Communication-Based Semantics for Recursive Session-Typed Processes

Ryan Kavanagh  September 28, 2021

Committee: Stephen Brookes, co-chair
Frank Pfenning, co-chair
Jan Hoffmann
Luís Caires, Universidade Nova de Lisboa
Gordon Plotkin, University of Edinburgh
Programs written in session-typed programming languages are guaranteed to obey their protocols.
Programs written in session-typed programming languages are guaranteed to obey their protocols.
Programs written in session-typed programming languages are guaranteed to obey their protocols.
Programs written in session-typed programming languages are guaranteed to obey their protocols.
Programs written in session-typed programming languages are guaranteed to obey their protocols.
"Program equivalence is arguably one of the most interesting and at the same time important problems in formal verification.”\textsuperscript{1}

There are existing notions of program equivalence for session-typed languages:
There are existing notions of program equivalence for session-typed languages:

- **Wadler’s Classical Processes (CP):** Atkey [2017] gives a relational semantics.
- **Hypersequent CP:** Kokke et al. [2019] give a denotational semantics using Brzozowski derivatives.
Some Existing Approaches to Equivalence

There are existing notions of program equivalence for session-typed languages:

- Synchronous session-typed $\pi$-calculus: Castellan and Yoshida [2019] give a game semantics.
Some Existing Approaches to Equivalence

There are existing notions of program equivalence for session-typed languages:

- Synchronous session-typed $\pi$-calculus: Castellan and Yoshida [2019] give a game semantics.
Some Existing Approaches to Equivalence

There are existing notions of program equivalence for session-typed languages:

- Synchronous session-typed $\pi$-calculus: Castellan and Yoshida [2019] give a game semantics.

**Problem:** It is not clear how to extend these approaches to handle full-featured languages.
When one attempts to combine language concepts, unexpected and counterintuitive interactions arise. At this point, even the most experienced designer’s intuition must be buttressed by a rigorous definition of what the language means. — John Reynolds, 1990
We want to reason about programs in a session-typed language with:

- general recursion at the program and type level
- functional programming features
- higher-order features: send/receive channels and programs
A **process** is a computational agent that interacts with its environment solely through communication.

**Communication** is a sequence of atomic observable events caused by a process.
Communication is the only observable phenomenon of processes!
Communication-based semantics elucidate the structure of session-typed languages and allow us to reason about programs written in these languages.
We will study “Polarized SILL”, a language with:

1. a functional programming layer
2. session-typed message passing concurrency
3. general recursion (types and programs)
4. higher-order features: processes can send/receive channels and programs
Contributions

We give Polarized SILL

1. **An observed communication semantics**
2. **A communication-based testing equivalences framework**
3. **A communication-based denotational semantics**

and we use these semantics to reason about processes.
Where

- $c_i$ — channel name
- $A_i$ — protocol (session type) for channel $c_i$
- $P$ — process
Polarized SILL

\[
\begin{array}{c}
\begin{array}{c}
\vdots \\
\end{array} \\
\end{array}
\quad P \\
\begin{array}{c}
\begin{array}{c}
\vdots \\
\end{array} \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
c_1 : A_1 \\
\vdots \\
c_n : A_n \\
\end{array}
\quad P 
\]

\[
\begin{array}{c}
\begin{array}{c}
\vdots \\
\end{array} \\
\end{array}
\quad c_0 : A_0 \\
\begin{array}{c}
\begin{array}{c}
\vdots \\
\end{array} \\
\end{array}
\end{array}
\]

\textbf{Uses} \quad \textbf{Provides}

Where

- \( c_i \) — channel name
- \( A_i \) — protocol (session type) for channel \( c_i \)
- \( P \) — process
Abbreviate as:

\[ c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 \quad (n \geq 0) \]

\[ \Delta \vdash P :: c_0 : A_0 \]

where \( \Delta = c_1 : A_1, \ldots, c_n : A_n \).
Bit Streams in SILL

Bit stream protocol:

\[ \text{bits} = (b0: \text{bits}) \oplus (b1: \text{bits}) \]

Example communications satisfying \text{bits}:

\[ b0 \ b1 \ b0 \ b0 \ldots \rightarrow c_0 : \text{bits} \]
\[ b1 \rightarrow c_1 : \text{bits} \]
\[ \rightarrow c_2 : \text{bits} \]
Bit Stream protocol:

\[ \text{bits} = (b_0 : \text{bits}) \oplus (b_1 : \text{bits}) \]

