma category** $F \downarrow G$ [Rie16, Exercise 1.3.vi] is given by the data:

- objects are triples (d, e, f) where d is an object of $\mathbf{D}$, e is an object of $\mathbf{E}$, and $f : Fd \to Ge$ is a morphism in $\mathbf{C}$; and
- morphisms $(d, e, f) \to (d', e', f')$ are pairs of morphisms (h, k) where $h : d \to d'$ in $\mathbf{D}$ and $k : e \to e'$ in $\mathbf{E}$ are such that $f' \circ Fh = Gk \circ f$, i.e., such that the following diagram commutes in $\mathbf{C}$:

$$\begin{array}{ccc} Fd & \xrightarrow{f} & Ge \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle Gk} \\ Fd' & \xrightarrow{f'} & Ge' \end{array}$$

Given an object C of $\mathbf{C}$, we write $C \downarrow G$ for $K_C \downarrow G$, where $K_C : \bullet \to \mathbf{C}$ is the constant functor onto the object C. We also write $F \downarrow \mathbf{C}$ for $F \downarrow \mathsf{id}_{\mathbf{C}}$.

CHAPTER 3

Completed Work

This chapter gives an overview of work that has been submitted for publication. An extended version of this work is available as [Kav20].

3.1. A Domain Semantics for Polarized SILL

The polarized SILL programming language [PG15; TCP13] cohesively integrates functional computation and message-passing concurrent computation. It is natural to ask: when are two SILL programs equivalent? The answer is deeply intertwined with the *semantics* of the language: before we can say that two pieces of syntax mean "the same thing", we must know the meaning of syntax. We use our computational intuitions and expected equivalences to motivate a domain-theoretic semantics.

Polarized SILL's functional layer is the simply-typed λ-calculus with a fixed-point operator and a call-by-value evaluation semantics. Encapsulated open processes are a base type. A hypothetical judgment $\Psi \Vdash M : \tau$ means the functional term M has functional type τ under the structural context Ψ of functional variables $x_i : \tau_i$. These judgments are inductively defined by the rules of fig. 3.1.

The semantics of the functional layer is standard [Cro93; Gun92; Rey09; Ten95]. A functional type τ denotes a Scott domain (an ω-algebraic bounded-complete pointed dcpo) $[\![\tau]\!]$. A structural context Ψ is interpreted as the Ψ-indexed product $[\![\Psi]\!] = \prod_{x_i \in \Psi} [\![\tau_i]\!]$. A functional term $\Psi \Vdash M : \tau$ denotes a continuous function $[\![\Psi \Vdash M : \tau]\!] : [\![\Psi]\!] \to [\![\tau]\!]$ in the category ω-**aBC** of Scott domains and continuous functions.

The process layer arises from a proofs-as-programs correspondence between the sequent calculus formulation of intuitionistic linear logic and the session-typed π-calculus [CP10]. A session type A describes a protocol for communicating over a channel. A process P provides a service A on a channel c while using zero or more services A_i on channels c_i. The used services form a linear context $\Delta = c_1 : A_1, \ldots, c_n : A_n$. The process P can use values from the functional layer. These are abstracted by a structural context Ψ of functional variables. These data are captured by the hypothetical judgment $\Psi \,;\, \Delta \vdash P :: c : A$. These judgments are inductively defined by the rules of fig. 3.2.

We cannot interpret processes $\Psi \,;\, \Delta \vdash P :: c : A$ in the same way as functional terms, that is, as functions $[\![\Psi]\!] \times [\![\Delta]\!] \to [\![A]\!]$. This is due to the fundamental difference between *variables* and *channel names*. A variable $x : \tau$ in the context Ψ stands for a value of type τ. A channel name in $\Delta, c : A$ does not stand for a value, but rather for a channel whose bidirectional communications obey its type. Informally, we interpret processes as continuous functions

$$[\![\Psi \,;\, \Delta \vdash P :: c : A]\!] : [\![\Psi]\!] \to [\text{“}\mathsf{inputOn}(\Delta, c : A)\text{”} \to \text{“}\mathsf{outputOn}(\Delta, c : A)\text{”}] \tag{3}$$

where we use the notation $[\cdot \to \cdot]$ for function spaces. This style of interpretation takes its roots in the semantics given by Kahn [Kah74] for dataflow. Processes should be monotone, for a longer input prefix should result in no less output. They should also be continuous, for a process should not be able to observe an entire infinitely-long communication before sending output.

To make this precise, we use the relationship between the polarity of a session type and the direction of communication along a channel of that type [PG15]. Session types can be partitioned as *positive* or *negative*. When looking at a judgment $\Psi \,;\, \Delta \vdash P :: c : A$, we can imagine "positive

$$\frac{\Psi\ ;\ \overline{a_i : A_i} \vdash P :: a : A}{\Psi \Vdash a \leftarrow \{P\} \leftarrow \overline{a_i} : \{a : A \leftarrow \overline{a_i : A_i}\}}\ (\text{I-}\{\}) \qquad \frac{}{\Psi, x : \tau \Vdash x : \tau}\ (\text{F-Var}) \qquad \frac{\Psi, x : \tau \Vdash M : \tau}{\Psi \Vdash \mathsf{fix}\ x.M : \tau}\ (\text{F-Fix})$$

$$\frac{\Psi, x : \tau \Vdash M : \sigma}{\Psi \Vdash \lambda x : \tau.M : \tau \to \sigma}\ (\text{F-Fun}) \qquad \frac{\Psi \Vdash M : \tau \to \sigma \quad \Psi \Vdash N : \tau}{\Psi \Vdash MN : \sigma}\ (\text{F-App})$$

Figure 3.1. Term formation in polarized SILL

$$\frac{}{\Psi\ ;\ a : A \vdash b \leftarrow a :: b : A}\ (\text{Fwd}) \qquad \frac{\Psi\ ;\ \Delta_1 \vdash P :: a : A \quad \Psi\ ;\ a : A, \Delta_2 \vdash Q :: c : C}{\Psi\ ;\ \Delta_1, \Delta_2 \vdash a \leftarrow P;\ Q :: c : C}\ (\text{Cut})$$

$$\frac{}{\Psi\ ;\ \cdot \vdash \mathsf{close}\ a :: a : \mathbf{1}}\ (\mathbf{1}R) \qquad \frac{\Psi\ ;\ \Delta \vdash P :: c : C}{\Psi\ ;\ \Delta, a : \mathbf{1} \vdash \mathsf{wait}\ a; P :: c : C}\ (\mathbf{1}L)$$

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

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

$$\frac{\Psi\ ;\ \Delta \vdash P :: a : A_k \quad (k \in L)}{\Psi\ ;\ \Delta \vdash a.k; P :: a : \oplus\{l : A_l\}_{l \in L}}\ (\oplus R_k) \qquad \frac{\Psi\ ;\ \Delta, a : A_l \vdash P_l :: c : C \quad (\forall l \in L)}{\Psi\ ;\ \Delta, a : \oplus\{l : A_l\}_{l \in L} \vdash \mathsf{case}\ a\ \{l_l \Rightarrow P_l\}_{i \in I} :: c : C}\ (\oplus L)$$

$$\frac{\Psi\ ;\ \Delta \vdash P_l :: a : A_l \quad (\forall l \in L)}{\Psi\ ;\ \Delta \vdash \mathsf{case}\ a\ \{l \Rightarrow P_l\}_{l \in L} :: a : \&\{l : A_l\}_{l \in L}}\ (\&R) \qquad \frac{\Psi\ ;\ \Delta, a : A_k \vdash P :: c : C \quad (k \in L)}{\Psi\ ;\ \Delta, a : \&\{l : A_l\}_{l \in L} \vdash a.k; P :: c : C}\ (\&L_k)$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: a : A}{\Psi\ ;\ \Delta, b : B \vdash \mathsf{send}\ a\ b;\ P :: a : B \otimes A}\ (\otimes R^*) \qquad \frac{\Psi\ ;\ \Delta, a : A, b : B \vdash P :: c : C}{\Psi\ ;\ \Delta, a : B \otimes A \vdash b \leftarrow \mathsf{recv}\ a;\ P :: c : C}\ (\otimes L)$$

$$\frac{\Psi\ ;\ \Delta, b : B \vdash P :: a : A}{\Psi\ ;\ \Delta \vdash b \leftarrow \mathsf{recv}\ a;\ P :: a : B \multimap A}\ (\multimap R) \qquad \frac{\Psi\ ;\ \Delta, a : A \vdash P :: c : C}{\Psi\ ;\ \Delta, b : B, a : B \multimap A \vdash \mathsf{send}\ a\ b;\ P :: c : C}\ (\multimap L)$$

$$\frac{\Psi \Vdash M : \tau \quad \Psi\ ;\ \Delta \vdash P :: a : A}{\Psi\ ;\ \Delta \vdash \_ \leftarrow \mathsf{output}\ a\ M; P :: a : \tau \wedge A}\ (\wedge R) \qquad \frac{\Psi, x : \tau\ ;\ \Delta, a : A \vdash Q :: c : C}{\Psi\ ;\ \Delta, a : \tau \wedge A \vdash x \leftarrow \mathsf{input}\ a; Q :: c : C}\ (\wedge L)$$

$$\frac{\Psi, x : \tau\ ;\ \Delta \vdash Q :: a : A}{\Psi\ ;\ \Delta \vdash x \leftarrow \mathsf{input}\ a; Q :: a : \tau \supset A}\ (\supset R) \qquad \frac{\Psi \Vdash M : \tau \quad \Psi\ ;\ \Delta, a : A \vdash P :: c : C}{\Psi\ ;\ \Delta, a : \tau \supset A \vdash \_ \leftarrow \mathsf{output}\ a\ M; P :: c : C}\ (\supset L)$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: a : [\rho\alpha.A/\alpha]A \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^+_{\mathsf{s}}}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ a\ \mathsf{unfold}; P :: a : \rho\alpha.A}\ (\rho^+R_U) \qquad \frac{\Psi\ ;\ \Delta, a : [\rho\alpha.A/\alpha]A \vdash P :: c : C \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^+_{\mathsf{s}}}{\Psi\ ;\ \Delta, a : \rho\alpha.A \vdash \mathsf{unfold} \leftarrow \mathsf{recv}\ a; P :: c : C}\ (\rho^+L_U)$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: a : \rho\alpha.A \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^+_{\mathsf{s}}}{\Psi\ ;\ \Delta \vdash \mathsf{send}\ a\ \mathsf{fold}; P :: a : [\rho\alpha.A/\alpha]A}\ (\rho^+R_F) \qquad \frac{\Psi\ ;\ \Delta, a : \rho\alpha.A \vdash P :: c : C \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^+_{\mathsf{s}}}{\Psi\ ;\ \Delta, a : [\rho\alpha.A/\alpha]A \vdash \mathsf{fold} \leftarrow \mathsf{recv}\ a; P :: c : C}\ (\rho^+L_F)$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: a : [\rho\alpha.A/\alpha]A \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^-_{\mathsf{s}}}{\Psi\ ;\ \Delta \vdash \mathsf{unfold} \leftarrow \mathsf{recv}\ a; P :: a : \rho\alpha.A}\ (\rho^-R_U) \qquad \frac{\Psi\ ;\ \Delta, a : [\rho\alpha.A/\alpha]A \vdash P :: c : C \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^-_{\mathsf{s}}}{\Psi\ ;\ \Delta, a : \rho\alpha.A \vdash \mathsf{send}\ a\ \mathsf{unfold}; P :: c : C}\ (\rho^-L_U)$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: a : \rho\alpha.A \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^-_{\mathsf{s}}}{\Psi\ ;\ \Delta \vdash \mathsf{fold} \leftarrow \mathsf{recv}\ a; P :: a : [\rho\alpha.A/\alpha]A}\ (\rho^-R_F) \qquad \frac{\Psi\ ;\ \Delta, a : \rho\alpha.A \vdash P :: c : C \quad \cdot \vdash \rho\alpha.A\ \mathsf{type}^-_{\mathsf{s}}}{\Psi\ ;\ \Delta, a : [\rho\alpha.A/\alpha]A \vdash \mathsf{send}\ a\ \mathsf{fold}; P :: c : C}\ (\rho^-L_F)$$

$$\frac{\Psi \Vdash M : \{a : A \leftarrow \overline{a_i : A_i}\} \quad \Psi\ ;\ a : A, \Delta \vdash Q :: c : C}{\Psi\ ;\ \overline{a_i : A_i}, \Delta \vdash a \leftarrow \{M\} \leftarrow \overline{a}_i; Q :: c : C}\ (\text{E-}\{\})$$

Figure 3.2. Process formation in polarized SILL

information" as flowing left-to-right and "negative information" as flowing right-to-left. For example, when the provided service A is positive, communication on c is sent by P; when A is negative, it is received by P. As P executes, the type of a channel $b : B$ in $\Delta, c : A$ evolves, sometimes becoming a positive subphrase of B, sometimes a negative subphrase B. It is this evolution that causes bidirectionality.

We use polarity to split a bidirectional communication into a pair of unidirectional communications. Given a protocol B, its *positive aspect* prescribes the left-to-right communication, while its *negative aspect* prescribes the right-to-left communication. This leads to two interpretations, each a Scott domain. The negative and positive aspects $[\![B]\!]^-$ and $[\![B]\!]^+$ respectively contain the "negative" and "positive" portions of the bidirectional communication. These polarized also aspects have strong computational motivations. The informal interpretation (3) is then made precise as

$$\frac{}{\Xi \vdash \mathbf{1}\ \mathsf{type}_\mathsf{s}^+}\ (\mathrm{C}\mathbf{1}) \qquad \frac{}{\Xi, \alpha\ \mathsf{type}_\mathsf{s}^p \vdash \alpha\ \mathsf{type}_\mathsf{s}^p}\ (\mathrm{CVar}) \qquad \frac{\Xi, \alpha\ \mathsf{type}_\mathsf{s}^p \vdash A\ \mathsf{type}_\mathsf{s}^p}{\Xi \vdash \rho\alpha.A\ \mathsf{type}_\mathsf{s}^p}\ (\mathrm{C}\rho)$$

$$\frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^-}{\Xi \vdash \downarrow A\ \mathsf{type}_\mathsf{s}^+}\ (\mathrm{C}\downarrow) \qquad \frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^+}{\Xi \vdash \uparrow A\ \mathsf{type}_\mathsf{s}^-}\ (\mathrm{C}\uparrow) \qquad \frac{\Xi \vdash A_l\ \mathsf{type}_\mathsf{s}^+ \quad (\forall\, l \in L)}{\Xi \vdash \oplus\{l : A_l\}_{l\in L}\ \mathsf{type}_\mathsf{s}^+}\ (\mathrm{C}\oplus) \qquad \frac{\Xi \vdash A_l\ \mathsf{type}_\mathsf{s}^- \quad (\forall\, l \in L)}{\Xi \vdash \&\{l : A_l\}_{l\in L}\ \mathsf{type}_\mathsf{s}^-}\ (\mathrm{C}\&)$$

$$\frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^+ \quad \Xi \vdash B\ \mathsf{type}_\mathsf{s}^+}{\Xi \vdash A \otimes B\ \mathsf{type}_\mathsf{s}^+}\ (\mathrm{C}\otimes) \qquad \frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^+ \quad \Xi \vdash B\ \mathsf{type}_\mathsf{s}^-}{\Xi \vdash A \multimap B\ \mathsf{type}_\mathsf{s}^-}\ (\mathrm{C}\multimap) \qquad \frac{\tau\ \mathsf{type}_\mathsf{f} \quad \sigma\ \mathsf{type}_\mathsf{f}}{\tau \to \sigma\ \mathsf{type}_\mathsf{f}}\ (\mathrm{T}\to)$$

$$\frac{\tau\ \mathsf{type}_\mathsf{f} \quad \Xi \vdash A\ \mathsf{type}_\mathsf{s}^+}{\Xi \vdash \tau \wedge A\ \mathsf{type}_\mathsf{s}^+}\ (\mathrm{C}\wedge) \qquad \frac{\tau\ \mathsf{type}_\mathsf{f} \quad \Xi \vdash A\ \mathsf{type}_\mathsf{s}^-}{\Xi \vdash \tau \supset A\ \mathsf{type}_\mathsf{s}^-}\ (\mathrm{C}\supset) \qquad \frac{\cdot \vdash A_i\ \mathsf{type}_\mathsf{s} \quad (0 \le i \le n)}{\{a_0 : A_0 \leftarrow a_1 : A_1, \ldots, a_n : A_n\}\ \mathsf{type}_\mathsf{f}}\ (\mathrm{T}\{\})$$

FIGURE 3.3. Type formation in polarized SILL

the continuous function

$$[\![\Psi\ ;\ \Delta \vdash P :: c : A]\!] : [\![\Psi]\!] \to [\,[\![\Delta]\!]^+ \times [\![c : A]\!]^- \to [\![\Delta]\!]^- \times [\![c : A]\!]^+\,] \tag{4}$$

in ω-**aBC**, where $[\![\Delta]\!]^p$ is the indexed product $\prod_{c_i^p} [\![A_i]\!]^p$ for $p \in \{-, +\}$.

Process composition $a \leftarrow P;\ Q$ arises from the proofs-as-processes interpretation of the (Cut) rule of intuitionistic linear logic:

$$\frac{\Psi\ ;\ \Delta_1 \vdash P :: a : A \quad \Psi\ ;\ a : A, \Delta_2 \vdash Q :: c : C}{\Psi\ ;\ \Delta_1, \Delta_2 \vdash a \leftarrow P;\ Q :: c : C}\ (\mathrm{Cut})$$

Operationally, the process $\Psi\ ;\ \Delta_1, \Delta_2 \vdash a \leftarrow P;\ Q :: c : C$ spawns two processes $\Psi\ ;\ \Delta_1 \vdash P :: a : A$ and $\Psi\ ;\ a : A, \Delta_2 \vdash Q :: c : C$ communicating along the shared channel a of type A. We follow prior work on the semantics of dataflow [Kah74] and on the geometry of interactions [AJ94; AHS02] and capture this feedback using a least fixed point. Given $u \in [\![\Psi]\!]$ and $(\delta_1^+, \delta_2^+, c^-) \in [\![\Delta_1, \Delta_2]\!]^+ \times [\![c : C]\!]^-$, we define

$$[\![\Psi\ ;\ \Delta_1, \Delta_2 \vdash a \leftarrow P;\ Q :: c : C]\!]u(\delta_1^+, \delta_2^+, c^-) = (\delta_1^-, \delta_2^-, c^+) \tag{5}$$

where δ_1^-, δ_2^-, a^-, a^+, and c^+ form the least solution to the equations

$$(\delta_1^-, a^+) = [\![\Psi\ ;\ \Delta_1 \vdash P :: a : A]\!]u(\delta_1^+, a^-),$$
$$(\delta_2^-, a^-, c^+) = [\![\Psi\ ;\ a : A, \Delta_2 \vdash Q :: c : C]\!]u(\delta_2^+, a^+, c^-).$$

This fixed point operation is an instance of the "trace operator" of traced monoidal categories [AHS02; CŞ90; JSV96]. A trace operator is a family of functions $\mathsf{Tr}^X_{A,B} : [A \times X \to B \times X] \to [A \to B]$ natural in A and B, dinatural in X, and satisfying a collection of axioms. It captures the "parallel composition plus hiding" operation considered by Milner [Mil80, pp. 20f.] and used by Abramsky et al. [AGN96]. As a result, we can reason about process composition using general identities for trace operators.

The judgment $A\ \mathsf{type}_\mathsf{s}^p$ means that A is a session type with polarity p. Open types are captured using the hypothetical judgment $\Xi \vdash A\ \mathsf{type}_\mathsf{s}^p$, where $\Xi = \alpha_1\ \mathsf{type}_\mathsf{s}^{p_1}, \ldots, \alpha_n\ \mathsf{type}_\mathsf{s}^{p_n}$ is a structural context of polarized type variables. We abbreviate these hypothetical judgments as $\Xi \vdash A\ \mathsf{type}_\mathsf{s}$ or $\Xi \vdash A$ when no ambiguity arises. The judgment $\tau\ \mathsf{type}_\mathsf{f}$ means that τ is a functional type. These judgments are inductively defined by the rules of fig. 3.3.