Example communications satisfying \( \text{bits} \):

- \( b_0 b_1 b_0 b_0 \ldots \rightarrow c_0 : \text{bits} \)
- \( b_1 \perp \rightarrow c_1 : \text{bits} \)
- \( \perp \rightarrow c_2 : \text{bits} \)
Flipping Bits

\[ i : \text{bits} \overset{b0\ b1\ b0}{\rightarrow} \overset{b1\ b0\ b1}{\rightarrow} o : \text{bits} \]

\[ i : \text{bits} \vdash F :: o : \text{bits} \]

\[ o \leftarrow F \leftarrow i = \text{case } i \{ \begin{array}{l}
  b0 \Rightarrow o.b1; \quad o \leftarrow F \leftarrow i \\
  b1 \Rightarrow o.b0; \quad o \leftarrow F \leftarrow i
\end{array} \]
Flipping Bits

\[
i : \text{bits} \quad \xrightarrow{b0 b1 b0} \quad F \quad \xrightarrow{b1 b0 b1} \quad o : \text{bits}
\]

\[
i : \text{bits} \quad |- \quad F : : \quad o : \text{bits}
\]

\[
o \leftarrow F \leftarrow i = \text{case } i \{ \begin{align*}
    & b0 \Rightarrow o.b1; o \leftarrow F \leftarrow i \\
    | & b1 \Rightarrow o.b0; o \leftarrow F \leftarrow i
\end{align*} \}
\]
Flipping Bits

\[
\begin{align*}
  i : \text{bits} & \quad \xrightarrow{b_0 b_1 b_0} \quad F \quad \xrightarrow{b_1 b_0 b_1} \quad o : \text{bits} \\
  i : \text{bits} & \mid - F :: o : \text{bits} \\
  o & \leftarrow F \leftarrow i = \text{case } i \{ \ b_0 \Rightarrow o.b_1; \ o & \leftarrow F \leftarrow i \\
  & \mid b_1 \Rightarrow o.b_0; \ o & \leftarrow F \leftarrow i \} 
\end{align*}
\]
Flipping Bits

\[
i : \text{bits} \dashv F :: o : \text{bits}
\]

\[
o \leftarrow F \leftarrow i = \text{case } i \{ \begin{array}{l}
b0 \Rightarrow o.b1; o \leftarrow F \leftarrow i \\
| \\
b1 \Rightarrow o.b0; o \leftarrow F \leftarrow i \end{array}\}
\]
Flipping Bits

\[
i : \text{bits} \xrightarrow{\begin{array}{c} b_0 b_1 b_0 \\ \end{array}} F \xrightarrow{\begin{array}{c} b_1 b_0 b_1 \\ \end{array}} o : \text{bits}
\]

\[
i : \text{bits} \vdash F :: o : \text{bits}
\]

\[
o \leftarrow F \leftarrow i = \text{case } i \{ \begin{array}{l}
  b_0 \Rightarrow o.b_1; o \leftarrow F \leftarrow i \\
  b_1 \Rightarrow o.b_0; o \leftarrow F \leftarrow i
\end{array}
\}
\]
i : bits \vdash F :: o : bits

\[ o \leftarrow F \leftarrow i = \text{case } i \{ \ b0 \Rightarrow o.b1; \ o \leftarrow F \leftarrow i \ \\
| \ b1 \Rightarrow o.b0; \ o \leftarrow F \leftarrow i \ \} \]
Flipping Bits

\[ i : \text{bits} \rightarrow b0 \ b1 \ b0 \rightarrow F \rightarrow b1 \ b0 \ b1 \rightarrow o : \text{bits} \]

\[
i : \text{bits} |- F :: o : \text{bits}\\
o \leftarrow F \leftarrow i = \text{case } i \{ \begin{array}{l}
b0 \Rightarrow o.b1; \ o \leftarrow F \leftarrow i \\
b1 \Rightarrow o.b0; \ o \leftarrow F \leftarrow i \\
\end{array} \}
\]
Flipping Bits

\[ i : \text{bits} \rightarrow F \rightarrow o : \text{bits} \]

\[ i : \text{bits} \mid - F :: o : \text{bits} \]

\[ o <- F <- i = \text{case } i \ { \begin{array}{l} \text{b0 } \Rightarrow o.b1; \ o <- F <- i \\ \text{b1 } \Rightarrow o.b0; \ o <- F <- i \end{array} } \]
i : bits |− F : : o : bits

o <- F <- i = case i { b0 ⇒ o.b1; o <- F <- i
| b1 ⇒ o.b0; o <- F <- i }
Flipping Bits

\[ i : \text{bits} \quad \xrightarrow{\text{b0 b1 b0}} \quad F \quad \xrightarrow{\text{b1 b0 b1}} \quad o : \text{bits} \]

\[
\begin{align*}
i : \text{bits} \quad | \quad F : : o : \text{bits} \\
o \leftarrow F \leftarrow i &= \text{case } i \{ \ b0 \Rightarrow o.\text{b1}; \ o \leftarrow F \leftarrow i \\
&\quad | \ b1 \Rightarrow o.\text{b0}; \ o \leftarrow F \leftarrow i \} 
\end{align*}
\]
Flipping Bits

\[
i : \text{bits} \;\vdash \; F :: o : \text{bits}
\]

\[
o \leftarrow F \leftarrow i = \text{case } i \{ \begin{array}{l}
b0 \Rightarrow o.b1; o \leftarrow F \leftarrow i \\
b1 \Rightarrow o.b0; o \leftarrow F \leftarrow i \end{array}
\}
\]
Observed Communication
Semantics
Idea: The meaning of a process is the communications we observe during its execution.
Idea: The meaning of a process is the communications we observe during its execution.

Questions:

1. What are observed communications?
Idea: The meaning of a process is the communications we observe during its execution.

Questions:

1. What are observed communications?
2. How do we observe them?
A session type specifies permitted communications.
A session type specifies permitted communications.

Write $w \in A$ to mean $w$ is a communication satisfying the session type $A$. 
A session type specifies permitted communications.

Write $w \in A$ to mean $w$ is a communication satisfying the session type $A$.

**Examples:**

- The *empty communication* $\bot \in A$.
- *Bit stream communications* are $(b0, w) \in \text{bits}$ and $(b1, w) \in \text{bits}$ where $w \in \text{bits}$. 
Observing Communications

\[ c_1 : A_1 \]
\[ \vdots \]
\[ c_n : A_n \]

\[ P \]

\[ c_0 : A_0 \]
Observing Communications

\[ \begin{align*}
  c_1 : A_1 & \quad \vdash \quad c_0 : A_0 \\
  c_n : A_n & \quad \vdash \quad c_0 : A_0
\end{align*} \]
Observing Communications

\[ P : c_0 : A_0 \rightsquigarrow (\langle n \rangle l, \langle r \rangle r) \]

\[ c_1 : A_1, \ldots, c_n : A_n \]

\[ w_1 \in A_1 \]

\[ \vdots \]

\[ w_n \in A_n \]

\[ w_0 \in A_0 \]

\[ c_0 : A_0 \]
\[ \langle c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 \rangle_{c_0, \ldots, c_n} = (c_0 : w_0, \ldots, c_n : w_n). \]
Consider the process $S$ sending a stream of zero bits:

\[ \vdash S :: i : \text{bits} \]
\[ i \leftarrow S = i.b0; i \leftarrow S \]
Consider the process $S$ sending a stream of zero bits:

$\vdash S :: i : \text{bits}$

$i \leftarrow S = i.b0; i \leftarrow S$
Consider the process $S$ sending a stream of zero bits:

$\vdash S :: i : \text{bits}$

$i \leftarrow S = i.b0; i \leftarrow S$

\[
\langle \vdash S :: i : \text{bits} \rangle_i = (i : (b0, (b0, (b0, \ldots))))
\]
Observing Communication Between Processes

\[ C \]

- \( b_1 : B_1 \)
- \( b_i : B_i \)
- \( b_m : B_m \)
- \( c_1 : A_1 \)
- \( c_j : A_j \)
- \( c_0 : A_0 \)
- \( c_n : A_n \)
- \( b_0 : B_0 \)
Observing Communication Between Processes