We build on standard domain-theoretic techniques to interpret session types and functional types. Hypothetical judgments $\Xi \vdash A$ denote locally continuous functors on a category of domains. Recursive types are interpreted as solutions to domain equations. Let ω-$\mathbf{aBC}_{\perp!}$ be the subcategory of ω-**aBC** whose morphisms are strict functions. We write $[\![\Xi]\!]$ for the Ξ-indexed product $\prod_{\alpha\in\Xi} \omega$-$\mathbf{aBC}_{\perp!}$. The denotations $[\![\Xi \vdash A]\!]^-$ and $[\![\Xi \vdash A]\!]^+$ are then locally continuous functors from $[\![\Xi]\!]$ to ω-$\mathbf{aBC}_{\perp!}$.

Our semantics does not use ω-algebraicity or bounded-completeness and all occurrences of ω-**aBC** and ω-$\mathbf{aBC}_{\perp!}$ could respectively be replaced by $\mathbf{DCPO}_\perp$ and $\mathbf{DCPO}_{\perp!}$. We mention the additional structure only for informational purposes.

To give the reader a flavour of the semantics, we give the semantic clauses for sending and receiving channels (section 3.1.1), and the semantic clauses involving recursive types (section 3.1.2).

3.1.1. Sending and Receiving Channels. The provider send $a\ b$; P sends the channel b over the channel a and continues as P. When the client $b \leftarrow \mathsf{recv}\ a$; P receives a channel over a, it binds it to the name b and continues as P. A service of type $B \otimes A$ sends a channel of type B and continues as a service of type A.

We cannot directly observe a channel, only the communications that are sent over it. For this reason, we treat communications of type $A \otimes B$ as a pair of communications: one for the sent channel and one for the continuation channel. This is analogous to the denotation of $A \otimes B$ given by Atkey [Atk17]. We account for the potential absence of communication by lifting.

$$[\![\Xi \vdash A \otimes B\ \mathsf{type_s}]\!]^- = [\![\Xi \vdash A\ \mathsf{type_s}]\!]^- \times [\![\Xi \vdash B\ \mathsf{type_s}]\!]^- \tag{6}$$

$$[\![\Xi \vdash A \otimes B\ \mathsf{type_s}]\!]^+ = \left([\![\Xi \vdash A\ \mathsf{type_s}]\!]^+ \times [\![\Xi \vdash B\ \mathsf{type_s}]\!]^+\right)_\perp \tag{7}$$

To send the channel b over a, the provider send $a\ b$; P must relay the positive communication from b^+ to the $[\![B]\!]^+$-component of $[\![a : B \otimes A]\!]^+$. It must also relay the negative information on the $[\![B]\!]^-$-component of $[\![a : B \otimes A]\!]^-$ to b^-. The continuation process P handles all other communication.

$$\begin{aligned}&[\![\Psi\ ;\ \Delta, b : B \vdash \mathsf{send}\ a\ b;\ P :: a : B \otimes A]\!]u(\delta^+, b^+, (a_B^-, a_A^-))\\ &= (\delta^-, a_B^-, [(b^+, a_A^+)])\ \text{ where } [\![\Psi\ ;\ \Delta \vdash P :: a : A]\!]u(\delta^+, a_A^-) = (\delta^-, a_A^+).\end{aligned} \tag{8}$$

The client $b \leftarrow \mathsf{recv}\ a$; Q blocks until it receives a channel on a. When it receives a positive communication $[(a_B^+, a_A^+)]$ on $[\![a : B \otimes A]\!]^+$, it unpacks it into the two positive channels $[\![a : A, b : B]\!]^+$ expected by Q. It then repacks the negative communication $[\![a : A, b : B]\!]^-$ produced by Q and relays it over $[\![a : B \otimes A]\!]^-$.

$$\begin{aligned}&[\![\Psi\ ;\ \Delta, a : B \otimes A \vdash b \leftarrow \mathsf{recv}\ a;\ Q :: c : C]\!]u(\delta^+, a^+, c^-)\\ &= \mathsf{strict}_{a^+}\left(\lambda(\delta^+, a^+ : [(a_B^+, a_A^+)], c^-).(\delta^-, (b^-, a^-), c^+)\right)\\ &\text{where } [\![\Psi\ ;\ \Delta, a : A, b : B \vdash Q :: c : C]\!]u(\delta^+, a_A^+, a_B^+, c^-) = (\delta^-, a^-, b^-, c^+).\end{aligned} \tag{9}$$

We validate our semantic clauses via the following η-like property:

Proposition 1. *For all* $\Psi\ ;\ \Delta_1 \vdash P :: a : A$ *and* $\Psi\ ;\ \Delta_2, a : A, b : B \vdash Q :: c : C$, *we have* $a \leftarrow P;\ Q \equiv b \leftarrow (\mathsf{send}\ a\ b;\ P)\ ;\ (b \leftarrow \mathsf{recv}\ a;\ Q)$.

3.1.2. Recursive types. The variable rule (TVar) is interpreted by projection. We interpret recursive types by the parametrized solution of a recursive domain equation. Every locally continuous functor $G : \omega\text{-}\mathbf{aBC} \to \omega\text{-}\mathbf{aBC}$ has a canonical fixed point $\mathsf{FIX}(G)$ in $\omega\text{-}\mathbf{aBC}$. Given a locally continuous functor $F : [\![\Xi]\!] \times \omega\text{-}\mathbf{aBC}_{\perp!} \to \omega\text{-}\mathbf{aBC}_{\perp!}$, the mapping $D \mapsto \mathsf{FIX}(F(D, -))$ extends to a locally continuous functor $F^\dagger : [\![\Xi]\!] \to \omega\text{-}\mathbf{aBC}_{\perp!}$ [AJ95, Proposition 5.2.7]. There exists a canonical natural isomorphism $\mathsf{Fold} : F \circ \langle \mathsf{id}_{[\![\Xi]\!]}, F^\dagger \rangle \Rightarrow F^\dagger$ with inverse Unfold. The rule (Cρ) denotes:

$$[\![\Xi \vdash \rho\alpha.A\ \mathsf{type_s}]\!]^p = \left([\![\Xi, \alpha\ \mathsf{type_s} \vdash A\ \mathsf{type_s}]\!]^p\right)^\dagger \quad (p \in \{-, +\}) \tag{10}$$

Processes can fold or unfold recursive types by sending or receiving fold or unfold messages. The direction of communication is determined by the polarity of the recursive type. For example, a process providing a positive recursive type sends the fold and unfold messages, while a process providing a negative recursive type receives the messages. The semantics is given by pre- and post-composition with the canonical natural isomorphisms Fold and Unfold for interpretation (10). By the substitution property [Kav20, Proposition 16] and interpretation (10),

$$[\![\Xi, \alpha\ \mathsf{type_s} \vdash A\ \mathsf{type_s}]\!]^p \circ \left\langle \mathsf{id}_{[\![\Xi]\!]}, \left([\![\Xi, \alpha\ \mathsf{type_s} \vdash A\ \mathsf{type_s}]\!]^p\right)^\dagger \right\rangle = [\![\Xi \vdash [\rho\alpha.A/\alpha]A\ \mathsf{type_s}]\!]^p.$$

Let $\mathsf{Fold}^p : [\![\Xi \vdash [\rho\alpha.A/\alpha]A\ \mathsf{type}_s]\!]^p \Rightarrow [\![\Xi \vdash \rho\alpha.A]\!]^p$ be the aforementioned natural isomorphism and let Unfold^p be its inverse. We interpret $(\rho^+ R_U)$ and $(\rho^+ L_U)$ by:

$$
\begin{aligned}
&[\![\Psi\ ;\ \Delta \vdash \mathsf{send}\ a\ \mathsf{unfold}; P :: a : \rho\alpha.A]\!]u \\
&= \left(\mathsf{id} \times \left(a^+ : \mathsf{Fold}^+\right)\right) \circ [\![\Psi\ ;\ \Delta \vdash P :: a : [\rho\alpha.A/\alpha]A]\!]u \circ \left(\mathsf{id} \times \left(a^- : \mathsf{Unfold}^-\right)\right)
\end{aligned} \tag{11}
$$

$$
\begin{aligned}
&[\![\Psi\ ;\ \Delta, a : \rho\alpha.A \vdash \mathsf{unfold} \leftarrow \mathsf{recv}\ a; P :: c : C]\!]u \\
&= \left(\mathsf{id} \times \left(a^- : \mathsf{Fold}^-\right)\right) \circ [\![\Psi\ ;\ \Delta, a : [\rho\alpha.A/\alpha]A \vdash P :: c : C]\!]u \circ \left(\mathsf{id} \times \left(a^+ : \mathsf{Unfold}^+\right)\right)
\end{aligned} \tag{12}
$$

The interpretations of the other six process rules for recursive types are analogous.

Proposition 2. *If* $\Psi\ ;\ \Delta_1 \vdash P :: a : [\rho\alpha.A/\alpha]A$ *and* $\Psi\ ;\ \Delta_2, a : [\rho\alpha.A/\alpha]A \vdash Q :: c : C$, *then* $a \leftarrow P;\ Q \equiv a \leftarrow (\mathsf{send}\ a\ \mathsf{unfold}; P);\ (\mathsf{unfold} \leftarrow \mathsf{recv}\ a; Q)$.

CHAPTER 4

Ongoing Work

4.1. 2-Categorical Analysis of Fixed Points of Functors

Functional programming languages often support recursive types. For example, in Standard ML we can define the type of (unary) natural numbers as:

```
datatype nat = Zero
             | Succ of nat
```

We interpret this definition as saying that a natural number is either zero (constructor `Zero`) or the successor of some natural number n (constructed as `Succ` n). Semantically, `nat` denotes a domain D satisfying the domain equation $D \cong (\texttt{Zero} : \top) \uplus (\texttt{Succ} : D)$. Indeed, every inhabitant of `nat` is either the tag `Zero` (accompanied by no other information), or an inhabitant of type `nat` tagged by `Succ`. This domain D is a fixed point of the functor $F_{\texttt{nat}}(X) = (\texttt{Zero} : \top) \uplus (\texttt{Succ} : X)$. More generally, a recursive type denotes a fixed point of the functor its definition induces. As a result, giving the denotation of a recursive type corresponds to finding a fixed point of a functor.

The techniques for finding fixed points of functors on categories of domains are well understood [AJ95, Chapter 5; AL91, Chapter 10; Gie+03, § IV-7; Gun92, Chapter 10; GM89; SP82; Ten91, Chapter 10]. Consider a locally continuous functor $F : \mathbf{D} \to \mathbf{D}$ on a suitable category $\mathbf{D}$ of domains. To construct a domain X such that $F(X) \cong X$, we begin by constructing an "ω-chain" F^ω of the iterates of F applied to the initial object:

$$\bot \to F\bot \to F^2\bot \to F^3\bot \to \cdots$$

Here, $\bot \to F\bot$ is given by initiality, and $F^{n+1}\bot \to F^{n+2}\bot$ is $F(F^n\bot \to F^{n+1}\bot)$. If the colimit $\mathsf{FIX}(F)$ of this diagram exists, then it is a fixed point of F. We call the isomorphisms $F(\mathsf{FIX}(F)) \to \mathsf{FIX}(F)$ and $\mathsf{FIX}(F) \to F(\mathsf{FIX}(F))$ witnessing the fixed point fold and unfold, respectively.

The solution $\mathsf{FIX}(F)$ is *canonical* in the following sense. An F-algebra is a pair (A, α) where A is a domain and $\alpha : FA \to A$ is a morphism in $\mathbf{D}$. An F-algebra homomorphism $(A, \alpha) \to (B, \beta)$ is a morphism $f : A \to B$ in $\mathbf{D}$ such that the following diagram commutes:

$$\begin{array}{ccc} FA & \xrightarrow{\alpha} & A \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FB & \xrightarrow{\beta} & B \end{array} \tag{13}$$

These F-algebras and their homomorphisms form a category $\mathbf{D}^F$. The solution $\mathsf{FIX}(F)$ is canonical in the sense that $(\mathsf{FIX}(F), \mathsf{fold})$ is the initial F-algebra: given any other F-algebra (D, δ), there exists a unique F-algebra homomorphism $(\mathsf{FIX}(F), \mathsf{fold}) \to (D, \delta)$.

Sometimes when programming we find the need to define mutually-recursive data types. Consider, for example, the types of even and odd natural numbers:

```
datatype even = Zero
              | E of odd
     and odd  = O of even
```

This type definition captures the fact that an even number is either zero or the successor of an odd number, and that an odd number is the successor of an even one. The types `even` and `odd`

respectively denote solutions $D_{\mathtt{e}}$ and $D_{\mathtt{o}}$ to the system of domain equations:

$$D_{\mathtt{e}} \cong (\mathtt{Zero} : \top) \uplus (\mathtt{E} : D_{\mathtt{o}})$$
$$D_{\mathtt{o}} \cong (\mathtt{O} : D_{\mathtt{e}}),$$

These are the solutions to the system of equations:

$$X_{\mathtt{e}} \cong F_{\mathtt{even}}(X_{\mathtt{e}}, X_{\mathtt{o}}) \tag{14}$$
$$X_{\mathtt{o}} \cong F_{\mathtt{odd}}(X_{\mathtt{e}}, X_{\mathtt{o}}) \tag{15}$$

where $F_{\mathtt{even}}$ and $F_{\mathtt{odd}}$ are the functors:

$$F_{\mathtt{even}}(X_{\mathtt{e}}, X_{\mathtt{o}}) = (\mathtt{Zero} : \top) \uplus (\mathtt{E} : X_{\mathtt{o}}),$$
$$F_{\mathtt{odd}}(X_{\mathtt{e}}, X_{\mathtt{o}}) = (\mathtt{O} : X_{\mathtt{e}}).$$

We can use Bekič's rule [Bek84, § 2] to solve this system of equations. We think of eq. (14) as a family of equations parametrized by $X_{\mathtt{o}}$. If we could solve for $X_{\mathtt{e}}$, then we would get a parametrized family of solutions $F^{\dagger}_{\mathtt{even}}(X_{\mathtt{o}})$ such that

$$F^{\dagger}_{\mathtt{even}}(X_{\mathtt{o}}) \cong F_{\mathtt{even}}(F^{\dagger}_{\mathtt{even}}(X_{\mathtt{o}}), X_{\mathtt{o}}) \tag{16}$$

for all domains $X_{\mathtt{o}}$. Substituting this for $X_{\mathtt{e}}$ in eq. (15) gives the domain equation

$$X_{\mathtt{o}} \cong F_{\mathtt{odd}}(F^{\dagger}_{\mathtt{even}}(X_{\mathtt{o}}), X_{\mathtt{o}}).$$

Solving for $X_{\mathtt{o}}$ gives the solution $D_{\mathtt{o}}$. Substituting $D_{\mathtt{o}}$ for $X_{\mathtt{o}}$ in eq. (16), we see that $D_{\mathtt{e}} = F^{\dagger}_{\mathtt{even}}(D_{\mathtt{o}})$ is the other part of the solution.

The above example motivates techniques to solve parametrized domain equations. These techniques are also well-understood and are summarized by proposition 3. Given a category $\mathbf{D}$ of domains, we write $\mathbf{D}_{\perp !}$ for the subcategory of strict morphisms.

Proposition 3 ([AJ95, Proposition 5.2.7]). *Let $\mathbf{D}$ and $\mathbf{E}$ be categories of pointed domains closed under bilimits. Let $F : \mathbf{D}_{\perp !} \times \mathbf{E}_{\perp !} \to \mathbf{E}_{\perp !}$ be a locally continuous functor and write F_D for $F(D, \cdot)$. Then the following defines a locally-continuous functor $F^{\dagger}$ from $\mathbf{D}_{\perp !}$ to $\mathbf{E}_{\perp !}$:*

- *On objects : $D \mapsto \mathsf{FIX}(F_D)$,*
- *on morphisms : $(f : D \to D') \mapsto \bigsqcup^{\uparrow}_{n \in N} e'_n \circ f_n \circ e^p_n$,*

where $e : F^{\omega}_D \to \mathsf{FIX}(F_D)$ and $e' : F^{\omega}_{D'} \to \mathsf{FIX}(F_{D'})$ are the colimiting cones, and where the sequence $(f_n)_{n \in N}$ is defined recursively by $f_{\mathtt{o}} = \mathsf{id}_{\perp}$, $f_{n+1} = F(f_n, f)$.

Though proposition 3 gives an explicit "recipe" for the mapping $F \mapsto F^{\dagger}$, it says nothing about the nature of this mapping. Is this mapping functorial? Do F and $F^{\dagger}$ satisfy any of the identities of fixed-point theory, e.g., the left zero identity or the parameter identities of Bloom and Ésik [BÉ96]? In what sense is the family of solutions $F^{\dagger}$ canonical? Our semantics of recursive session types depends crucially on the answers to these questions and others.

4.1.1. Background and Related Work. **O**-categories [SP82] generalize categories of domains to provide just the structure required to find solutions to domain equations. We refer the reader to chapter 2 for their definition.

Let ω be the category whose objects are natural numbers, and where there exists a morphism $m \to n$ if and only if $m \leq n$. An ω-**chain** is a ω-shaped diagram.

Given an **O**-category $\mathbf{K}$ and a cocone $\mu : \Phi \to A$ in $\mathbf{K}^e$ on an ω-chain Φ, we say μ is an **O-colimit** [SP82, Definition 7] of Φ whenever $(\mu_n \circ \mu^p_n)_{n \in \mathbb{N}}$ is an ascending chain in $[A \to A]$ and $\bigsqcup^{\uparrow}_{n \in \mathbb{N}} \mu_n \circ \mu^p_n = \mathsf{id}_A$.

An **O**-category $\mathbf{K}$ has **locally determined ω-colimits of embeddings** [SP82, Definition 8] if for all ω-chains $\Phi : \omega \to \mathbf{K}^e$ and cocones $\mu : \Phi \to A$ in $\mathbf{K}^e$, μ is colimiting in $\mathbf{K}^e$ if and only if μ is an **O**-colimit. Dually, a cone $\mu : A \to \Phi$ in $\mathbf{K}^p$ on an ω^{op}-chain Φ is an **O-limit** [SP82, Definition 7] of Φ whenever $(\mu^p_n \circ \mu_n)_{n \in \mathbb{N}}$ is an ascending chain in $[A \to A]$ and $\bigsqcup^{\uparrow}_{n \in \mathbb{N}} \mu^p_n \circ \mu_n = \mathsf{id}_A$.

Our interest in **O**-colimits is due to:

PROPOSITION 4 ([SP82, Propositions A and D]). *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category,* Φ *an* ω*-chain in* $\mathbf{K}^e$*, and* $\mu : \Phi \to A$ *a cocone in* $\mathbf{K}$*. The cocone* μ *is colimiting in* $\mathbf{K}$ *if and only if it is an* $\mathbf{O}$*-colimit. If* μ *is colimiting in* $\mathbf{K}$*, then it is also colimiting in* $\mathbf{K}^e$*.*

Proposition 4 and its dual imply the limit-colimit-coincidence.

PROPOSITION 5 (Limit-colimit coincidence [SP82, Theorem 2]). *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category,* Φ *an* ω*-chain in* $\mathbf{K}^e$*, and* $\phi : \Phi \to A$ *a cocone in* $\mathbf{K}$*. Let* $\Phi^p : \omega^{\mathrm{op}} \to \mathbf{K}^p$ *be* ω^{op}*-chain obtained by taking the projection of each embedding. The following are equivalent:*

(1) ϕ *is an* ω*-colimit of* Φ *in* $\mathbf{K}$*,*
(2) ϕ *is an* $\mathbf{O}$*-colimit of* Φ*,*
(3) ϕ^p *is an* ω^{op}*-limit of* Φ^p *in* $\mathbf{K}$*, and*
(4) ϕ^p *is an* $\mathbf{O}$*-limit of* Φ^p*.*

Bloom and Ésik [BÉ95] study the interaction between the Conway identities and external dagger operators on 2-categories. Their definition of external dagger operator [BÉ95, Definition 2.6] concerns only horizontal morphisms. They use initial algebras to define an external dagger operator on horizontal morphisms in 2-categories. Indeed, given a horizontal morphism $F : \mathbf{E} \times \mathbf{D} \to \mathbf{E}$, they take $F^\dagger : \mathbf{D} \to \mathbf{E}$ to be the carrier of the initial algebra of the horizontal morphism H from proposition 11 below.

4.1.2. Contributions. I propose continuing to explore the 2-categorical structure [Mac98, § 10.3] of fixed points of functors on $\mathbf{O}$-categories. This section contains some answers to the questions following proposition 3. These answers lead to new questions, which we also pose.