$c_1 : A_1$
$c_j : A_j$
$c_n : A_n$

$P$

$c_0 : A_0$
Observing Communication Between Processes

\[ b_1 : B_1 \]
\[ b_i : B_i \]
\[ b_m : B_m \]
\[ b_0 : B_0 \]
Observing Communication Between Processes

\[
\langle b_1 : B_1, \ldots, b_m : B_m \vdash C[P] :: b_0 : B_0 \rangle_{b_0, \ldots, b_m} = (b_0 : w_0, \ldots, b_m : w_m)
\]
Observing Communication Between Processes

\[
\langle b_1 : B_1, \ldots, b_m : B_m \vdash C[P] :: b_0 : B_0 \rangle_{b_0, \ldots, b_m} = (b_0 : w_0, \ldots, b_m : w_m)
\]

\[
\langle b_1 : B_1, \ldots, b_m : B_m \vdash C[P] :: b_0 : B_0 \rangle_{c_0, \ldots, c_n} = (c_0 : w'_0, \ldots, c_n : w'_n)
\]
More Example Observed Communications

\[ \begin{align*}
S & \quad \text{b0b0b0...} \quad \text{i:bits} \\
F & \quad \text{b1b1b1...} \quad \text{o:bits}
\end{align*} \]
More Example Observed Communications

S \[\text{b}_0 \text{b}_0 \text{b}_0 \ldots \] F \[\text{b}_1 \text{b}_1 \text{b}_1 \ldots \]
\[\text{i:bits} \quad \text{o:bits}\]
More Example Observed Communications

S

b0 b0 b0 ... i:bits

F

b1 b1 b1 ... o:bits
More Example Observed Communications

\[
\langle \vdash C[F] :: o : \text{bits} \rangle_o = (o : (b1, (b1, (b1, \ldots ))))
\]
More Example Observed Communications

\[ \langle \! \langle - C[F] :: o : \text{bits} \rangle \! \rangle_o = (o : (b1, (b1, (b1, \ldots)))) \]

\[ \langle \! \langle - C[F] :: o : \text{bits} \rangle \! \rangle_i = (i : (b0, (b0, (b0, \ldots)))) \]
More Example Observed Communications

\[
\begin{align*}
&\langle \vdash C[F] :: o : \text{bits} \rangle_o = (o : (b_1, (b_1, (b_1, \ldots)))) \\
&\langle \vdash C[F] :: o : \text{bits} \rangle_i = (i : (b_0, (b_0, (b_0, \ldots)))) \\
&\langle \vdash C[F] :: o : \text{bits} \rangle_{i,o} = (i : (b_0, \ldots), o : (b_1, \ldots))
\end{align*}
\]
Theorem

Observed communications are independent of the choice of fair execution.
Theorem: Observed communications are independent of the choice of fair execution.
Theorem

Observed communications are independent of the choice of fair execution.
Theorem

Observed communications are independent of the choice of fair execution.
Theorem

*Observed communications are independent of the choice of fair execution.*
Contributions

We give Polarized SILL

1. **An observed communication semantics**
2. **A communication-based testing equivalences framework**
3. **A communication-based denotational semantics**

and we use these semantics to reason about processes.
Communication-Based Testing
Equivalences
Main Idea: Two processes are equivalent if we cannot observe any differences through experimentation.
Performing Experiments

Processes $P$ and $Q$ are equivalent according to

are equivalent according to

if
Processes $P$ and $Q$ are internally communication equivalent if for all

\[ P = Q \]
An equivalence relation $\equiv$ is a congruence if

for all
Theorem

*Internal communication equivalence is not a congruence relation.*
External Communication Equivalence

Processes $P$ and $Q$ are **externally communication equivalent** if

$$P = Q$$

for all
Theorem

*External communication equivalence is a congruence relation.*
Theorem

*External communication equivalence is a congruence relation.*

**Barbed congruence** is the canonical notion of process equivalence.
Properties of External Communication Equivalence

**Theorem**

*External communication equivalence is a congruence relation.*

**Barbed congruence** is the canonical notion of process equivalence.

**Theorem**

*Processes are external communication equivalent if and only if they are barbed congruent.*
“Processes are equivalent if [...] for all”
Contributions

We give Polarized SILL

1. An observed communication semantics
2. A communication-based testing equivalences framework
3. A communication-based denotational semantics

and we use these semantics to reason about processes.
Denotational Semantics
The Denotational Approach

**Syntax / Programs**

**Mathematical Objects**

**Compositional**: the meaning of a program is a function of the meanings of its parts.
The Denotational Approach

**Compositional**: the meaning of a program is a function of the meanings of its parts.

Programs $C$ and $C'$ are **semantically equivalent** if $[C] = [C']$. 
A protocol $A$ denotes a domain $⟦A⟧$ of permissible communications.