We begin by stating without proof that the mapping $F \mapsto F^\dagger$ is functorial. Given an $\mathbf{O}$-category $\mathbf{K}$ and an object K of $\mathbf{K}$, let $\mathbf{Link}_{\mathbf{K}}(K)$ be the full subcategory of $K \downarrow [\mathbf{K} \xrightarrow{\text{l.c.}} \mathbf{K}]$ spanned by objects (ϕ, F) where $\phi_K : K \to FK$ is an embedding. Objects (ϕ, F) of this category are called "links" because they form the start of an ω-chain.

PROPOSITION 6 (Functoriality of FIX). *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category and assume* $\mathbf{K}^e$ *is* ω*-cocomplete. Let* K *be an object of* $\mathbf{K}$*. There exists a functor* $\mathsf{FIX}_K : \mathbf{Link}_{\mathbf{K}}(K) \to \mathbf{K}$ *where* $\mathsf{FIX}_K(\phi, F)$ *is a fixed point of* F*. If* $\mathbf{K}$ *has locally determined* ω*-colimits of embeddings, then this functor is locally continuous. If* $\eta : (\phi, F) \to (\gamma, G)$ *lies in* $\mathbf{K}^e$*, then so does* $\mathsf{FIX}_K \eta$*.*

QUESTION 1. *In proposition 6, we privilege the "objects axis" over the "functor axis". Indeed,* FIX_K *is a functor out of a category* $\mathbf{Link}_{\mathbf{K}}(K)$ *for some object* K *of* $\mathbf{K}$*. Can we reformulate the statement so that* FIX *is a functor out of some category* $\mathbf{Link}_{\mathbf{K}}$ *that does not depend on a particular choice of object* K*? The categories* $\mathbf{Link}_{\mathbf{K}}(K)$ *are indexed by objects* K *of* $\mathbf{K}$*. Is there a fibrational structure lurking in the background?*

Proposition 7 states that folding and unfolding fixed points of functors F is natural in F. To do so, we first explain in lemma 1 how unfolding a fixed point is a functorial operation. Given an object (ϕ, F) of $\mathbf{Link}_{\mathbf{K}}(K)$, let $(\phi, F)^\omega$ be the ω-chain generated by iteratively applying F to the morphism $\phi : K \to FK$.

LEMMA 1. *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category with locally determined* ω*-colimits of embeddings, and assume* $\mathbf{K}^e$ *is* ω*-cocomplete. Let* K *be an object of* $\mathbf{K}$*. The following defines a functor* $\mathsf{UNF}_K : \mathbf{Link}_{\mathbf{K}}(K) \to \mathbf{K}$*.*

- *On objects:* $\mathsf{UNF}_K(\phi, F) = F(\mathsf{FIX}_K(\phi, F))$*,*
- *on morphisms:* $\mathsf{UNF}_K(\rho : (\phi, F) \to (\gamma, G)) = \bigsqcup^{\uparrow}_{n \in \mathbb{N}} G\nu_n \circ \rho^{\omega}_{n+1} \circ (F\mu_n)^p$*, where* $\mu : (\phi, F)^\omega \to \mathsf{FIX}_K(\phi, F)$ *and* $\nu : (\gamma, G)^\omega \to \mathsf{FIX}_K(\gamma, G)$ *are the colimiting cones.*

PROPOSITION 7. *Let* $\mathbf{K}$ *be an* $\mathbf{O}$*-category with locally determined* ω*-colimits of embeddings, and assume* $\mathbf{K}^e$ *is* ω*-cocomplete. Let* K *be an object of* $\mathbf{K}$*. Then there exists a natural isomorphism* $\mathsf{unfold} : \mathsf{FIX}_K \to \mathsf{UNF}_K$ *with inverse* $\mathsf{fold} : \mathsf{UNF}_K \to \mathsf{FIX}_K$*. They are explicitly given as follows. Let*

(ϕ, F) be an object of $\mathbf{Link}_{\mathbf{K}}(K)$, and let $\mu : (\phi, F)^\omega \to \mathsf{FIX}_K(\phi, F)$ be the colimiting cone. The components are:

$$\mathsf{fold}_{(\phi,F)} = \bigsqcup_{n\in N}^{\uparrow} \mu_{n+1} \circ F(\mu_n)^p : F(\mathsf{FIX}_K(\phi, F)) \to \mathsf{FIX}_K(\phi, F)$$
$$\mathsf{unfold}_{(\phi,F)} = \bigsqcup_{n\in N}^{\uparrow} F(\mu_n) \circ \mu_{n+1}^p : \mathsf{FIX}_K(\phi, F) \to F(\mathsf{FIX}_K(\phi, F)).$$

Naturality means that for any $\rho : (\phi, F) \to (\gamma, G)$, the following squares commutes in $\mathbf{K}$:

$$\begin{array}{ccc} \mathsf{FIX}_K(\phi,F) & \xrightarrow{\mathsf{unfold}_{(\phi,F)}} & F(\mathsf{FIX}_K(\phi,F)) \\ {\scriptstyle \mathsf{FIX}_K\rho}\downarrow & & \downarrow{\scriptstyle \mathsf{UNF}_K\rho} \\ \mathsf{FIX}_K(\gamma,G) & \xrightarrow{\mathsf{unfold}_{(\gamma,G)}} & G(\mathsf{FIX}_K(\gamma,G)), \end{array} \qquad \begin{array}{ccc} \mathsf{FIX}_K(\phi,F) & \xleftarrow{\mathsf{fold}_{(\phi,F)}} & F(\mathsf{FIX}_K(\phi,F)) \\ {\scriptstyle \mathsf{FIX}_K\rho}\downarrow & & \downarrow{\scriptstyle \mathsf{UNF}_K\rho} \\ \mathsf{FIX}_K(\gamma,G) & \xleftarrow{\mathsf{fold}_{(\gamma,G)}} & G(\mathsf{FIX}_K(\gamma,G)). \end{array}$$

Our original interest was in solutions to parametrized solutions to domain equations. Proposition 3 requires that the categories of domains have strict morphisms. We generalize this concept to **O**-categories. An **O**-category **K** has **strict morphisms** if it has a zero object $\mathrm{o}_{\mathbf{K}}$ and the limiting cone $\mathrm{o} : \mathrm{o}_{\mathbf{K}} \to \mathrm{id}_{\mathbf{K}}$ lies in $\mathbf{K}^e$. This definition captures just the structure required and allows us to prove a variety of expected properties, e.g., proposition 8.

Proposition 8. *If **K** is an **O**-category with strict morphisms, then for all objects A and B, the zero morphism $\mathrm{o}_{AB} : A \to B$ is the least element of $[A \to B]$.*

Question 2. *Fiore [Fio94, Definition 8.5.10] defines strict morphisms between complete objects. How are these definitions related?*

An **O**-category **K supports canonical fixed points** if

(1) it has strict morphisms,
(2) locally determined ω-colimits of embeddings, and
(3) $\mathbf{K}^e$ is ω-cocomplete.

In this case, we write FIX for $\mathsf{FIX}_\perp$.

Proposition 9 generalizes proposition 3. It states that finding solutions to parametrized domain equations is functorial.

Proposition 9. *Let **E** and **D** be **O**-categories. Assume **E** supports canonical fixed points. The following composition is a well-defined functor:*

$$(\cdot)^\dagger = [\mathrm{id}_{\mathbf{D}} \to \mathsf{FIX}] \circ \Lambda : [\mathbf{D} \times \mathbf{E} \xrightarrow{\text{l.c.}} \mathbf{E}] \to [\mathbf{D} \xrightarrow{\text{l.c.}} \mathbf{E}].$$

*Explicitly, $F^\dagger D = \mathsf{FIX}(F_D)$ is the canonical fixed point of the partial application of F to the object D. When **E**, **D**, and F satisfy the hypotheses of proposition 3, then $F^\dagger$ is the locally continuous functor given by that proposition.*

This operation satisfies a *weak fixed-point* identity (diagram 17). This weak fixed-point identity generalizes the fixed-point identity [BÉ96, p. 10] to the case of functors, where the two functors are not equal but only naturally isomorphic:

(17)
$$\begin{array}{ccc} \mathbf{D} & \xrightarrow{\langle \mathrm{id}, F^\dagger \rangle} & \mathbf{D} \times \mathbf{E} \\ & \searrow^{F^\dagger} & \cong \downarrow F \\ & & \mathbf{E} \end{array}$$

This identity is given by proposition 10:

Proposition 10 (Weak fixed point identity). *Let* $\mathbf{E}$ *and* $\mathbf{D}$ *be* $\mathbf{O}$*-categories. Assume* $\mathbf{E}$ *supports canonical fixed points. Let* $F : \mathbf{E} \times \mathbf{D} \to \mathbf{E}$ *be a locally continuous functor. There exist natural transformations*

$$\mathsf{Unfold} : F^\dagger \to F \circ \langle \mathsf{id}_{\mathbf{D}}, F^\dagger \rangle$$
$$\mathsf{Fold} : F \circ \langle \mathsf{id}_{\mathbf{D}}, F^\dagger \rangle \to F^\dagger$$

that form a natural isomorphism $F^\dagger \cong F \circ \langle \mathsf{id}, F^\dagger \rangle$*. Let* D *be an object of* $\mathbf{D}$*. The* D*-components for these natural transformations are given by*

$$\mathsf{Unfold}_D = \mathsf{unfold}_{(\perp, F_D)} : \mathsf{FIX}(F_D) \to \mathsf{UNF}(F_D),$$
$$\mathsf{Fold}_D = \mathsf{fold}_{(\perp, F_D)} : \mathsf{UNF}(F_D) \to \mathsf{FIX}(F_D),$$

where unfold *and* fold *are the natural isomorphisms given by proposition 7.*

The parametrized solutions given by proposition 9 are canonical in the following sense:

Proposition 11. *Let* $\mathbf{E}$ *and* $\mathbf{D}$ *be* $\mathbf{O}$*-categories, and assume* $\mathbf{E}$ *supports canonical fixed points. Let* $F : \mathbf{E} \times \mathbf{D} \to \mathbf{E}$ *be a locally continuous functor. Let* $F^\dagger : \mathbf{D} \to \mathbf{E}$ *be given by proposition 9, and* Fold *and* Unfold *by proposition 10. Then* $(F^\dagger, \mathsf{Fold})$ *and* $(F^\dagger, \mathsf{Unfold})$ *are respectively the initial algebra and terminal coalgebra for the functor*

$$H = F \circ \langle -, \mathsf{id}_{\mathbf{D}} \rangle : [\mathbf{D} \xrightarrow{\text{l.c.}} \mathbf{E}] \to [\mathbf{D} \xrightarrow{\text{l.c.}} \mathbf{E}].$$

1. *Given any other* H*-algebra* (A, α)*, the mediating morphism* $\phi : F^\dagger \to A$ *is a natural pointwise-embedding. The component* ϕ_D *is the unique* F_D*-algebra homomorphism* $(F^\dagger D, \mathsf{Fold}_D) \to (GD, \gamma_D)$*.*
2. *Given any other* H*-coalgebra* (Γ, γ)*, the mediating morphism* $\rho : \Gamma \to F^\dagger$ *is a natural pointwise-projection. The component* ρ_D *is the unique* F_D*-coalgebra homomorphism* $(GD, \gamma_D) \to (F^\dagger D, \mathsf{Unfold}_D)$*.*
3. *Given an* H*-algebra* (A, α) *where* α *is a natural embedding,* (A, α^p) *is an* H*-coalgebra and* (ϕ, ρ) *form a natural e-p-pair.*

We have suggestively used the dagger notation $F^\dagger$ of Bloom and Ésik [BÉ95; BÉ96] to describe parametrized fixed points. Bloom and Ésik study many useful identities that dagger operations can satisfy. Which identities does the dagger operator from proposition 9 satisfy? The cartesian Conway identities [BÉ96, Definition 3.3; BÉ95, § 3] axiomatize iteration theories and imply many other identities. Does proposition 9 satisfy the cartesian Conway identities?

Our interest in these identities is not just of a theoretical nature: they are very practical. For example, to show that our interpretations of types as natural transformations (see section 4.2) respect the substitution property, we must show that proposition 9 satisfies variants of the "parameter identity" [BÉ96, p. 8]. In particular, to show that the natural transformations for the recursive types satisfy the substitution property, we rely on corollary 3. Corollary 1 tells us that $(\cdot)^\dagger$ is natural in $\mathbf{D}$.

Corollary 1 (Parameter identity). *Let* $\mathbf{C}$*,* $\mathbf{D}$*, and* $\mathbf{E}$ *be* $\mathbf{O}$*-categories. Assume* $\mathbf{E}$ *supports canonical fixed points. Let* $F : \mathbf{D} \times \mathbf{E} \to \mathbf{E}$ *and* $G : \mathbf{C} \to \mathbf{D}$ *be locally continuous functors. Then* $(F \circ (G \times \mathsf{id}_{\mathbf{E}}))^\dagger = F^\dagger \circ G$*.*

The weak fixed point identity and the parameter identity interact as follows:

Corollary 2 (Parameter identity II). *Let* $\mathbf{C}$*,* $\mathbf{D}$*, and* $\mathbf{E}$ *be* $\mathbf{O}$*-categories. Assume* $\mathbf{E}$ *supports canonical fixed points. Let* $F : \mathbf{D} \times \mathbf{E} \to \mathbf{E}$ *and* $G : \mathbf{C} \to \mathbf{D}$ *be locally continuous functors, and let* $W = (F \circ (G \times \mathsf{id}_{\mathbf{E}}))$*. Let* Fold^F *and* Fold^W *be the natural isomorphisms given by proposition 10. Then* $\mathsf{Fold}^W = \mathsf{Fold}^F G$ *and* $\mathsf{Unfold}^W = \mathsf{Unfold}^F G$*.*

Corollary 3 (Parameter identity III). *Let* $\mathbf{C}$*,* $\mathbf{D}$*, and* $\mathbf{E}$ *be* $\mathbf{O}$*-categories. Assume* $\mathbf{E}$ *supports canonical fixed points. Let* $F, I : \mathbf{D} \times \mathbf{E} \to \mathbf{E}$ *and* $G, J : \mathbf{C} \to \mathbf{D}$ *be locally continuous functors, and let* $W = (F \circ (G \times \mathsf{id}_{\mathbf{E}}))$ *and* $V = (I \circ (J \times \mathsf{id}_{\mathbf{E}}))$*. Let* $\eta : F \to G$ *and* $\phi : G \to J$*. Then* $(\eta * \phi)^\dagger = \eta^\dagger * \phi : W^\dagger \to V^\dagger$*.*

Corollary 4 is an instance of what Bloom and Ésik [BÉ96, p. 10] call the "left zero identity".

Corollary 4 (Left zero identity). *Let* $\mathbf{D}$ *and* $\mathbf{E}$ *be* $\mathbf{O}$*-categories. Assume* $\mathbf{E}$ *supports canonical fixed points and let* $F : \mathbf{D} \to \mathbf{E}$ *be a locally continuous functor. Then*

$$\left(\mathbf{E} \times \mathbf{D} \xrightarrow{\pi} \mathbf{D} \xrightarrow{F} \mathbf{E}\right)^{\dagger} = \mathbf{D} \xrightarrow{F} \mathbf{E}.$$

We conclude by considering size issues. We frequently use functor categories $[\mathbf{C} \to \mathbf{D}]$ where $\mathbf{C}$ and $\mathbf{D}$ are categories of domains. The categories of domains that we care about are never small, but definitions of $[\mathbf{C} \to \mathbf{D}]$ typically require that $\mathbf{C}$ be small. We can use a hierarchy of *universes* as described in [Sch72, § 3.2] to address these issues. We conjecture that we could alternatively use a single universe, as described in [Mac69b].

4.2. Canonical Interpretations of Session Types and Junk-Freedom

In section 3.1, we gave polarized interpretations $[\![A]\!]^-$ and $[\![A]\!]^+$ to each session type A. A pair $(a^-, a^+) \in [\![A]\!]^- \times [\![A]\!]^+$ is meant to capture a bidirectional communication on a channel of type A. However, there are many such pairs that do not capture an actual bidirectional communication. To address this issue, we introduce a third interpretation $[\![A]\!]$, the **canonical interpretation** of A, that captures entire bidirectional communications in a single domain. This canonical interpretation is defined so that $[\![A]\!]$ is a subdomain of $[\![A]\!]^- \times [\![A]\!]^+$. The pairs (a^-, a^+) that capture bidirectional communications are then those in the image of $[\![A]\!]$.

To relate the interpretations of open types, we generalize from a single embedding-projection pair to natural embeddings. We define two new denotations $(\!|\, \Xi \vdash A\ \mathsf{type_s} \,|\!)^-$ and $(\!|\, \Xi \vdash A\ \mathsf{type_s} \,|\!)^+$ that are natural transformations from the canonical interpretations to the polarized aspects. They satisfy proposition 12, which intuitively tells us that the domain of bidirectional communications "lives in" the product of the polarized aspects (unidirectional communications).

Proposition 12. *If* $\Xi \vdash A\ \mathsf{type_s}$ *does not use (C*ρ*), then there exists a pair of natural transformations* $(\!|\, \Xi \vdash A\ \mathsf{type_s} \,|\!)^-$ *and* $(\!|\, \Xi \vdash A\ \mathsf{type_s} \,|\!)^+$ *such that*

$$[\![\Xi \vdash A\ \mathsf{type_s}]\!] \xrightarrow{\langle (\!|\, \Xi \vdash A\ \mathsf{type_s} \,|\!)^-, (\!|\, \Xi \vdash A\ \mathsf{type_s} \,|\!)^+ \rangle} [\![\Xi \vdash A\ \mathsf{type_s}]\!]^- \times [\![\Xi \vdash A\ \mathsf{type_s}]\!]^+$$

is a natural embedding.

For order-theoretic reasons, proposition 12 requires us to restrict the interpretations of types from functors on ω-$\mathbf{aBC}_{\perp!}$ to functors on its subcategory of meet-homomorphisms. It is unclear whether proposition 12 holds for all $\Xi \vdash A\ \mathsf{type_s}$, i.e., whether it also holds for recursive types.

I expect that this ongoing work will be key to eliminating "semantic junk" so that I can show soundness and adequacy of my denotational semantics relative to standard substructural operational semantics for SILL-like languages (see section 5.1 below). In particular, we would like the interpretations of processes to not produce semantic junk. More precisely, we would like a process to map the unidirectional portion of a bidirectional communication to the corresponding unidirectional portion in the opposite direction. To make this explicit, set

$$\mathcal{I}(a_1 : A_1, \ldots, a_n : A_n \vdash a_o : A_o) = \left(\prod_{i=1}^{n} (a_i^+ : (\!|\, A_i \,|\!)^+)\right) \times (a_o^- : (\!|\, A_o \,|\!)^-)$$

$$\mathcal{O}(a_1 : A_1, \ldots, a_n : A_n \vdash a_o : A_o) = \left(\prod_{i=1}^{n} (a_i^- : (\!|\, A_i \,|\!)^-)\right) \times (a_o^+ : (\!|\, A_o \,|\!)^+)\,.$$

As an initial definition, we could say that a function $p \in [\![\Delta \vdash a : A]\!]$ is "junk-free" if for all $x \in \mathrm{dom}(p)$, $(x, p(x)) \in \mathrm{im}(\langle \mathcal{I}(\Delta \vdash a : A), \mathcal{O}(\Delta \vdash a : A) \rangle)$. This captures the above intuitions, but is too naïve. Indeed, if p is the denotation of a process, that process could get stuck and not consume its entire input when producing its output. The following definition addresses this possibility. A function $p \in [\![\Delta \vdash a : A]\!]$ is **junk-free** if for all $x \in \mathrm{dom}(p)$, there exists a $y \sqsubseteq x$ such that $p(y) = p(x)$ and $(y, p(y)) \in \mathrm{im}(\langle \mathcal{I}(\Delta \vdash a : A), \mathcal{O}(\Delta \vdash a : A) \rangle)$. We say that p is **adjointly**

junk-free if these y are determined by a lower adjoint $\overleftarrow{p} \dashv p$, i.e., if p has a lower adjoint $\overleftarrow{p}$ such that for all $x \in \mathrm{dom}(p)$, $\langle \overleftarrow{p} \circ p, p \rangle(x) \in \mathrm{im}(\langle \mathcal{I}(\Delta \vdash a : A), \mathcal{O}(\Delta \vdash a : A) \rangle)$. (Recall from [AJ95, Proposition 3.1.12] that if $\overleftarrow{p} \dashv p$, then $p \circ \overleftarrow{p} \circ p = p$.)

Conjecture 1 (Junk-freedom). *We restrict our attention to the fragment without recursive types. For all processes* $\Psi \;;\; a_1 : A_1, \ldots, a_n : A_n \vdash P :: a_o : A_o$ *and environments* $u \in [\![\Psi]\!]$, $[\![\Psi \;;\; \Delta \vdash P :: a : A]\!]u$ *is adjointly junk-free.*

Conjecture 1 so far holds for the fragment containing (Fwd) (with a different junk-free interpretation of the rule), (Cut), unit, and shifts.

CHAPTER 5

Core Proposed Work

This chapter proposes work that should form the remainder of the core contributions of my thesis.

5.1. Soundness and Adequacy

The operational behaviour of session-typed languages is typically specified by a substructural operational semantics [Sim12], a form of multiset rewriting [CS09]. These operational semantics motivated the denotational semantics in section 3.1. It is incumbent upon us to show that the denotational semantics faithfully reflects the operational semantics of our languages. To make this formal, we need to show that our denotational semantics is *sound* and *adequate* relative to an appropriate operational notion of observation. By soundness, we mean that observationally equivalent processes should be denotationally equivalent. By adequacy, we mean that denotationally equivalent processes are observationally equivalent.

5.1.1. Operational notions of observation. Our definitions of soundness and adequacy presuppose that we have fixed an operational notion of observation. We briefly consider existing operational notions of observation.

Morris-style contextual equivalence [Mor68] is not a meaningful notion of equivalence in our setting. This is because we care primarily about how a process interacts with its environment, i.e., the messages it sends and receives, rather than whether or not it terminates. To illustrate this, let $\mathsf{B} = \rho\beta.\, \oplus\{\mathtt{l} : \beta\}$ and consider the recursive process `cp`:

```
x : B |- cp :: y : B
y <- cp <- x = case x { l => y.l; y <- cp <- x }
```

The right hand side of the second line is the body of the process `cp`, while the left hand side binds the channel names y and x in the body. We use the syntactic sugar `y <- cp <-x` for the tail call `t <- cp <- x; y <- t`. We will use repeatedly use this syntactic sugar in the examples below. Compare `cp` to the forwarding process:

```
x : B |- y <- x :: y : B
```

Under standard operational semantics, the forwarding process immediately terminates after globally identifying the channels x and y, while the process `cp` never terminates. However, they have identical interactions with their environments: whenever the label `l` arrives on x, it is immediately forwarded to y.

Pérez et al. [Pér+14] introduced linear logical relations for session-typed processes. Toninho [Ton15, Chapter 6] gives an account for session-typed processes with corecursion and coinductive session types. Neither account supports general recursion. To address this, Toninho [Ton15, pp. 137f.] proposes using step-indexed logical relations, but concedes that this approach is technically challenging.

Wadler [Wad14] gave a computational interpretation to classical linear logic called classical processes (CP). Though CP had a reduction relation semantics, it did not have an obvious notion of observation. One difficulty in giving CP processes such a notion is that communication in CP is synchronous, and CP processes block if they try to communicate along unconnected channels. However, if you give a CP process a partner to communicate with, then you cannot observe their communications. To address these issues, Atkey [Atk17] introduced observed communications

semantics. Atkey's solution was to define a new operational semantics for CP in which an external observer can see the messages exchanged on channels. He then showed that his relational denotational semantics was sound and adequate relative to his observed communication semantics.

5.1.2. Standard operational semantics. As described above, operational semantics for SILL-based languages are given by substructural operational semantics. These substructural operational semantics operate on a collection of linear (ephemeral) and persistent judgments, called a **configuration**. The linear judgment $\mathsf{proc}(c, P)$ describes a process P offering a service on channel c. The linear judgment $\mathsf{msg}(c, m)$ describes a message m on channel c. The persistent judgment $\mathbf{eval}(M, v)$ captures that the functional term M evaluates to the value v.

To run a process $\cdot\ ;\ \Delta \vdash P :: c : A$, we first form the **initial configuration** whose sole judgments are $\mathsf{proc}(c, P)$, and $\mathbf{eval}(M, v)$ for all functional terms M and v such that M evaluates to v. Transitions are given by rules of the form $\mathcal{C} \to \mathcal{C}'$, where $\mathcal{C}$ and $\mathcal{C}'$ are configurations. Given a configuration $\mathcal{C}, \mathcal{D}$, executing the aforementioned rule produces the configuration $\mathcal{C}', \mathcal{D}$.

In our asynchronous setting, processes can send messages even if they have no communication partner. For example, the close c process sends the close message $*$ on c and terminates. This is captured by the rule:

$$\mathsf{proc}(c, \mathsf{close}\ c) \to \mathsf{msg}(c, *). \tag{18}$$

Its client $\mathsf{wait}\ c; P$ waits until it receives a close message and then continues with the continuation process P:

$$\mathsf{msg}(c, *), \mathsf{proc}(d, \mathsf{wait}\ c; P) \to \mathsf{proc}(d, P). \tag{19}$$

Messages sent on different channels are not ordered. However, messages sent on the same channel must be received in the order in which they were sent. To capture channels' queue-like structure, we use a fresh continuation channel for subsequent communication. This is captured by messages of the form $m; c \leftarrow d$, where m is the data sent on c and d is the channel on which future communications will be sent. As a result, we can imagine the configuration $\mathsf{msg}(c, m_1; c \leftarrow d), \mathsf{msg}(d, m_2; d \leftarrow e), \mathsf{msg}(e, m_3; e \leftarrow f)$ as encoding the queue of messages m_1, m_2, m_3, where m_1 was sent first.

We illustrate this pattern with the rules for internal choice. Here, we send a label l_j on channel c, and communication continues on channel d:

$$\mathsf{proc}(c, c.l_j; P), \to \mathsf{msg}(c, c.l_j; c \leftarrow d), \mathsf{proc}(d, [d/c]P) \quad (d \text{ fresh}) \tag{20}$$

Receiving a label message causes a case process to select the corresponding branch. We update the branch with the new channel name for the continuation.

$$\mathsf{msg}(c, c.l_j; c \leftarrow d), \mathsf{proc}(b, \mathsf{case}\ c\ (l_i \Rightarrow P_i)_{i \in I}) \to \mathsf{proc}(b, [d/c]P_j) \tag{21}$$

In the asynchronous setting, the cut rule spawns two new processes that communicate along a fresh channel. It does not depend on the structure of the composed processes (cf. (1) in the synchronous setting of chapter 2).

$$\mathsf{proc}(c, a \leftarrow P;\ Q) \to \mathsf{proc}(d, [d/a]P), \mathsf{proc}(c, d/a]Q) \quad (d \text{ fresh}) \tag{22}$$

Sending and receiving channels is captured by substitution of channel names and poses no difficulty:

$$\mathsf{proc}(a, \mathsf{send}\ a\ b;\ P) \to \mathsf{proc}(d, [d/a]P), \mathsf{msg}(a, \mathsf{send}\ a\ b;\ a \leftarrow d) \quad (d \text{ fresh}) \tag{23}$$

$$\mathsf{msg}(a, \mathsf{send}\ a\ b;\ a \leftarrow d), \mathsf{proc}(c, e \leftarrow \mathsf{recv}\ a;\ Q) \to \mathsf{proc}(c, [b, d/e, a]Q) \tag{24}$$

5.1.3. A conjectural sketch of our approach. We give an operational notion of observation to polarized SILL. Because our ultimate goal is to relate our denotational semantics to existing operational semantics, we must define an operational notion observation that faithfully captures the existing operational semantics. To do so, we define a variant of the above substructural operational semantics. We then use the ideas underlying Atkey's [Atk17] observed communication semantics to extract an observation from an execution trace for a process.

To simplify the definition of observations, we modify the syntax for processes to keep track of types when they cannot be easily inferred. For example, the (Cut) rule becomes:

$$\frac{\Psi\ ;\ \Delta_1 \vdash P :: a : A \quad \Psi\ ;\ a : A, \Delta_2 \vdash Q :: c : C}{\Psi\ ;\ \Delta_1, \Delta_2 \vdash a : A \leftarrow P;\ Q :: c : C}\ (\textsc{Cut})$$

The transition rules are as in the usual substructural operational semantics, except that we track channels' types. The persistent judgment $\mathbf{type}(c : A)$ means that the channel c has type A. The transition steps then become:

$$(25)\quad \mathsf{proc}(c, c.l_j; P), \mathbf{type}(c : \oplus\{l_i : A_i\}) \to \mathsf{msg}(c, c.l_j; c \leftarrow d), \mathsf{proc}(d, [d/c]P), \mathbf{type}(d : A_j) \quad (d \text{ fresh})$$

$$(26)\quad \mathsf{proc}(c, a : A \leftarrow P;\ Q) \to \mathsf{proc}(d, [d/a]P), \mathsf{proc}(c, [d/a]Q), \mathbf{type}(d : A) \quad (d \text{ fresh})$$

$$(27)\quad \mathsf{proc}(a, \mathsf{send}\ a\ b;\ P), \mathbf{type}(a : A \otimes B) \to \mathsf{proc}(d, [d/a]P), \mathsf{msg}(a, \mathsf{send}\ a\ b;\ a \leftarrow d), \mathbf{type}(d : B) \quad (d \text{ fresh})$$

We allow for multiple transition steps to be taken in parallel in an execution. To ensure the following definition as an invariant when defining execution traces to ensure that fresh channels names do not clash. We say that a configuration $\mathcal{C}$ is **agreeable** if

(1) the multiset $\mathcal{C}$ is actually a set, and
(2) for each channel name c, if $\mathbf{type}(c : A)$ and $\mathbf{type}(c : B)$ appear in $\mathcal{C}$, then $A = B$.

Let $\mathcal{C}_0, \ldots, \mathcal{C}_n, \mathcal{C}_{n+1}$ and $\mathcal{C}'_0, \ldots, \mathcal{C}'_n, \mathcal{C}_{n+1}$ be two agreeable configurations. Assume that $\mathcal{C}_i \to \mathcal{C}'_i$ for $0 \le i \le n$. An **execution step** is a transition

$$\mathcal{C}_0, \ldots, \mathcal{C}_n, \mathcal{C}_{n+1} \to_e \mathcal{C}'_0, \ldots, \mathcal{C}'_n, \mathcal{C}_{n+1}$$

We call $\mathcal{C}_0, \ldots, \mathcal{C}_n$ the *active* subconfigurations and $\mathcal{C}_{n+1}$ the *stationary* subconfiguration. We typically write $\mathcal{S}$ for stationary subconfigurations.

An **execution trace** $\{\mathcal{C}_i\}_i$ is a (potentially empty or infinite) maximal sequence of execution steps $\mathcal{C}_0 \to_e \mathcal{C}_1 \to_e \cdots$. We identify execution traces up-to renaming of freshly-generated channels. Note that the definition of execution step builds agreeability into each of the configurations. We write $\mathcal{S}_i$ for the stationary subconfiguration in the execution step $\mathcal{C}_i \to_e \mathcal{C}_{i+1}$.

An execution trace $\{\mathcal{C}_i\}_i$ is **fair** if for all i, whenever $\mathcal{D}$ is a subconfiguration of $\mathcal{S}_i$ and $\mathcal{D} \to \mathcal{E}$, there exists a $k > i$ such that

- $\mathcal{C}_k = \mathcal{D}_0, \ldots, \mathcal{D}_n, \mathcal{S}_k$ for $n \ge 0$, and
- $\mathcal{D}$ is $\mathcal{D}_j$ for some $0 \le j \le n$.

Fairness ensures that all processes that can take an action eventually do so.

The **initial configuration** of a process $\cdot\ ;\ c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ is the configuration $\mathsf{proc}(c, P), \mathbf{type}(c_0 : A_0), \ldots, \mathbf{type}(c_n : A_n)$, along with all valid $\mathbf{eval}(M, v)$ judgments. A **fair execution** of $\cdot\ ;\ \Delta \vdash P :: c : A$ is a fair execution trace from the process's initial configuration.

5.1.3.1. *Observations from executions.* We colour-code the modes of judgments, where inputs to a judgment are in blue and outputs are in red.

Consider an execution $T = \{\mathcal{C}_i\}_i$. Let $\mathcal{T}$ be the set-theoretic union of the $\mathcal{C}_i$, that is, $x \in \mathcal{T}$ if and only if $x \in \mathcal{C}_i$ for some i.

Write $T \vdash c : A$ if $\mathbf{type}(c : A)$ appears in $\mathcal{C}_i$ for some i. This is conjectured to be a well-defined function.

We use the judgment $T \rightsquigarrow v \mathbin{/} c$ to mean that we observed a communication v on channel c during T. This judgment is coinductively defined by the following rules.

Subject to the side condition that $\mathsf{msg}(c, m)$ is not in $\mathcal{T}$, we have the axiom

$$\frac{}{T \rightsquigarrow \bot \mathbin{/} c}$$

It tells us that we observed no communications on the live channel c. We observe close messages directly:

$$\frac{\mathsf{msg}(c, *) \in \mathcal{T}}{T \rightsquigarrow * \mathbin{/} c}$$

We observe label transmission as labelling observations on the continuation channel:

$$\frac{\mathsf{msg}(c, c.l; c \leftarrow d) \in \mathcal{T} \quad T \rightsquigarrow v \mathbin{/} d}{T \rightsquigarrow (l, v) \mathbin{/} c}$$

We observe forwarding as copying between channels:

$$\frac{\mathsf{msg}(c, c \leftarrow d) \in \mathcal{T} \quad T \rightsquigarrow v \mathbin{/} d}{T \rightsquigarrow v \mathbin{/} c}$$

We observe channel transmission as pairing:

$$\frac{\mathsf{msg}(c, \mathsf{send}\ c\ a;\ c \leftarrow d) \in \mathcal{T} \quad T \rightsquigarrow u \mathbin{/} a \quad T \rightsquigarrow v \mathbin{/} d}{T \rightsquigarrow (u, v) \mathbin{/} c}$$

Conjecture 2. *The judgment $T \rightsquigarrow v \mathbin{/} c$ defines a function from executions and channel names to communications. Explicitly, if $T \rightsquigarrow v \mathbin{/} c$ and $T \rightsquigarrow w \mathbin{/} c$, then $v = w$.*

We write $T \rightsquigarrow v \in A \mathbin{/} c$ whenever $T \vdash c : A$ and $T \rightsquigarrow v \mathbin{/} c$. In this case, we expect v to live in $[\![A]\!]$, where $[\![A]\!]$ is the canonical interpretation of A described in section 4.2.

Conjecture 3. *Let T be a fair execution of $\cdot\ ;\ c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. For all $0 \le i \le n$, $T \rightsquigarrow v_i \in A_i \mathbin{/} c_i$ for some unique $v_i \in [\![A_i]\!]$.*

The following conjecture captures the confluence property typically enjoyed by SILL-style languages:

Conjecture 4. *Let T and R be a fair executions of $\cdot\ ; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. For all $0 \le i \le n$, if $T \rightsquigarrow v_i \in A_i \mathbin{/} c_i$ and $R \rightsquigarrow w_i \in A_i \mathbin{/} c_i$, then $v_i = w_i$.*

We use conjecture 3 to define the observation $[\![T]\!]$ induced by a fair execution T of a process $\cdot\ ;\ c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. It is the tuple $(c_i : v_i)_{0 \le i \le n} \in [\![c_0 : A_0, \ldots, c_n : A_n]\!]$ where $T \rightsquigarrow v_i \mathbin{/} c_i$ for $0 \le i \le n$. By conjecture 4, we have $\wr \cdot\ ; \Delta \vdash P :: c : A \wr = [\![T]\!]$ for all fair executions T of P. Every process has a fair execution $\mathcal{F}$. The **operational observation** $\wr \cdot\ ; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 \wr$ of $\cdot\ ; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ is then the element of $[\![c_0 : A_0, \ldots, c_n : A_n]\!]$ given by

$$\wr \cdot\ ; \Delta \vdash P :: c : A \wr = [\![\mathcal{F}]\!].$$

5.1.3.2. *Relating operational observations and denotations.* We relate the operational and denotational sides using conjectures 5 and 6. Conjecture 5 tells us that if we observe a bidirectional communication on the operational side, we can split it up into unidirectional communications that are related by the denotation of the process.

Conjecture 5. *Assume P is a process satisfying $\cdot\ ;\ c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$. Let $(c_i : v_i)_{0 \le i \le n} \in \wr \cdot\ ; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 \wr$ be arbitrary. Let $v_i^p = (\!|A_i|\!)^p(v_i)$ for $p \in \{-, +\}$ be the polarized aspects of v_i. Then*

$$[\![\cdot\ ; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0]\!] \bot (c_1^+ : v_1^+, \ldots, c_n^+ : v_n^+, c_0^- : v_0^-)$$
$$= (c_1^- : v_1^-, \ldots, c_n^- : v_n^-, c_0^+ : v_0^+).$$

Conjecture 6 tells us that if the denotation of a process produces an output in the absence of input, then this input/output can be reassembled into a bidirectional communications that can be observed operationally. We write merge_{A_i} for the projection associated with the embedding $\langle (\!| A_i |\!)^-, (\!| A_i |\!)^+ \rangle$ from proposition 12.

Conjecture 6. *If* $\cdot\,;\, c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ *and*

$$\begin{aligned}&[\![\cdot\,;\, c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0]\!]\bot(c_1^+ : \bot, \ldots, c_n^+ : \bot, c_0^- : \bot)\\ &= (c_1^- : v_1^-, \ldots, c_n^- : v_n^-, c_0^+ : v_0^+),\end{aligned}$$

then

$$\left(\prod_{0\le i\le n} \mathsf{merge}_{A_i}\right)(c_i^- : v_i^-, c_i^+ : v_i^+)_{0\le i\le n} \in \wr\cdot\,;\, c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 \wr.$$

It might be interesting to generalize conjecture 6 to arbitrary input. To do so, we would need to translate input from the environment on the denotational side to message judgments in the initial configuration on the operational side.

5.1.3.3. *Observational equivalence.* An **observation context** $\cdot\,;\,\Delta \vdash C[\cdot]^{\Delta'}_{a:A} :: c : C$ is a context derived using the process typing rules of fig. 3.2, plus exactly one instance of the axiom

$$\frac{}{\cdot\,;\,\Delta' \vdash [\cdot]^{\Delta'}_{a:A} :: a : A}\ (\textsc{Hole})$$

Given a context $\cdot\,;\,\Delta \vdash C[\cdot]^{\Delta'}_{a:A} :: c : C$ and a process $\cdot\,;\,\Delta' \vdash P :: a : A$, we let $C[P]$ be the result of "plugging" P into the hole, that is, of replacing the axiom (Hole) by the derivation $\Delta' \vdash P :: a : A$ in the derivation $\Delta \vdash C[\cdot]^{\Delta'}_{a:A} :: c : C$.

We say that processes $\cdot\,;\,\Delta \vdash P :: c : C$ and $\cdot\,;\,\Delta \vdash Q :: c : C$ are **observationally equivalent**, $P \approx Q$, if $\wr\cdot\,;\,\Delta' \vdash C[P] :: b : B\wr = \wr\cdot\,;\,\Delta' \vdash C[Q] :: b : B\wr$ for all observation contexts $\cdot\,;\,\Delta' \vdash C[\cdot]^{\Delta}_{c:C} :: b : B$.

5.1.3.4. *Statements of soundness and adequacy.* Processes $\Psi\,;\,\Delta \vdash P :: c : C$ and $\Psi\,;\,\Delta \vdash Q :: c : C$ are **denotationally equivalent**, $P \equiv Q$, if $[\![\Psi\,;\,\Delta \vdash P :: c : C]\!] = [\![\Psi\,;\,\Delta \vdash Q :: c : C]\!]$.

Conjecture 7 (Soundness). *For all processes* $\cdot\,;\,\Delta \vdash P :: c : A$ *and* $\cdot\,;\,\Delta \vdash Q :: c : A$*, if* $P \approx Q$*, then* $P \equiv Q$.

Conjecture 8 (Adequacy). *For all processes* $\cdot\,;\,\Delta \vdash P :: c : A$ *and* $\cdot\,;\,\Delta \vdash Q :: c : A$*, if* $P \equiv Q$*, then* $P \approx Q$.

5.2. Domain Semantics for Adjoint Logic

Adjoint logic gives a framework for conservatively combining multiple intuitionistic logics with varying structural properties [Pru+18]. Computational interpretations of adjoint logic uniformly combine message-passing concurrency, shared-memory functionality, and sequential computation [PP19b]. The message-passing interpretations permit communication patterns not possible in languages like SILL that use binary session types [PP19a]. These include *multicast*, i.e., sending one message to multiple clients, and replicable services, where a service replicates itself on-demand to handle requests from multiple clients. Giving these computational interpretations a denotational semantics provides all of the benefits described in chapter 1.

5.2.1. Related work. Adjoint logic was first defined in Reed's [Ree09] unpublished note. It is parametrized by a preorder M of "modes (of truth)". Adjoint logic combines different logics by stratifying their propositions across modes. Each mode q has an associated set $\sigma(q)$ of structural properties satisfied by judgments at that mode. For cut-elimination to hold, σ must be monotone. Propositions can be shuttled across M-related modes $p \le q$ using an adjoint pair $F_{q\ge p} \dashv U_{p\le q}$ of shift operators. A proof of a judgments at mode p can only depend on judgments at modes $q \ge p$.

Benton's LNL [Ben94; Ben95] is a prototpical example of an adjoint logic. LNL combines linear and non-linear logic using two modes. The linear mode L satisfies only exchange, while

the non-linear mode NL satisfies weakening and contraction. The ordering L < NL ensures that a proof of a non-linear proposition can depend only on other non-linear propositions.

Recall that in Reed's definition, there exists exactly one adjoint pair $F_{q \geq p} \dashv U_{p \leq q}$ for each $p \leq q$. Various applications require weakening this to allow for multiple unrelated adjunctions between modes. To accomplish this, Licata and Shulman [LS16] generalized adjoint logic so that it is parametrized by a 2-category of modes instead of a preorder of modes. They show that their generalization is sound and complete relative to a categorical semantics. Licata, Shulman, and Riley [LSR17a; LSR17b] extend this work to capture restrictions enforced by more general context structures such as trees and lists.

Pruiksma et al. [Pru+18] give three equivalent formulations of adjoint logic. The first is a variant of Reed's [Ree09] with explicit structural rules. The second makes the structural rules implicit. The third formulation is a polarized and focused presentation of adjoint logic.

Pfenning and Griffith [PG15] gave a message-passing interpretation to an adjoint logic with three modes of truth L < F < U. The linear layer L satisfied only the exchange rule. The affine layer F additionally satisfied weakening. The unrestricted layer U satisfied exchange, weakening, and contraction. Each layer gave rise to different computational behaviours. For example, linear propositions gave rise to the usual session-typed channels, while channels corresponding to affine proposition could be discarded. Pruiksma and Pfenning [PP19a] generalized this work to give a message-passing interpretation to full adjoint logic.

Pruiksma and Pfenning [PP19b] explores the consequences of changing to ordinary sequent calculus to the semi-axiomatic sequent calculus. They give the semi-axiomatic sequent calculus a message-passing interpretation, a shared-memory interpretation, and a sequential semantics. They use adjoint logic to coherently combine these three interpretations. They then give a rational reconstruction of SILL (albeit with a linear functional layer) and of linear futures.

5.2.2. Contribution. As a first step, I propose extending the domain semantics of section 3.1 to the message-passing interpretation of adjoint logic given by Pfenning and Griffith [PG15]. Once I have determined the ideas required to handle this limited setting, I propose extending the work to handle the interpretations of full adjoint logic [PP19a; PP19b].

5.3. Semantics for Dependent Session Types

SILL allows processes to send functional values of type τ over channels of type $\tau \wedge A$ and $\tau \supset A$. These protocols simply specify the type of values transmitted, but say nothing about the relationships between transmitted values. However, real-world protocols often require these kinds of specifications. Consider, for example, the 3-way handshake [Tom75; SD78; RFC793] used to negotiate a TCP connection. The first host A chooses a random value x and sends it to its partner B. The host B chooses a random value y and sends the tuple $(x+1, y)$ to A. The host A then sends the tuple $(x+1, y+1)$ back to B, thereby establishing the connection. To fully specify this protocol in a session-typed setting, types must refer to values, i.e., we must use a dependent type theory.

Dependent SILL [TCP11; CPT12] replaces the types $\tau \wedge A$ and $\tau \supset A$ from fig. 3.3 with their dependent counterparts $\exists x : \tau.A$ and $\forall x : \tau.A$, whose corresponding processes are formed by the following four rules:

$$\frac{\Psi \Vdash M : \tau \quad \Psi\,;\,\Delta \vdash P :: a : A}{\Psi\,;\,\Delta \vdash \_ \leftarrow \mathsf{output}\ a\ M; P :: a : \exists x : \tau.A}\ (\exists R)$$

$$\frac{\Psi, x : \tau\,;\,\Delta, a : A \vdash Q :: c : C}{\Psi\,;\,\Delta, a : \exists x : \tau.A \vdash x \leftarrow \mathsf{input}\ a; Q :: c : C}\ (\exists L)$$

$$\frac{\Psi, x : \tau\,;\,\Delta \vdash Q :: a : A}{\Psi\,;\,\Delta \vdash x \leftarrow \mathsf{input}\ a; Q :: a : \forall x : \tau.A}\ (\forall R)$$

$$\frac{\Psi \Vdash M : \tau \quad \Psi\,;\,\Delta, a : A \vdash P :: c : C}{\Psi\,;\,\Delta, a : \forall x : \tau.A \vdash \_ \leftarrow \mathsf{output}\ a\ M; P :: c : C}\ (\forall L)$$

Assuming that the underlying functional language has been suitably extended to support dependent types (including identity types), we can then capture the 3-way handshake protocol using the type:

$$\exists a : \mathsf{nat}.\forall b : \mathsf{nat} \times \mathsf{nat}.\forall p : (\pi_1 b = a + 1).$$
$$\exists c : \mathsf{nat} \times \mathsf{nat}.\exists p' : (\pi_1 c = a + 1 \wedge \pi_2 c = \pi_2 b + 1).\mathbf{1}$$

Let H abbreviate the above type, then the following process provides a service of type H:

```
|- A :: h : H
h <- A = _ <- output h (rand ());    % send x
         (b1, b2) <- input  h;       % receive (z, y)
         p <- input h;               % receive proof of z = x + 1
         _ <- output h (b1, b2 + 1); % send (x + 1, y + 1)
         _ <- output h (p, refl);    % send proof that we did so
         close h                     % end session
```

A service providing or using the protocol H incurs a communication overhead compared to its (under-specified) non-dependent form

$$\tau \wedge ((\tau \times \tau) \supset ((\tau \times \tau) \wedge \mathbf{1})).$$

Indeed, the dependent form incurs communication overhead by transmitting proof terms, even though they are not computationally relevant. The *proof irrelevance* [Pfe01; TCP11, § 4; Gom19, pp. 59ff.] technique has been used to reduce or eliminate this overhead by specifying proofs that can be erased from programs at run-time. Because they are erased, they are never transmitted, thereby reducing communication overhead. Proof irrelevance is implemented using a collection of typing rules that ensure that the irrelevant proofs exist and that they can be erased without affecting computation.

Concretely, given a functional type τ, the (functional) **bracket type** $[\tau]$ is inhabited by terms $M : \tau$ that can be erased before runtime without changing the meaning of the program [AB04]. To make this precise, we first allow functional contexts to also include assumptions $x \div \tau$ that mean the variable x abstracts over computationally irrelevant values of type τ. So that irrelevant terms can depend on irrelevant values, we define a promotion operator on functional contexts that promotes irrelevant hypotheses into ordinary ones:

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

The introduction and elimination rules for bracketed types are:

$$\frac{\Psi^{\oplus} \Vdash M : \tau}{\Psi \Vdash [M] : [\tau]}\ ([]\,\mathrm{I}) \qquad \frac{\Psi \Vdash M : [\tau] \quad \Psi, x \div \tau \Vdash N : \sigma}{\Psi \Vdash \mathsf{let}\ [x]\ =\ M\ \mathsf{in}\ N : \sigma}\ ([]\mathrm{E})$$

Given a suitable erasure operation [PCT11, Definition 6], the above rules ensure that erasure do not meaningfully affect the computational behaviour of terms [PCT11, Theorem 2].

Applying the proof irrelevance technique to the above 3-way handshake example, the protocol H becomes:

$$\exists a : \mathsf{nat}.\forall b : \mathsf{nat} \times \mathsf{nat}.\forall p : [\pi_1 b = a + 1].$$
$$\exists c : \mathsf{nat} \times \mathsf{nat}.\exists p' : [\pi_1 c = a + 1 \wedge \pi_2 c = \pi_2 b + 1].\mathbf{1}$$

So far we have only considered dependency on functional values. Toninho and Yoshida [TY18a; TY18b] extend a SILL-style language with protocols that depend on transmitted data. To motivate this dependency, they give the following protocol as an example. Consider a server that receives a boolean value true or false. If it receives true, then the service sends a natural number and

terminates. If it receives false, then the service sends a boolean and then terminates. With existing techniques, this protocol can be approximated by following session type:

$$\mathsf{Bool} \supset \oplus\{\mathtt{t} : \mathsf{Nat} \wedge \mathbf{1}, \mathtt{f} : \mathsf{Bool} \wedge \mathbf{1}\}.$$

However, this type does not prevent the server from taking the `f` branch even if it received true. By extending the type system with dependent pattern matching, they can encode the intended protocol as the type

$$\forall x : \mathsf{Bool}.\mathsf{if}\ x\ (\mathsf{Nat} \wedge \mathbf{1})(\mathsf{Bool} \wedge \mathbf{1}).$$

More generally, their language supports type-level λ-abstraction over terms and session types.

5.3.1. Related work. Pfenning [Pfe01] introduced a dependent type theory that captures proof irrelevance. Toninho, Caires, and Pfenning [TCP11] build on this work to give the first dependent-session-typed language based on intuitionistic linear logic. They discussed using proof irrelevance to reduce the communication overhead caused by transmitting proof objects. Their work assumes that you can define type families in the underlying term language. Pfenning, Caires, and Toninho [PCT11] use dependent session types to model various aspects of systems employing certified code. They propose two techniques to reduce the overhead of communicating and working with proof objects. The first is proof irrelevance, and they give a theorem stating that erasures do not affect the computational behaviour of terms. The second technique involves delegating proof verification to a trusted verifier. This verifier returns a signed digital certificate attesting that the proof is correct. Caires, Pfenning, and Toninho [CPT12] survey these two papers.

Griffith and Gunter [GG13] give a refinement session type system for the π-calculus. They also give an inference algorithm for their type system.

Gommerstadt [Gom19, Chapter 6] presents monitoring for dependently-session-typed contracts in the presence of proof irrelevance. Monitors need to check that transmitted values satisfy the propositions specified by the protocol even if the proof objects are computationally irrelevant. To do so, a proof object may still need to be transmitted if the proposition does not belong to a decidable fragment.

Muijnck-Hughes, Brady, and Vanderbauwhede [MBV19] give an embedding of dependent multi-party session types in the dependently-typed language Idris. They illustrate practical applications of dependently-session-typed languages by implementing a series of examples, including the above 3-way handshake, an "echo" server, and a server performing arithmetic operations.

5.3.2. Proposed work. I propose to first extend my semantics from section 3.1 to handle types depending on functional terms. This extension should also support proof irrelevance. I conjecture that I will need to use fibrations [Jac99], a standard technique for semantics of dependent type theories. Once I can capture dependencies on functional terms, I propose extending the semantics to the language given by Toninho and Yoshida [TY18a; TY18b].

5.4. Applications

To evaluate the tractability of my semantics, I propose applying it to the problems of *monitoring* and *program equivalence*.

5.4.1. Monitoring. Concurrent contracts [GJP18; Gom19] enforce communication protocols at run-time. Concurrent contracts are specified by a collection of session-typed partial identity processes called "monitors". A monitor observes communications along a channel and raises a global alarm if ever it witnesses a communication that violates the prescribed protocol.

To be a monitor, a process must be **transparent** [GJP18, § 4.4], i.e., observationally equivalent to a partial identity process. Formally defining transparency requires a complex bisimulation-style construction. Showing that a process satisfies this definition is non-trivial. I naïvely conjecture that domain-theoretic techniques will simplify the situation. Indeed, I conjecture that a process $\Psi\ ;\ b : B \vdash P :: a : B$ is transparent if and only if for all $u \in [\![\Psi]\!]$, $[\![\Psi\ ;\ b : B \vdash P :: a : B]\!]u \sqsubseteq [\![\Psi\ ;\ b : B \vdash a \leftarrow b :: a : B]\!]u$, where we recall that $[\![\Psi\ ;\ b : B \vdash a \leftarrow b :: a : B]\!]u$ is the identity up-to

renaming of channels. I conjecture that this conceptually simpler definition simplifies the task of checking that a process is indeed a monitor.

5.4.2. Process Equivalences. Proving program equivalences is in the best of cases a thorny problem. Recall that denotational semantics automatically give a notion of program equivalence. To show that this equivalence is a tractable solution to this problem, I propose proving several non-trivial program equivalences. For the sake of exposition, I assume that our recursive types are equirecursive instead of isorecursive (see section 6.1 for details).

5.4.2.1. *Bit counters.* We consider counter processes that receive `inc` and `val` messages. When they receive a `inc` message, they increment an internal counter. When they receive a `val` message, they send back the total number of `inc` messages received as a sequence of bits, where the least significant bit is sent first. We specify a sequence of bits by the following protocol:

$$\mathsf{bits} = \oplus\{\mathtt{b0} : \mathsf{bits}, \mathtt{b1} : \mathsf{bits}, \$: \mathbf{1}\}.$$

In this protocol, `$` denotes the end of the stream. The counter protocol is:

$$\mathsf{ctr} = \&\{\mathtt{inc} : \mathsf{ctr}, \mathtt{val} : \mathsf{bits}\}.$$

We consider two implementations of a counter, taken from [PK18, pp. L21.10ff.] and [DHP18].

In the first case, we implement a counter `ctr1` using an object-oriented approach, where we store the count as a chain of processes, each representing a single bit in the binary representation of the count. The `zero` and `one` processes respectively store the zero and one bits. The `empty` process acts as the end of the chain of bits.

```
a :: ctr |- zero :: c : ctr
c <- zero <- a = case c { inc => % incrementing a zero bit turns it
                                 % into a one bit
                                 c <- one <- a
                       | val => % send the zero bit:
                                 c.b0;
                                 % forward the count from the rest
                                 % of the chain of processes:
                                 a.val; c <- a }
a :: ctr |- one :: c : ctr
c <- one <- a = case c { inc => % incrementing a one bit turns it
                                % to a zero. Send inc to a to carry.
                                a.inc; c <- zero <- a
                       | val => c.b1; a.val; c <- a }
|- empty :: c : ctr
c <- empty = case c { inc => a <- empty; c <- one <- a
                    | val => c.$; close c }
|- ctr1 :: c : ctr
c <- ctr1 = e <- empty; c <- zero <- e
```

In the second case, every time the counter receives the `inc` label, it spawns a process `succ` that increments the bit stream storing the current count.

```
|- zero' :: b : bits
b <- zero' = b.b0; b.$; close b
a :: bits |- succ :: b : bits
b <- succ <- a = case a { $  => a.b1; a.$; b <- a
                        | b0 => a.b1; b <- a
                        | b1 => a.b0; b <- succ <- a }
b : bits |- ctr2' :: c : ctr
c <- ctr2' <- b = case c { inc => a <- succ <- b;
                                  c <- ctr2' <- a
```

```
                              | val => c <- b }
|- ctr2 :: c : ctr
c <- ctr2 = z <- zero';
            c <- ctr2' <- z
```

We expect that the processes `ctr1` and `ctr2` should denote the same function.

5.4.2.2. *Sieve of Eratosthenes.* Assume that our functional language has conditionals, natural numbers nat, and a test "`m | n`" that passes if and only if m divides n. Let the type of streams of natural numbers be given by

$$\mathsf{ns} = \mathsf{nat} \wedge \mathsf{ns}.$$

Given a natural number n, the process `filterMultiples` n forwards all natural numbers on channel i to o except for multiples of n.

```
n : nat; i : ns |- filterMultiples n :: o : ns
o <- filterMultiples n <- i =
  m <- recv i;
  if (n | m) then
    % do not send m if divisible by n
    o <- filterMultiples n <- i
  else
    % send m otherwise
    send o m;
    o <- filterMultiples n <- i
```

We can then use this process to write the sieve process:

```
i : ns |- sieve :: o : ns
o <- sieve <- i =
  p <- recv i;                % receive the next prime on i
  t <- filterMultiples p <- i; % filter out its multiples from i
  o <- sieve <- t             % recurse with the sieve
```

The following process produces a stream of natural numbers starting from n:

```
|- natsFrom n :: o : ns
o <- natsFrom n =
  send o n;
  o <- natsFrom (n + 1)
```

Finally, we would like to show that the following process sends a natural number n on the stream p if and only if n is prime:

```
|- primes :: p : ns
p <- primes =
  n <- natsFrom 2;
  p <- sieve <- n
```

5.4.2.3. *Queues from stacks.* A standard programming exercise is to implement a queue using two stacks. The types of stacks and queues are respectively given by

$$\tau\ \mathsf{stack} = \&\{\mathsf{push} : \tau \multimap (\tau\ \mathsf{stack}), \mathsf{pop} : \oplus\{\mathsf{item} : \tau \wedge (\tau\ \mathsf{stack}), \mathsf{err} : \mathbf{1}\}\},$$
$$\tau\ \mathsf{queue} = \&\{\mathsf{enq} : \tau \multimap (\tau\ \mathsf{queue}), \mathsf{deq} : \oplus\{\mathsf{item} : \tau \wedge (\tau\ \mathsf{queue}), \mathsf{err} : \mathbf{1}\}\}.$$

We can implement a stack as a chain of `cellS` n processes, each storing a value of type τ (we asciify τ as `T`). The empty stack is given by the process `emptyS`.

```
n : T; r : T stack |- cellS n :: s : T stack
s <- cellS n <- r = case r { push => v <- recv s;
                                     t <- cellS n <- r;
                                     s <- cellS v <- t
                           | pop => s.item;
```

```
                                    send s n;
                                    s <- r }
|- emptyS :: s : T stack
s <- emptyS = case s { push => v <- recv s;
                               e <- emptyS;
                               s <- cellS v <- e
                     | pop => s.err; close s }
```

We can similarly implement a queue as a chain of processes, each storing a value of type τ:

```
n : T; r : T queue |- cellQ n :: q : T queue
q <- cellQ n <- l = case q { enq => % receive a value v to enqueue
                                    v <- recv q;
                                    % tell the tail to prepare to
                                    % enqueue a value
                                    r.enq;
                                    % send it the value to enqueue
                                    send r v;
                                    % recurse so that n remains at
                                    % the head of the queue
                                    q <- cellQ n <- r
                          | deq => q.item;
                                    send q n;
                                    q <- r }
|- emptyQ :: q : T queue
q <- emptyQ = case q { enq => v <- recv q;
                              e <- emptyQ;
                              q <- cellQ v <- e
                     | deq => q.err; close q }
```

Alternatively, we can use two stacks. The `flip` process flips all of the elements of a stack l onto a stack r. The result is provided channel f.

```
l : T stack, r : T stack |- flip :: f : T stack
f <- flip <- l, r =
  l.pop;
  case l { err => close l; f <- r
         | item => n <- recv l;
                   r.push;
                   send r n;
                   f <- flip <- l, r }
```

The core logic of the process implementing a queue using two stacks is in the `ssq` process. It uses an "input stack" i and an "output stack" o. When a client tries to dequeue an element, it first tries to pop the element from the stack o. If o is empty, i.e., if o sends the label `err`, then `ssq` flips the input stack onto the output stack and tries again. If o is still empty, then the queue is also empty and `ssq` raises an error. When a client tries to enqueue an element, `ssq` pushes it onto the input stack.

```
i : T stack, o : T stack |- ssq :: q : T queue
q <- ssq <- i, o =
  case q { enq => v <- recv q;
                  i.push;
                  send i v;
                  q <- ssq <- i, o
         | deq => o.pop;
                  case o { item => q.item;
```

```
                    v <- recv o;
                    send q v;
                    q <- ssq <- i, o
  | err => close o;
           e <- empty;
           o <- flip <- i, e;
           o.pop;
           case o { item => q.item;
                            v <- recv o;
                            send q v;
                            i <- emptyS;
                            q <- ssq <- i, o
                  | err => q.err; q <- o }}}
```

The process emptySSQ provides an empty queue implemented using two stacks.

```
|- emptySSQ :: q : T queue
q <- emptySSQ =
  i <- emptyS;
  o <- emptyS;
  q <- ssq <- i, o
```

We expect that emptyQ and emptySSQ are equivalent.

CHAPTER 6

Optional Proposed Work

This chapter explores optional work that would be interesting to explore as support for my thesis statement.

6.1. Semantics for Equirecursion

The semantics in section 3.1 treats recursive types *isorecursively*. This means that a recursive type is taken to be isomorphic to its unfolding, and processes fold and unfold recursive types by sending fold and unfold messages. An alternate approach is to treat recursive types *equirecursively*, where we deem a recursive type to be definitionally equal to its unfolded form. To ensure that this is well defined, we require that recursive types be contractive. Recall that a type T is **contractive** [Pie02, p. 300] if for any subexpression of T of the form $\rho\alpha.\rho\alpha_1.\rho\cdots.\rho\alpha_n.A$, A is not α. The judgments $A\ \mathsf{type}_{\mathsf{s}}^{p}$ and $A\ \mathsf{ctype}_{\mathsf{s}}^{p}$ mean A is a session type or a contractive session type with polarity p. These judgments are inductively defined by the rules in fig. 6.1. We abbreviate these judgments as $\Xi \vdash A$ where no ambiguity arises. As in the isorecursive case, we interpret judgments $\Xi \vdash A$ as functors $[\![\Xi \vdash A]\!], [\![\Xi \vdash A]\!]^p : \mathbf{M}^{\Xi} \to M$ for $p \in \{-, +\}$.

The judgment $\Xi \vdash A \equiv A'$ means that the types $\Xi \vdash A$ and $\Xi \vdash A'$ are definitionally equal. It is inductively defined by the rules in fig. 6.2. These rules can be extended so that $\equiv$ becomes a congruence relation. We interpret this judgment as a natural isomorphism $[\![\Xi \vdash A \equiv A']\!] : [\![\Xi \vdash A]\!] \cong [\![\Xi \vdash A']\!]$ between the canonical interpretations of section 4.2. Conjecture 9 shows that we can transfer these natural isomorphisms from the canonical interpretations to the polarized aspects. These natural isomorphisms are all well-behaved with regards to substitution and weakening. Let $\mathsf{merge}_{\Xi \vdash A}$ be the natural projection associated with the natural embedding $\langle (\!| \Xi \vdash A |\!)^-, (\!| \Xi \vdash A |\!)^+ \rangle$.

Conjecture 9. *If* $\psi = [\![\Xi \vdash A \equiv B]\!] : [\![\Xi \vdash A\ \mathsf{type}_{\mathsf{s}}]\!] \cong [\![\Xi \vdash B\ \mathsf{type}_{\mathsf{s}}]\!]$, *then there exist natural isomorphisms* $\psi^+ : [\![\Xi \vdash A\ \mathsf{type}_{\mathsf{s}}]\!]^+ \cong [\![\Xi \vdash B\ \mathsf{type}_{\mathsf{s}}]\!]^+$ *and* $\psi^- : [\![\Xi \vdash A\ \mathsf{type}_{\mathsf{s}}]\!]^- \cong [\![\Xi \vdash B\ \mathsf{type}_{\mathsf{s}}]\!]^-$ *that commute with the natural e-p-pairs specifying the polarized aspects, i.e., the following diagram commutes:*

$$\begin{array}{ccccc}
[\![\Xi \vdash A\ \mathsf{type}_{\mathsf{s}}]\!] & \xrightarrow{\langle (\!| A |\!)^-, (\!| A |\!)^+ \rangle} & [\![\Xi \vdash A\ \mathsf{type}_{\mathsf{s}}]\!]^- \times [\![\Xi \vdash A\ \mathsf{type}_{\mathsf{s}}]\!]^+ & \xrightarrow{\mathsf{merge}_A} & [\![\Xi \vdash A\ \mathsf{type}_{\mathsf{s}}]\!] \\
\downarrow{\scriptstyle \psi} & & \downarrow{\scriptstyle \psi^- \times \psi^+} & & \downarrow{\scriptstyle \psi} \\
[\![\Xi \vdash B\ \mathsf{type}_{\mathsf{s}}]\!] & \xrightarrow[\langle (\!| B |\!)^-, (\!| B |\!)^+ \rangle]{} & [\![\Xi \vdash B\ \mathsf{type}_{\mathsf{s}}]\!]^- \times [\![\Xi \vdash B\ \mathsf{type}_{\mathsf{s}}]\!]^+ & \xrightarrow[\mathsf{merge}_B]{} & [\![\Xi \vdash B\ \mathsf{type}_{\mathsf{s}}]\!].
\end{array}$$

We use an equivalence $\cdot \vdash A' \equiv A$ between closed types A and A' as follows:

$$\frac{\Psi\ ;\ \Delta, a : A' \vdash P :: b : B \quad \cdot \vdash A' \equiv A}{\Psi\ ;\ \Delta, a : A \vdash P :: b : B}\ (\textsc{Equiv-L})$$

$$\frac{\Psi\ ;\ \Delta \vdash P :: a : A' \quad \cdot \vdash A' \equiv A}{\Psi\ ;\ \Delta \vdash P :: a : A}\ (\textsc{Equiv-R})$$

$$\frac{}{\Xi, \alpha\ \mathsf{type}_\mathsf{s}^p \vdash \alpha\ \mathsf{type}_\mathsf{s}^p}\ (\textsc{TVar}) \qquad \frac{}{\Xi \vdash \mathbf{1}\ \mathsf{ctype}_\mathsf{s}^+}\ (\mathrm{C}\mathbf{1})$$

$$\frac{\Xi \vdash A\ \mathsf{ctype}_\mathsf{s}^p}{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^p}\ (\mathrm{TC}) \qquad \frac{\Xi, \alpha\ \mathsf{type}_\mathsf{s}^p \vdash A\ \mathsf{ctype}_\mathsf{s}^p}{\Xi \vdash \rho\alpha.A\ \mathsf{ctype}_\mathsf{s}^p}\ (\mathrm{C}\rho)$$

$$\frac{\Xi \vdash A_i\ \mathsf{type}_\mathsf{s}^+ \quad (\forall i \in I)}{\Xi \vdash \oplus\{l_i : A_i\}_{i\in I}\ \mathsf{ctype}_\mathsf{s}^+}\ (\mathrm{C}\oplus) \qquad \frac{\Xi \vdash A_i\ \mathsf{type}_\mathsf{s}^- \quad (\forall i \in I)}{\Xi \vdash \&\{l_i : A_i\}_{i\in I}\ \mathsf{ctype}_\mathsf{s}^-}\ (\mathrm{C}\&)$$

$$\frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^+ \quad \Xi \vdash B\ \mathsf{type}_\mathsf{s}^+}{\Xi \vdash A \otimes B\ \mathsf{ctype}_\mathsf{s}^+}\ (\mathrm{C}\otimes) \qquad \frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^+ \quad \Xi \vdash B\ \mathsf{type}_\mathsf{s}^-}{\Xi \vdash A \multimap B\ \mathsf{ctype}_\mathsf{s}^-}\ (\mathrm{C}\multimap)$$

$$\frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^+}{\Xi \vdash \uparrow A\ \mathsf{ctype}_\mathsf{s}^-}\ (\mathrm{C}\uparrow) \qquad \frac{\Xi \vdash A\ \mathsf{type}_\mathsf{s}^-}{\Xi \vdash \downarrow A\ \mathsf{ctype}_\mathsf{s}^+}\ (\mathrm{C}\downarrow)$$

$$\frac{\tau\ \mathsf{type}_\mathsf{f} \quad \Xi \vdash A\ \mathsf{type}_\mathsf{s}^-}{\Xi \vdash \tau \supset A\ \mathsf{ctype}_\mathsf{s}^-}\ (\mathrm{C}\supset) \qquad \frac{\tau\ \mathsf{type}_\mathsf{f} \quad \Xi \vdash A\ \mathsf{type}_\mathsf{s}^+}{\Xi \vdash \tau \wedge A\ \mathsf{ctype}_\mathsf{s}^+}\ (\mathrm{C}\wedge)$$

FIGURE 6.1. Type formation rules in the equirecursive setting

$$\frac{}{\Xi \vdash \rho\alpha.A \equiv [\rho\alpha.A/\alpha]A}\ (\text{E-}\rho)$$

$$\frac{}{\Xi \vdash A \equiv A}\ (\text{E-Refl}) \qquad \frac{\Xi \vdash A' \equiv A}{\Xi \vdash A \equiv A'}\ (\text{E-Sym}) \qquad \frac{\Xi \vdash A \equiv B \quad \Xi \vdash B \equiv C}{\Xi \vdash A \equiv C}\ (\text{E-Trans})$$

FIGURE 6.2. Definitional equality of types in the equirecursive setting

By conjecture 9, there exist natural isomorphisms $[\![A' \equiv A]\!]^p : [\![A']\!]^p \to [\![A]\!]^p$ for $p \in \{-, +\}$. The rules (Equiv-L) and (Equiv-R) are respectively interpreted as:

$$[\![\Psi\ ;\ a : A, \Delta \vdash P :: b : B]\!]u$$
$$= ((a^- : [\![A' \equiv A]\!]^-) \times \mathsf{id}) \circ [\![\Psi\ ;\ \Delta, a : A' \vdash P :: b : B]\!]u \circ \left(\left(a^+ : ([\![A' \equiv A]\!]^+)^{-1}\right) \times \mathsf{id}\right),$$
$$[\![\Psi\ ;\ \Delta \vdash P :: a : A]\!]u$$
$$= ((a^+ : [\![A' \equiv A]\!]^+) \times \mathsf{id}) \circ [\![\Psi\ ;\ \Delta \vdash P :: a : A']\!]u \circ \left(\left(a^- : ([\![A' \equiv A]\!]^-)^{-1}\right) \times \mathsf{id}\right).$$

Processes no longer have unique typing derivations because of the rules (Equiv-L) and (Equiv-R). As a result, we must show that denotations are coherent, i.e., that all derivations of the same process induce equal denotations. This remains an open problem. I propose proving this result. I also propose showing exactly how the isorecursive and equirecursive semantics are related.

6.2. Semantics for Subtyping

The semantic ideas underlying the equirecursive formulation should be easily adaptable to the study of subtyping. For the sake of illustration, let us consider choice types. For semantic reasons that will be made clear below, the subtyping relation <: needs to be polarized. When $I \subseteq J$, we expect for internal choice that

$$\oplus\{l_i : A_i\}_{i\in I} <:^+ \oplus\{l_i : A_i\}_{i\in J},$$

Indeed, any process that provides the type on the left "also provides" the type on the right. If a client accepts all of the labels on the right, then it necessarily accepts all of the labels on the left. The case for external choice is symmetric:

$$\&\{l_i : A_i\}_{i\in J} <:^- \&\{l_i : A_i\}_{i\in I}.$$

As a possible application, we would like to know whether a process providing a "smaller" interface is simulated by a process providing a larger interface. With $I \subseteq J$, consider the following

two processes:

$$\Psi\,;\,\Delta \vdash P :: z : \oplus\{l_i : A_i\}_{i\in I},$$
$$\Psi\,;\,\Delta \vdash Q :: z : \oplus\{l_i : A_i\}_{i\in J}.$$

Does Q simulate P? Formally, this corresponds to asking if $[\![\Psi\,;\,\Delta \vdash P :: z : \oplus\{l_i : A_i\}_{i\in J}]\!] \sqsubseteq [\![\Psi\,;\,\Delta \vdash Q :: z : \oplus\{l_i : A_i\}_{i\in J}]\!]$. The catch is that P and Q have different types, so we need some means of taking $[\![\Psi\,;\,\Delta \vdash P :: z : \oplus\{l_i : A_i\}_{i\in I}]\!]$ to $[\![\Psi\,;\,\Delta \vdash P :: z : \oplus\{l_i : A_i\}_{i\in J}]\!]$. To do so, I conjecture that we can follow the pattern of type equivalence and of conjecture 9.

First, we introduce judgments $\Xi \vdash A <:^p B$ for $p \in \{-,+\}$. We interpret these judgments as natural embeddings

$$[\![\Xi \vdash A <:^p B]\!] : [\![\Xi \vdash A]\!] \Rightarrow [\![\Xi \vdash B]\!].$$

We use an analog of conjecture 9 to transfer the embedding from the canonical interpretations of A and B to their polarized aspects. Here is the positive version, the negative version will be symmetric in polarities.

Conjecture 10. *If $\epsilon = [\![\Xi \vdash A <:^+ B]\!] : [\![\Xi \vdash A\ \mathsf{type_s}]\!] \Rightarrow [\![\Xi \vdash B\ \mathsf{type_s}]\!]$, then there exist natural embeddings $\epsilon^+ : [\![\Xi \vdash A\ \mathsf{type_s}]\!]^+ \Rightarrow [\![\Xi \vdash B\ \mathsf{type_s}]\!]^+$ and $\epsilon^- : [\![\Xi \vdash B\ \mathsf{type_s}]\!]^- \Rightarrow [\![\Xi \vdash A\ \mathsf{type_s}]\!]^-$ that commute with the natural e-p-pairs specifying the polarized aspects, i.e., the following diagrams commute:*

$$\begin{array}{ccc} [\![\Xi \vdash A\ \mathsf{type_s}]\!] & \xrightarrow{(\!|A|\!)^-} & [\![\Xi \vdash A\ \mathsf{type_s}]\!]^- \\ \downarrow{\scriptstyle \epsilon} & & \uparrow{\scriptstyle \epsilon^-} \\ [\![\Xi \vdash B\ \mathsf{type_s}]\!] & \xrightarrow[(\!|B|\!)^-]{} & [\![\Xi \vdash B\ \mathsf{type_s}]\!]^- \end{array} \qquad \begin{array}{ccc} [\![\Xi \vdash A\ \mathsf{type_s}]\!] & \xrightarrow{(\!|A|\!)^+} & [\![\Xi \vdash A\ \mathsf{type_s}]\!]^+ \\ \downarrow{\scriptstyle \epsilon} & & \downarrow{\scriptstyle \epsilon^+} \\ [\![\Xi \vdash B\ \mathsf{type_s}]\!] & \xrightarrow[(\!|B|\!)^+]{} & [\![\Xi \vdash B\ \mathsf{type_s}]\!]^+. \end{array}$$

To use the subtyping judgments, we extend the process typing rules to handle subtyping with the following rules for positive subtyping (and their negative counterparts):

$$\frac{\Psi\,;\,\Delta, z : B \vdash P :: x : C \quad A <:^+ B}{\Psi\,;\,\Delta, z : A \vdash P :: x : C}\ (\textsc{SubT-+-L})$$

$$\frac{\Psi\,;\,\Delta \vdash P :: x : A \quad A <:^+ B}{\Psi\,;\,\Delta \vdash P :: x : B}\ (\textsc{SubT-+-R})$$

Adding these four rules introduces the same issues as did the rules (Equiv-L) and (Equiv-R) in section 6.1: processes no longer have a unique derivation and we must show that the denotations are coherent.

Interpret (SubT-+-R) as follows:

$$\begin{aligned}&[\![\Psi\,;\,\Delta \vdash P :: x : B]\!]u \\ &\quad = (\mathsf{id} \times (x^+ : [\![\Xi \vdash A <:^+ B]\!]^+)) \circ [\![\Psi\,;\,\Delta \vdash P :: x : A]\!]u \circ (\mathsf{id} \times (x^- : [\![\Xi \vdash A <:^+ B]\!]^-))\,,\end{aligned}$$

and (SubT-+-L) as

$$\begin{aligned}&[\![\Psi\,;\,\Delta, z : A \vdash P :: x : C]\!]u \\ &\quad = (\mathsf{id} \times (z^- : [\![\Xi \vdash A <:^+ B]\!]^-)) \circ [\![\Psi\,;\,\Delta, z : B \vdash P :: x : C]\!]u \circ (\mathsf{id} \times (z^+ : [\![\Xi \vdash A <:^+ B]\!]^+))\,.\end{aligned}$$

These clauses explain why $[\![\Xi \vdash A <:^+ B]\!]^+$ and $[\![\Xi \vdash A <:^+ B]\!]^-$ go in "opposite directions".

I propose fleshing out the details of subtyping in this setting and exploring how this semantics relates to existing work on subtyping for session types by Gay and Hole [GH05], Gay and Vasconcelos [GV10], Chen et al. [Che+17], Dezani-Ciancaglini et al. [Dez+16], and Das and Pfenning [DP20].

6.3. Complete Axioms for Trace Operators

Communicating systems often involve feedback. For example, we interpreted process composition in polarized SILL using a fixed point that captured the feedback between two processes on their common channel (see interpretation (5)). Feedback-as-a-fixed-point is naturally captured by the concept of a trace [JSV96]. Intuitively, the trace $\mathsf{Tr}^X_{A,B} f : A \to B$ of a morphism $f : A \times X \to B \times X$ can be thought of as the result of the feedback loop, where the output on X is fed "back into" the input on X. Traces are explicitly given by definition 1, below.

We would like to be able to easily reason about process composition. To do so, it is helpful to have a battery of trace identities. Fortunately, traces are well-studied objects and many identities are already known. *Uniform* traces [Has97; Has99a; Has03] also give rise to an induction-flavoured proof principle [Sel99, p. 14] that can be used to show that two morphisms have the same trace. In the context of processes, this proof principle can be used to show that two process compositions have the same denotation.

In light of the above, refining our understanding of traces should help reason about the denotations of processes.

6.3.1. Background. Căzănescu and Ştefănescu [CŞ90, § 4.3] first introduced traces for symmetric monoidal categories under the name "biflow". They were interested in giving an algebraic formalism to study flowchart schemes with feedback. Traces were then independently rediscovered by Joyal, Street, and Verity [JSV96] in the setting of balanced monoidal categories. Their motivation was a generalization of traces for linear functions between finite dimensional vector spaces. This generalization is explored in greater detail by Ponto and Shulman [PS13, Example 3.1]. Below, we give the definition of traces in the setting of symmetric monoidal categories.

A **monoidal category** is a sextuple $(\mathbf{M}, \otimes, I, \lambda, \rho, \alpha, \sigma)$ satisfying the pentagon axiom [Eti+15, diagram (2.2)] and the triangle axiom [Eti+15, diagram (2.10)], where

- $\mathbf{M}$ is a category
- $\otimes : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$ is a tensor product on $\mathbf{M}$
- I is the unit of the tensor
- $\lambda : I \times A \Rightarrow A$ is a natural isomorphism witnessing that I is the left unit
- $\rho : A \otimes I \Rightarrow A$ is a natural isomorphism witnessing that I is the right unit
- $\alpha : (A \otimes B) \otimes C \Rightarrow A \otimes (B \otimes C)$ is a natural isomorphism witnessing the associativity of $\otimes$.

A monoidal category is **symmetric** if it is additionally equipped by a natural isomorphism $\sigma : A \otimes B \to B \otimes A$ satisfying various other axioms given on [BW99, p. 404]. For expository accounts of monoidal categories, see [Eti+15, Chapter 2] and [BW99, Chapter 16]

Definition 1 ([BH03, Definition 2.4]). *A* **trace** *on a symmetric monoidal category* $(\mathbf{M}, \otimes, I, \lambda, \rho, \alpha, \sigma)$ *is a family of functions*

$$\mathsf{Tr}^U_{A,B} : \mathbf{M}(A \otimes U, B \otimes U) \to \mathbf{M}(A, B)$$

satisfying the following conditions:

(1) Naturality in A (left tightening): if $f : A' \otimes U \to B \otimes U$ *and* $g : A \to A'$*, then*

$$\mathsf{Tr}^U_{A,B}(f \circ (g \otimes \mathsf{id}_U)) = \mathsf{Tr}^U_{A',B}(f) \circ g : A \to B.$$

(2) Naturality in B (right tightening): if $f : A \otimes U \to B' \otimes U$ *and* $g : B' \to B$*, then*

$$\mathsf{Tr}^U_{A,B}((g \otimes \mathsf{id}_U) \circ f) = g \circ \mathsf{Tr}^U_{A,B'}(f) : A \to B.$$

(3) Dinaturality (sliding): if $f : A \otimes U \to B \otimes V$ *and* $g : V \to U$*, then*

$$\mathsf{Tr}^U_{A,B}((\mathsf{id}_B \otimes g) \circ f) = \mathsf{Tr}^V_{A,B}(f \circ (\mathsf{id}_A \otimes g)) : A \to B.$$

(4) Action (vanishing): if $f : A \to B$*, then*

$$\mathsf{Tr}^I_{A,B}(\rho^{-1} \circ f \circ \rho) = f : A \to B,$$

and if $f : A \otimes (U \otimes V) \to B \otimes (U \otimes V)$*, then*

$$\mathsf{Tr}^{U\otimes V}_{A,B}(f) = \mathsf{Tr}^{U}_{A,B}\left(\mathsf{Tr}^{V}_{A\otimes U, B\otimes U}\left(\alpha^{-1}\circ f\circ\alpha\right)\right).$$

(5) Superposing: if $f : A \otimes U \to B \otimes U$*, then*

$$\mathsf{Tr}^{U}_{C\otimes A, C\otimes B}\left(\alpha^{-1}\circ(\mathsf{id}_C\otimes f)\circ\alpha\right) = \mathsf{id}_C \otimes \mathsf{Tr}^{U}_{A,B}(f) : C\otimes A \to C\otimes B.$$

(6) Yanking: for all U*,*

$$\mathsf{Tr}^{U}_{U,U}\left(\sigma_{U,U}\right) = \mathsf{id}_U : U \to U.$$

Ponto and Shulman [PS13] give an expository accounts of traces. Selinger [Sel11] gives an expository account of graphical languages used to reason and communicate about monoidal categories, including traced monoidal categories.

Relatedly, a category can have a parametrized fixed-point operator:

Definition 2 ([SP00, Definitions 2.2, 2.4]). *A* **parametrized fixed-point operator** *on a category* $\mathbf{M}$ *is a family of morphisms* $(\cdot)^\dagger : \mathbf{M}(X\times A \to A) \Rightarrow \mathbf{M}(X \to A)$ *satisfying:*

(1) Naturality: for any $g : X \to Y$ *and* $f : Y \times A \to A$*,*

$$f^\dagger \circ g = (f\circ(g\times \mathsf{id}_A))^\dagger : X \to A.$$

(2) The parametrized fixed-point property: for any $f : X \times A \to A$*,*

$$f\circ\langle \mathsf{id}_X, f^\dagger\rangle = f^\dagger : X \to A.$$

It is a **Conway operator** *if it additionally satisfies:*

(3) Parameterized dinaturality: for any $f : X \times B \to A$ *and* $g : X \times A \to B$*,*

$$f\circ\langle \mathsf{id}_X, (g\circ\langle\pi_X, f\rangle)^\dagger\rangle = (f\circ\langle\pi_1, g\rangle)^\dagger : X \to A.$$

(4) The diagonal property: for any $f : X \times A \times A \to A$*,*

$$(f\circ(\mathsf{id}_X\times\Delta))^\dagger = \left(f^\dagger\right)^\dagger : X \to A,$$

where $\Delta : A \to A \times A$ *is the diagonal map.*

In [Has99b, Proposition 7.1.4; Has03, Definition 2.2], Hasegawa generalizes Plotkin's uniformity principle [Plo78, Exercise 2.30] for fixed points to define **uniform traces**. In the case of the usual trace on a category of domains, uniformity means that for any strict morphism $h : X \to Y$ and morphisms $f : A\times X \to B\times X$ and $g : A\times Y \to B\times Y$, if $(\mathsf{id}_B\times h)\circ f = g\circ(\mathsf{id}_A\times h) : A\times B \to B\times Y$, then $\mathsf{Tr}^X_{A,B}(f) = \mathsf{Tr}^Y_{A,B}(g) : A \to B$. In terms of applications, that means that to show $\mathsf{Tr}^X_{A,B}(f) = \mathsf{Tr}^Y_{A,B}(g) : A \to B$, it is sufficient to find a strict $h : X \to Y$ satisfying the uniformity hypothesis.

6.3.2. Related Work. Traces have been widely used to model feedback and communication. For example, Abramsky [Abr96] uses traces to capture feedback for resumptions, and to capture symmetric feedback between interacting processes. Abramsky and Jagadeesan [AJ94] similarly use traces to capture feedback between processes when interpreting the Cut rule of linear logic in a Geometry of Interaction interpretation. Selinger [Sel99] uses traces to give a categorical account of asynchronous communication in networks of parallel processes. Abramsky, Haghverdi, and Scott [AHS02] give an axiomatic account of Girard's Geometry of Interaction in a categorical setting. They used traces to define composition of morphisms as symmetric feedback.

Traces and Conway operators are well-studied in general categorical settings. Hasegawa [Has99b, Theorem 7.1] and Hyland independently discovered [BH03, p. 281] that a cartesian category has a trace if and only if it has a Conway operator. Motivated by examples from functional programming, Benton and Hyland [BH03] generalized this result to Freyd categories. Hasegawa [Has03] uses uniform traces to construct new traced monoidal categories from existing ones. Simpson and Plotkin [SP00] give a complete axiomatic account of Conway operators in categories. As future work, they propose extending their work to give a complete equational theory for uniform traces.

6.3.3. Contribution. The trace operator on ω-**aBC** is uniform and is used to interpret process composition. As a result, it would be useful in reasoning about process composition to have a complete equational theory for uniform traces. I propose exploring this problem.

CHAPTER 7

Conclusion and Timeline

I have proposed an ambitious but achievable body of work to support my thesis statement:

> *Denotational semantics elucidate the structure of session-typed languages and allow us to reason about programs written in these languages in ways that are complementary to existing approaches.*

Indeed, in section 3.1 I presented a taste of a denotational semantics for SILL, a session-typed language with a functional layer and general recursion. I used this semantics to prove various η-like equivalences for processes. In the cited completed work [Kav20], I used this semantics to prove other interesting equivalences. For example, I showed that flipping a stream of bits twice is semantically equivalent to the forwarding (identity) process.

I proposed work in subsequent sections that builds on this semantics to further support my thesis statement. In section 5.1, I proposed showing that my semantics is sound and adequate relative to existing operational semantics. This establishes the complementary nature of my semantics.

In sections 5.2 and 5.3 I proposed to expand my semantics to account for features such as shared-memory and dependent session types. This will show that the semantic techniques I have develop scale to richer settings.

My proposed work supports the claim that denotational semantics can be used to reason about programs written in session-typed languages. I proposed using denotational semantics to reason about run-time monitoring of programs in section 5.4.1. In section 5.4.2, I proposed using my semantics to validate various non-trivial process equivalences and to verify the correctness of other processes.

I propose completing the proposed work according to the following tentative timeline.

February 2020: The work of section 3.1 has been submitted to FSCD 2020.

April 2020: A large portion of the work for section 4.1 is complete, but needs polishing. Several conjectures remain. I propose submitting this work to MFPS 2020. Its deadlines are March 30, 2020 (abstract submission) and April 3, 2020 (paper submission).

The work of section 5.1 is still in its preliminary stages and could be submitted to CONCUR 2020. Its deadlines are April 15, 2020 (abstract submission) and April 22, 2020 (paper submission).

July 2020: Submit the results of section 5.2 to POPL 2021. Its deadline is July 9, 2020.

October 2020: Submit the results of section 5.3 to FoSSaCS.

March/April 2021: The work of section 6.1 is partially complete, but is of low priority. No work has been done on section 6.2. Their results could be submitted to MFPS 2021 or CONCUR 2021.

Spring 2021: Write my dissertation.

May 2021: Defend.

Bibliography

[AB04] Steven Awodey and Andrej Bauer. "Propositions As [Types]". In: *Journal of Logic and Computation* 14.4 (Aug. 2004), pp. 447–471. ISSN: 1465-363X. DOI: 10.1093/logcom/14.4.447 (cit. on p. 27).

[Abr96] Samson Abramsky. "Retracing Some Paths in Process Algebra". In: *CONCUR '96: Concurrency Theory*. Concur '96 : 7th International Conference on Concurrency Theory (Pisa, Italy, Aug. 26–29, 1996). Ed. by Ugo Montanari and Vladimiro Sassone. Lecture Notes in Computer Science 1119. Springer-Verlag Berlin Heidelberg, 1996. ISBN: 978-3-540-70625-0. DOI: 10.1007/3-540-61604-7 (cit. on p. 37).

[AGM95] S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, eds. *Handbook of Logic in Computer Science*. Vol. 3: *Semantic Structures*. 5 vols. New York: Oxford University Press Inc., June 15, 1995. xv+490 pp. ISBN: 0-19-853762-X.

[AGN96] Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. "Interaction Categories and the Foundation of Typed Concurrent Programming". In: *Deductive Program Design*. NATO Advanced Study Institute on Deductive Program Design (Marktoberdorf, Germany, July 26–Aug. 7, 1994). Ed. by Manfred Broy. NATO ASI Series. Series F: Computer and Systems Sciences 152. NATO Scientific Affairs Division. Springer-Verlag Berlin Heidelberg, 1996, pp. 35–113. ISBN: 3-540-60947-4 (cit. on p. 9).

[AHS02] Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. "Geometry of Interaction and Linear Combinatory Algebras". In: *Mathematical Structures in Computer Science* 12.5 (Oct. 2002), pp. 625–665. ISSN: 1469-8072. DOI: 10.1017/s0960129502003730 (cit. on pp. 4, 9, 37).

[AJ94] S. Abramsky and R. Jagadeesan. "New Foundations for the Geometry of Interaction". In: *Information and Computation* 111.1 (May 15, 1994), pp. 53–119. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1041 (cit. on pp. 9, 37).

[AJ95] Samson Abramsky and Achim Jung. "Domain Theory". In: *Handbook of Logic in Computer Science*. Vol. 3: *Semantic Structures*. Ed. by S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum. 5 vols. New York: Oxford University Press Inc., June 15, 1995, pp. 1–168. ISBN: 0-19-853762-X (cit. on pp. 4, 10, 13, 14, 19).

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures. An Introduction to Category Theory for the Working Computer Scientist*. Foundations of Computing. Cambridge, Massachusetts: The MIT Press, 1991. xi+306 pp. ISBN: 0-262-01125-5 (cit. on p. 13).

[Atk17] Robert Atkey. "Observed Communication Semantics for Classical Processes". In: *Programming Languages and Systems*. 26th European Symposium on Programming, ESOP 2017 (Uppsala, Sweden, Apr. 22–29, 2017). Ed. by Hongseok Yang. Lecture Notes in Computer Science 10201. Berlin: Springer Berlin Heidelberg, 2017, pp. 56–82. ISBN: 978-3-662-54434-1. DOI: 10.1007/978-3-662-54434-1 (cit. on pp. 1, 10, 21, 23).

[BÉ95] Stephen L. Bloom and Zoltán Ésik. "Some Equational Laws of Initiality in 2CCC's". In: *International Journal of Foundations of Computer Science* 6.2 (1995), pp. 95–118. DOI: 10.1142/S0129054195000081 (cit. on pp. 15, 17).

[BÉ96] Stephen L. Bloom and Zoltán Ésik. "Fixed-Point Operations on ccc's. Part I". In: *Theoretical Computer Science* 155.1 (Feb. 25, 1996), pp. 1–38. ISSN: 0304-3975. DOI: 10.1016/0304-3975(95)00010-0 (cit. on pp. 4, 14, 16–18).

[Bek84] Hans Bekić. "Definable Operations in General Algebras, and the Theory of Automata and Flowcharts". In: Hans Bekič. *Programming Languages and Their Definition. Selected Papers*. Ed. by C. B. Jones. With an intro. by Cliff B. Jones. Lecture Notes in Computer Science 177. Springer-Verlag Berlin Heidelberg, 1984, pp. 30–55. ISBN: 978-3-540-38933-0. DOI: 10.1007/bfb0048939 (cit. on p. 14).

[Ben94] P. N. Benton. *A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models*. Preliminary Report. Tech. rep. UCAM-CL-TR-352. Cambridge, United Kingdom: Computer Laboratory, University of Cambridge, Oct. 1994. 65 pp. (cit. on p. 25).

[Ben95] P. N. Benton. "A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models". Extended Abstract. In: *Computer Science Logic*. 8th Workshop, CSL '94. Annual Conference of the European Association for Computer Science Logic, CSL '94 (Kazimierz, Poland, Sept. 25–30, 1994). Ed. by Leszek Pacholski and Jerzy Tiuryn. Lecture Notes in Computer Science 933. Springer-Verlag Berlin Heidelberg, 1995, pp. 121–135. ISBN: 978-3-540-49404-1. DOI: 10.1007/BFb0022251 (cit. on pp. 1, 25).

[BH03] Nick Benton and Martin Hyland. "Traced Premonoidal Categories". In: *RAIRO - Theoretical Informatics and Applications* 37.4 (Oct.–Dec. 2003), pp. 273–299. ISSN: 1290-385X. DOI: 10.1051/ita:2003020 (cit. on pp. 36, 37).

[BW96] Nick Benton and Philip Wadler. "Linear Logic, Monads and the Lambda Calculus". In: *Proceedings*. 11th Annual IEEE Symposium on Logic in Computer Science (New Brunswick, New Jersey, July 27–30, 1996). IEEE Computer Society Technical Committee on Mathematical Foundations of Computing. Los Alamitos, California: IEEE Computer Society Press, 1996, pp. 420–431. ISBN: 0-8186-7463-6. DOI: 10.1109/LICS.1996.561458 (cit. on p. 1).

[BW99] Michael Barr and Charles Wells. *Category Theory for Computing Science*. 3rd ed. Montreal, Quebec: Les Publications CRM, 1999. xvii+526 pp. ISBN: 2-921120-31-3 (cit. on p. 36).

[Che+17] Tzu-chun Chen et al. "On the Preciseness of Subtyping in Session Types". In: *Logical Methods in Computer Science* 13.2:12 (June 30, 2017), pp. 1–61. DOI: 10.23638/lmcs-13(2:12)2017 (cit. on p. 35).

[CP10] Luís Caires and Frank Pfenning. "Session Types as Intuitionistic Linear Propositions". In: *CONCUR 2010 — Concurrency Theory*. 21st International Conference, CONCUR 2010 (Paris, France, Aug. 31–Sept. 3, 2010). Ed. by Paul Gastin and François Laroussinie. Lecture Notes in Computer Science 6269. Springer-Verlag Berlin Heidelberg, 2010, pp. 222–236. ISBN: 978-3-642-15374-7. DOI: 10.1007/978-3-642-15375-4_16 (cit. on pp. 1, 3, 7).

[CPT12] Luís Caires, Frank Pfenning, and Bernardo Toninho. "Towards Concurrent Type Theory". In: *TLDI'12*. 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (Philadelphia, Pennsylvania, Jan. 28, 2012). New York, New York: Association for Computing Machinery, Inc., 2012, pp. 1–12. ISBN: 978-1-4503-1120-5. DOI: 10.1145/2103786.2103788 (cit. on pp. 26, 28).

[Cro93] Roy L. Crole. *Categories for Types*. Cambridge, United Kingdom: Cambridge University Press, 1993. xvii+335 pp. ISBN: 0-521-45701-7 (cit. on p. 7).

[CS09] Iliano Cervesato and Andre Scedrov. "Relating State-Based and Process-Based Concurrency Through Linear Logic (full-Version)". In: *Information and Computation* 207.10 (Oct. 2009): *13th Workshop on Logic, Language, Information and Computation (WoLLIC 2006)*, pp. 1044–1077. ISSN: 0890-5401. DOI: 10.1016/j.ic.2008.11.006 (cit. on p. 21).

[CŞ90] Virgil Emil Căzănescu and Gheorghe Ştefănescu. "Towards a New Algebraic Foundation of Flowchart Scheme Theory". In: *Fundamenta Informaticae* 13.2 (June 1990),

pp. 171–210 (cit. on pp. 4, 9, 36). Repr. of Virgil-Emil Căzănescu and Gheorghe Ştefănescu. "Towards a New Algebraic Foundation of Flowchart Scheme Theory". In: INCREST Preprint Series in Mathematics 43 (Dec. 1987). ISSN: 0250-3638.

[CY19] Simon Castellan and Nobuko Yoshida. "Two Sides of the Same Coin: Session Types and Game Semantics. A Synchronous Side and an Asynchronous Side". In: *Proceedings of the ACM on Programming Languages*. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (Cascais, Portugal, Jan. 13–19, 2019). Vol. 3. POPL. New York, New York: Association for Computing Machinery, Jan. 2, 2019, 27:1–27:29. DOI: 10.1145/3290340 (cit. on p. 1).

[Dez+16] Mariangiola Dezani-Ciancaglini et al. "Denotational and Operational Preciseness of Subtyping: A Roadmap. Dedicated to Frank de Boer on the Occasion of His 60th Birthday". In: *Theory and Practice of Formal Methods. Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. Ed. by Erika Ábrahám, Marcello Bonsangue, and Einar Broch Johnsen. Lecture Notes in Computer Science 9960. Switzerland: Springer International Publishing, 2016, pp. 155–172. ISBN: 978-3-319-30734-3. DOI: 10.1007/978-3-319-30734-3 (cit. on p. 35).

[DHP18] Ankush Das, Jan Hoffmann, and Frank Pfenning. "Parallel Complexity Analysis With Temporal Session Types". In: *Proceedings of the ACM on Programming Languages* 2.ICFP, 91 (Sept. 2018). ISSN: 2475-1421. DOI: 10.1145/3236786 (cit. on p. 29).

[DP19] Farzaneh Derakhshan and Frank Pfenning. *Circular Proofs as Session-Typed Processes: A Local Validity Condition*. Aug. 6, 2019. arXiv: 1908.01909v1 [cs.LO] (cit. on p. 1).

[DP20] Ankush Das and Frank Pfenning. *Session Types with Arithmetic Refinements and Their Application to Work Analysis*. Jan. 23, 2020. arXiv: 2001.04439v3 [cs.PL] (cit. on p. 35).

[Eti+15] Pavel Etingof et al. *Tensor Categories*. Mathematical Surveys and Monographs 2015. Providence, Rhode Island: American Mathematical Society, 2015. xvi+343 pp. ISBN: 978-1-4704-2024-6 (cit. on p. 36).

[Fio94] Marcelo P. Fiore. "Axiomatic Domain Theory in Categories of Partial Maps". PhD thesis. The University of Edinburgh Department of Computer Science, Oct. 1994. v+282 pp. (cit. on p. 16).

[GG13] Dennis Griffith and Elsa L. Gunter. "LiquidPi: Inferrable Dependent Session Types". In: *NASA Formal Methods*. 5th International Symposium, NFM 2013 (Moffett Field, California, May 14–16, 2013). Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Lecture Notes in Computer Science 7871. Springer-Verlag Berlin Heidelberg, 2013, pp. 185–197. ISBN: 978-3-642-38088-4. DOI: 10.1007/978-3-642-38088-4_13 (cit. on p. 28).

[GH05] Simon Gay and Malcolm Hole. "Subtyping for Session Types in the Pi Calculus". In: *Acta Informatica* 42.2-3 (Nov. 11, 2005), pp. 191–225. ISSN: 1432-0525. DOI: 10.1007/s00236-005-0177-z (cit. on p. 35).

[Gie+03] G. Gierz et al. *Continuous Lattices and Domains*. Encyclopedia of Mathematics and its Applications 93. Cambridge: Cambridge University Press, 2003. xxxvi+591 pp. ISBN: 0-521-80338-1. DOI: 10.1017/CBO9780511542725 (cit. on p. 13).

[GJP18] Hannah Gommerstadt, Limin Jia, and Frank Pfenning. "Session-Typed Concurrent Contracts". In: *Programming Languages and Systems*. 27th European Symposium on Programming, ESOP 2018 (Thessaloniki, Greece, Apr. 14–20, 2018). Ed. by Amal Ahmed. Lecture Notes in Computer Science 10801. Cham: Springer, 2018, pp. 771–798. ISBN: 978-3-319-89884-1. DOI: 10.1007/978-3-319-89884-1 (cit. on pp. 1, 28).

[GM89] Carl A. Gunter and Dana S. Mosses Peter D.and Scott. *Semantic Domains and Denotational Semantics*. Tech. rep. MS-CIS-89-16. Philadelphia, Pennsylvania: University of Pennsylvania Department of Computer and Information Science, Feb. 1989 (cit. on p. 13).

[Gom19] Hannah Gommerstadt. "Session-Typed Concurrent Contracts". PhD thesis. Pittsburgh, Pennsylvania: School of Computer Science, Carnegie Mellon University, Sept. 2019. xii+125 pp. (cit. on pp. 27, 28).

[Gun92] Carl A. Gunter. *Semantics of Programming Languages. Structures and Techniques*. Cambridge, Massachusetts: The MIT Press, 1992. 419 pp. ISBN: 0-262-07143-6 (cit. on pp. 4, 7, 13).

[GV10] Simon J. Gay and Vasco T. Vasconcelos. "Linear Type Theory for Asynchronous Session Types". In: *Journal of Functional Programming* 20.1 (Jan. 2010), pp. 19–50. ISSN: 1469-7653. DOI: 10.1017/s0956796809990268 (cit. on p. 35).

[Has03] Masahito Hasegawa. "The Uniformity Principle on Traced Monoidal Categories". In: *Electronic Notes in Theoretical Computer Science* 69 (Feb. 2003): *CTCS'02, Category Theory and Computer Science*, pp. 137–155. ISSN: 1571-0661. DOI: 10.1016/s1571-0661(04)80563-2 (cit. on pp. 36, 37).

[Has97] Masahito Hasegawa. "Models of Sharing Graphs". PhD thesis. University of Edinburgh, June 12, 1997. 142 pp. (cit. on p. 36).

[Has99a] Masahito Hasegawa. *Models of Sharing Graphs. A Categorical Semantics of `let` and `letrec`*. Distinguished Dissertations. Springer-Verlag London Limited, June 1999. xii+134 pp. ISBN: 978-1-4471-0865-8. DOI: 10.1007/978-1-4471-0865-8 (cit. on p. 36). Repr. of "Models of Sharing Graphs". PhD thesis. University of Edinburgh, June 12, 1997. 142 pp.

[Has99b] Masahito Hasegawa. "Recursion from Cyclic Sharing". In: *Models of Sharing Graphs. A Categorical Semantics of `let` and `letrec`*. Distinguished Dissertations. Springer-Verlag London Limited, June 1999. Chap. 7, pp. 83–101. ISBN: 978-1-4471-0865-8. DOI: 10.1007/978-1-4471-0865-8_7 (cit. on p. 37).

[Hon93] Kohei Honda. "Types for Dyadic Interaction". In: *CONCUR'93*. 4th International Conference on Concurrency Theory (Hildesheim, Germany, Aug. 23–26, 1993). Ed. by Eike Best. Lecture Notes in Computer Science 715. Berlin: Springer-Verlag Berlin Heidelberg, 1993, pp. 509–523. ISBN: 978-3-540-47968-0. DOI: 10.1007/3-540-57208-2_35 (cit. on p. 3).

[Jac99] Bart Jacobs. *Categorical Logic and Type Theory*. Ed. by S. Abramsky et al. Studies in Logic and the Foundations of Mathematics 141. Amsterdam, The Netherlands: Elsevier Science B.V., 1999. xvii+760 pp. ISBN: 0-444-50853-8 (cit. on p. 28).

[JSV96] André Joyal, Ross Street, and Dominic Verity. "Traced Monoidal Categories". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 119.3 (Apr. 1996), pp. 447–468. ISSN: 1469-8064. DOI: 10.1017/s0305004100074338 (cit. on pp. 4, 9, 36).

[Kah74] Gilles Kahn. "The Semantics of a Simple Language for Parallel Programming". In: *Information Processing 74*. 6th IFIP Congress 1974 (Stockholm, Sweden, Aug. 5–10, 1974). Ed. by Jack L. Rosenfeld. International Federation for Information Processing. North-Holland Publishing Company, 1974, pp. 471–475. ISBN: 0-7204-2803-3 (cit. on pp. 7, 9).

[Kav20] Ryan Kavanagh. *A Domain Semantics for Higher-Order Recursive Processes*. Feb. 14, 2020. arXiv: 2002.01960v2 [cs.PL] (cit. on pp. 1, 7, 10, 39).

[LM16] Sam Lindley and J. Garrett Morris. "Talking Bananas: Structural Recursion for Session Types". In: *ICFP'16*. 21st ACM SIGPLAN International Conference on Functional Programming (Nara, Japan, Sept. 18–24, 2016). Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM SIGPLAN. New York, New York: The Association for Computing Machinery, Inc., 2016, pp. 434–447. ISBN: 978-1-4503-4219-3 (cit. on p. 1).

[LS16] Daniel R. Licata and Michael Shulman. "Adjoint Logic With a 2-category of Modes". In: *Logical Foundations of Computer Science*. International Symposium, LFCS 2016 (Deerfield Beach, Florida, Jan. 4–7, 2016). Ed. by Sergei Artemov and Anil Nerode. Lecture Notes in Computer Science 9537. Cham, Switzerland: Springer International

Publishing AG, 2016, pp. 219–235. ISBN: 978-3-319-27683-0. DOI: 10.1007/978-3-319-27683-0_16 (cit. on p. 26).

[LSR17a] Daniel R. Licata, Michael Shulman, and Mitchell Riley. "A Fibrational Framework for Substructural and Modal Logics". In: *2nd International Conference on Formal Structures for Computation and Deduction*. FSCD 2017 (Oxford, United Kingdom, Sept. 3–9, 2017). Ed. by Dale Miller. Leibniz International Proceedings in Informatics 84. Saarbrücken/Wadern, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik GmbH, Sept. 2017, 25:1–25:22. ISBN: 978-3-95977-047-7. DOI: 10.4230/LIPICS.FSCD.2017.25 (cit. on p. 26).

[LSR17b] Daniel R. Licata, Michael Shulman, and Mitchell Riley. "A Fibrational Framework for Substructural and Modal Logics". Extended version. June 1, 2017. URL: http://dlicata.web.wesleyan.edu/pubs/lsr17multi/lsr17multi-ex.pdf (visited on 11/25/2019) (cit. on p. 26).

[Mac69a] S. Mac Lane, ed. *Reports of the Midwest Category Seminar III*. Lecture Notes in Mathematics 106. Springer-Verlag Berlin Heidelberg, 1969. 247 pp. ISBN: 978-3-540-36150-3. DOI: 10.1007/BFb0059139.

[Mac69b] Saunders Mac Lane. "One Universe As a Foundation for Category Theory". In: *Reports of the Midwest Category Seminar III*. Ed. by S. Mac Lane. Lecture Notes in Mathematics 106. Springer-Verlag Berlin Heidelberg, 1969, pp. 192–200. ISBN: 978-3-540-36150-3. DOI: 10.1007/bfb0059147 (cit. on p. 18).

[Mac98] Saunders Mac Lane. *Categories for the Working Mathematician*. 2nd ed. Graduate Texts in Mathematics 5. New York, Berlin, and Heidelberg: Springer-Verlag New York, Inc., 1998. xii+314 pp. ISBN: 0-387-98403-8 (cit. on p. 15).

[MBV19] Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. "Value-Dependent Session Design in a Dependently Typed Language". In: *Proceedings: Programming Language Approaches to Concurrency- and Communication-cEntric Software*. Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES) (Prague, Czech Republic, Apr. 7, 2019). Ed. by Francisco Martins and Dominic Orchard. Electronic Proceedings in Theoretical Computer Science 291. European Joint Conferences on Theory and Practice of Software. Apr. 2, 2019, pp. 47–59. DOI: 10.4204/EPTCS.291.5. arXiv: 1904.01288v1 [cs.PL] (cit. on p. 28).

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag Berlin Heidelberg, 1980. 171 pp. ISBN: 978-3-540-38311-6. DOI: 10.1007/3-540-10235-3 (cit. on p. 9).

[MO19] Francisco Martins and Dominic Orchard, eds. *Proceedings: Programming Language Approaches to Concurrency- and Communication-cEntric Software*. Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES) (Prague, Czech Republic, Apr. 7, 2019). Electronic Proceedings in Theoretical Computer Science 291. European Joint Conferences on Theory and Practice of Software. Mar. 31, 2019. DOI: 10.4204/EPTCS.291. arXiv: 1904.00396v1 [cs.PL].

[Mor68] James H. Morris. "Lambda-Calculus Models of Programming Languages". PhD thesis. Cambridge, Massachusetts: Massachusetts Institute of Technology, Dec. 1968 (cit. on p. 21).

[PCT11] Frank Pfenning, Luis Caires, and Bernardo Toninho. "Proof-Carrying Code in a Session-Typed Process Calculus". In: *Certified Programs and Proofs*. First International Conference, CPP 2011 (Kenting, Taywan, Dec. 7–9, 2011). Ed. by Jean-Pierre Jouannaud and Zhong Shao. Lecture Notes in Computer Science 7086. Springer-Verlag Berlin Heidelberg, 2011, pp. 21–36. ISBN: 978-3-642-25379-9. DOI: 10.1007/978-3-642-25379-9_4 (cit. on pp. 27, 28).

[Pér+12] Jorge A. Pérez et al. "Linear Logical Relations for Session-Based Concurrency". In: *Programming Languages and Systems*. 21st European Symposium on Programming, ESOP 2012 (Tallinn, Estonia, Mar. 24, 2012–Apr. 1, 2014). Ed. by Helmut Seidl. Lecture

Notes in Computer Science 7211. Heidelberg: Springer-Verlag Berlin Heidelberg, 2012, pp. 539–558. ISBN: 978-3-642-28869-2. DOI: 10.1007/978-3-642-28869-2_27 (cit. on p. 1).

[Pér+14] Jorge A. Pérez et al. "Linear Logical Relations and Observational Equivalences for Session-Based Concurrency". In: *Information and Computation* 239 (Dec. 2014), pp. 254–302. ISSN: 0890-5401. DOI: 10.1016/j.ic.2014.08.001 (cit. on pp. 1, 21).

[Pfe00] Frank Pfenning. "Structural Cut Elimination: I. Intuitionistic and Classical Logic". In: *Information and Computation* 157.1-2 (Feb. 25, 2000), pp. 84–141. ISSN: 0890-5401. DOI: 10.1006/inco.1999.2832 (cit. on p. 3).

[Pfe01] Frank Pfenning. "Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory". In: *16th Annual IEEE Symposium on Logic in Computer Science*. 16th Annual IEEE Symposium on Logic in Computer Science (Boston, Massachusetts, June 16–19, 2001). IEEE Computer Society Technical Committee on Mathematical Foundations of Computing. Los Alamitos, California: The Institute of Electrical and Electronics Engineers, Inc., 2001, pp. 221–230. ISBN: 0-7695-1281-X. DOI: 10.1109/LICS.2001.932499 (cit. on pp. 27, 28).

[PG15] Frank Pfenning and Dennis Griffith. "Polarized Substructural Session Types". In: *Foundations of Software Science and Computation Structures*. 18th International Conference, FOSSACS 2015 (London, United Kingdom, Apr. 11–18, 2015). Ed. by Andrew Pitts. Lecture Notes in Computer Science 9034. Berlin Heidelberg: Springer-Verlag GmbH Berlin Heidelberg, 2015, pp. 3–32. ISBN: 978-3-662-46678-0. DOI: 10.1007/978-3-662-46678-0_1 (cit. on pp. 7, 26).

[Pie02] Benjamin Pierce. *Types and Programming Languages*. Cambridge, Massachusetts: The MIT Press, 2002. xxi+623 pp. ISBN: 0-262-16209-1 (cit. on p. 33).

[PK18] Frank Pfenning and Ryan Kavanagh. "15-814 Types and Programming Languages". Lecture notes. Carnegie Mellon University, Pittsburgh, Pennsylvania, Sept. 4–Nov. 27, 2018. URL: https://www.cs.cmu.edu/~fp/courses/15814-f18/lectures/Notes-15814-f18.pdf (cit. on p. 29).

[Plo78] Gordon Plotkin. "The Category of Complete Partial Orders: a Tool for Making Meanings". In: *Summer School on Foundations of Artificial Intelligence and Computer Science*. Summer School on Foundations of Artificial Intelligence and Computer Science (Pisa, Italy, June 19–30, 1978). 1978 (cit. on p. 37).

[Pol81] Wolfgang Polak. "Program Verification Based on Denotation Semantics". In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'81 (Williamsburg, Virginia, Jan. 26–28, 1981). ACM SIGPLAN and ACM SIGACT. New York, New York: Association for Computing Machinery, Inc., 1981, pp. 149–158. ISBN: 0-89791-029-X. DOI: 10.1145/567532.567549 (cit. on p. 1).

[PP19a] Klaas Pruiksma and Frank Pfenning. "A Message-Passing Interpretation of Adjoint Logic". In: *Proceedings: Programming Language Approaches to Concurrency- and Communication-cEntric Software*. Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES) (Prague, Czech Republic, Apr. 7, 2019). Ed. by Francisco Martins and Dominic Orchard. Electronic Proceedings in Theoretical Computer Science 291. European Joint Conferences on Theory and Practice of Software. Apr. 2, 2019, pp. 60–79. DOI: 10.4204/EPTCS.291.6. arXiv: 1904.01290v1 [cs.PL] (cit. on pp. 25, 26).

[PP19b] Klaas Pruiksma and Frank Pfenning. "Back to Futures". Oct. 25, 2019. URL: https://www.cs.cmu.edu/~fp/papers/futures19.pdf (visited on 11/06/2019) (cit. on pp. 1, 25, 26).

[Pru+18] Klaas Pruiksma et al. "Adjoint Logic". Apr. 24, 2018. URL: https://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf (visited on 11/11/2019) (cit. on pp. 1, 25, 26).

[PS13] Kate Ponto and Michael Shulman. *Traces in Symmetric Monoidal Categories*. Oct. 24, 2013. arXiv: 1107.6032v2 [math.CT] (cit. on pp. 36, 37).

[Ree09] Jason Reed. "A Judgmental Deconstruction of Modal Logic". May 23, 2009. URL: https://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf (visited on 11/11/2019) (cit. on pp. 1, 25, 26).

[Rey09] John C. Reynolds. *Theories of Programming Languages*. New York, New York: Cambridge University Press, Apr. 2009. xii+500 pp. ISBN: 978-0-521-10697-9 (cit. on p. 7).

[RFC793] Information Sciences Institute, University of Southern California. *Transmission Control Protocol*. DARPA Internet Program Protocol Specification. RFC 793. Internet Engineering Task Force, Sept. 1981. DOI: 10.17487/RFC0793 (cit. on p. 26).

[Rie16] Emily Riehl. *Category Theory in Context*. Mineola, New York: Dover Publications, Inc, 2016. ISBN: 978-0-486-80903-8 (cit. on p. 5).

[Sch72] Horst Schubert. *Categories*. Trans. from the German by Eva Gray. Springer-Verlag Berlin Heidelberg, 1972. xi+385 pp. ISBN: 978-3-642-65364-3. DOI: 10.1007/978-3-642-65364-3 (cit. on p. 18).

[SD78] Carl A. Sunshine and Yogen K. Datal. "Connection Management in Transport Protocols". In: *Computer Networks* 2.6 (Dec. 1978), pp. 454–473. ISSN: 0376-5075. DOI: 10.1016/0376-5075(78)90053-3 (cit. on p. 26).

[Sel11] P. Selinger. "A Survey of Graphical Languages for Monoidal Categories". In: *New Structures for Physics*. Ed. by Bob Coecke. Lecture Notes in Physics 813. Springer-Verlag Berlin Heidelberg, 2011. Chap. 4, pp. 289–355. ISBN: 978-3-642-12821-9. DOI: 10.1007/978-3-642-12821-9_4 (cit. on p. 37).

[Sel99] Peter Selinger. "Categorical Structure of Asynchrony". In: *Electronic Notes in Theoretical Computer Science* 20 (1999): *MFPS XV, Mathematical Foundations of Progamming Semantics, Fifteenth Conference*, pp. 158–181. ISSN: 1571-0661. DOI: 10.1016/s1571-0661(04)80073-2 (cit. on pp. 36, 37).

[Sim12] Robert J. Simmons. "Substructural Logical Specifications". PhD thesis. Pittsburgh, Pennsylvania: Computer Science Department, Carnegie Mellon University, Nov. 14, 2012. xvi+300 pp. (cit. on p. 21).

[SP00] Alex Simpson and Gordon Plotkin. "Complete Axioms for Categorical Fixed-Point Operators". In: *15th Annual IEEE Symposium on Logic in Computer Science*. 15th Annual IEEE Symposium on Logic in Computer Science (Santa Barbara, California, June 26–28, 2000). IEEE Computer Society Technical Committee on Mathematical Foundations of Computing. Los Alamitos, California: IEEE Computer Society, 2000, pp. 30–41. ISBN: 0-7695-0725-5. DOI: 10.1109/LICS.2000.855753 (cit. on p. 37).

[SP82] M. B. Smyth and G. D. Plotkin. "The Category-Theoretic Solution of Recursive Domain Equations". In: *SIAM Journal on Computing* 11.4 (1982), pp. 761–783. DOI: 10.1137/0211062 (cit. on pp. 4, 13–15).

[TCP11] Bernardo Toninho, Luís Caires, and Frank Pfenning. "Dependent Session Types Via Intuitionistic Linear Type Theory". In: *PPDP'11*. 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming (Odense, Denmark, July 20–22, 2011). New York, New York: Association for Computing Machinery, Inc., 2011, pp. 161–172. ISBN: 978-1-4503-0776-5. DOI: 10.1145/2003476.2003499 (cit. on pp. 1, 26–28).

[TCP13] Bernardo Toninho, Luis Caires, and Frank Pfenning. "Higher-Order Processes, Functions, and Sessions: A Monadic Integration". In: *Programming Languages and Systems*. 22nd European Symposium on Programming, ESOP 2013 (Rome, Italy, Mar. 16–24, 2013). Ed. by Matthias Felleisen and Philippa Gardner. Lecture Notes in Computer Science 7792. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 350–369. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_20 (cit. on pp. 1, 3, 7).

[Ten91] Robert D. Tennent. *Semantics of Programming Languages*. Englewood Cliffs, New Jersey: Prentice Hall Inc, 1991. 236 pp. ISBN: 978-0-13-805599-8 (cit. on p. 13).

[Ten95] R. D. Tennent. “Denotational Semantics”. In: *Handbook of Logic in Computer Science*. Vol. 3: *Semantic Structures*. Ed. by S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum. 5 vols. New York: Oxford University Press Inc., June 15, 1995, pp. 169–322. ISBN: 0-19-853762-X (cit. on p. 7).

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. “An Interaction-Based Language and Its Typing System”. In: *PARLE'94. Parallel Architectures and Languages Europe*. 6th International PARLE Conference (Athens, Greece, July 4–8, 1994). Ed. by Costas Halatsis et al. Lecture Notes in Computer Science 10201. Berlin: Springer-Verlag Berlin Heidelberg, 1994, pp. 398–413. ISBN: 978-3-540-48477-6. DOI: 10.1007/3-540-58184-7_118 (cit. on p. 3).

[Tom75] Raymond S. Tomlinson. “Selecting Sequence Numbers”. In: *ACM SIGOPS Operating Systems Review* 9.3 (July 1975), pp. 11–23. ISSN: 0163-5980. DOI: 10.1145/563905.810894 (cit. on p. 26).

[Ton15] Bernardo Parente Coutinho Fernandes Toninho. “A Logical Foundation for Session-based Concurrent Computation”. English and Portuguese. PhD thesis. Universidade Nova de Lisboa, May 2015. xviii+178 pp. (cit. on pp. 1, 21).

[TY18a] Bernardo Toninho and Nobuko Yoshida. “Depending on Session-Typed Processes”. In: *Foundations of Software Science and Computation Structures*. 21st International Conference, FOSSACS 2018 (Thessaloniki, Greece, Apr. 14–20, 2018). Ed. by Christel Baier and Ugo Dal Lago. Lecture Notes in Computer Science 10803. European Joint Conferences on Theory and Practice of Software. Cham, Switzerland: SpringerOpen, 2018, pp. 128–145. ISBN: 978-3-319-89366-2. DOI: 10.1007/978-3-319-89366-2_7 (cit. on pp. 1, 27, 28, 48).

[TY18b] Bernardo Toninho and Nobuko Yoshida. *Depending on Session-Typed Processes*. Extended version of [TY18a]. Jan. 24, 2018. arXiv: 1801.08114v1 [cs.PL] (cit. on pp. 1, 27, 28).

[Wad14] Philip Wadler. “Propositions As Sessions”. In: *Journal of Functional Programming* 24.2-3 (Jan. 31, 2014), pp. 384–418. ISSN: 1469-7653. DOI: 10.1017/s095679681400001x (cit. on pp. 1, 3, 21).

[Wan95] Mitchell Wand. “Compiler Correctness for Parallel Languages”. In: *FPCA '95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, June 26–28, 1995). ACM SIGARCH and ACM SIGPLAN. New York, New York: Association for Computing Machinery, Inc., 1995, pp. 120–134. ISBN: 0-89791-719-7. DOI: 10.1145/224164.224193 (cit. on p. 1).