A process $c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ denotes a continuous function $⟦P⟧ : ⟦A_1⟧ \times \cdots \times ⟦A_n⟧ \to ⟦A_0⟧$. 
Significance: “New” input does not affect “old” output.

If

\[
\begin{align*}
c_1: \text{bits} & \rightarrow b_0b_0 \rightarrow P \rightarrow b_1b_1 \rightarrow c_0: \text{bits}
\end{align*}
\]

then never

\[
\begin{align*}
c_1: \text{bits} & \rightarrow b_0b_0b_0 \rightarrow P \rightarrow b_0b_0 \rightarrow c_0: \text{bits}
\end{align*}
\]
Slogan: Processes cannot decide to send output only after observing entire infinite inputs.
Slogan: Processes cannot decide to send output only after observing entire infinite inputs.
**Slogan:** Processes cannot decide to send output only after observing entire infinite inputs.
**Slogan:** Processes cannot decide to send output only after observing entire infinite inputs.
Slogan: Processes cannot decide to send output only after observing entire infinite inputs.
**Slogan:** Processes cannot decide to send output only after observing entire infinite inputs.

\[
\begin{align*}
&b_0 b_1 \ldots \\
&\vdots \quad \vdots \\
&\vdots \\
&b_0 b_1 \\
&\vdots \\
&b_0 \\
\end{align*}
\]

\[P\]
Slogan: Processes cannot decide to send output only after observing entire infinite inputs.
Continuity

**Slogan:** Processes cannot decide to send output only after observing entire infinite inputs.

![Diagram](image_url)
Slogan: Processes cannot decide to send output only after observing entire infinite inputs.
The **polarity** of a protocol is the direction in which its messages flow on channels.
Splitting Channels

\[ \Delta \vdash P :: c : A \]

\[ c : A \rightarrow \Delta \times c \]

\[ \Delta \vdash P :: c : A \]
Splitting Channels

\[ \Delta \vdash P :: c : A \]

\[ [P] : "\Delta \times c" \rightarrow "\Delta \times c" \]
A protocol $A$ denotes the domains

- $[A]$ of negative (right-to-left) communications, and
- $[A]$ of positive (left-to-right) communications.

A process $c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ denotes a continuous function

$$[P] : [A_1] \times \cdots \times [A_n] \times [A_0] \rightarrow [A_1] \times \cdots \times [A_n] \times [A_0]$$
A **protocol** $A$ denotes a decomposition function

$$\langle A \rangle : [A] \to [A] \times [A]$$

from the domain $[A]$ of complete communications into the domains

- $[A]$ of positive (left-to-right) communications,
- $[A]$ of negative (right-to-left) communications.
A process $c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ denotes a continuous function

$$[P] : [A_1] \times \cdots \times [A_n] \times [A_0] \to [A_1] \times \cdots [A_n] \times [A_0]$$

that is compatible with the decompositions $\langle A_i \rangle : [A_i] \to [A_i] \times [A_i]$. 
The Functional Layer

- Simply-typed $\lambda$-calculus with a fixed-point operator
- Typing judgment: $\Psi \vdash M : \tau$
The Functional Layer

- Simply-typed $\lambda$-calculus with a fixed-point operator
- Typing judgment: $\Psi \vdash M : \tau$
- Standard denotational semantics:

$$\left[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \right] : \left[ \tau_1 \right] \times \cdots \times \left[ \tau_n \right] \rightarrow \left[ \tau \right]$$
The Functional Layer

- Simply-typed $\lambda$-calculus with a fixed-point operator
- Typing judgment: $\Psi \vdash M : \tau$
- Standard denotational semantics:

$$\left[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \right] : \left[ \tau_1 \right] \times \cdots \times \left[ \tau_n \right] \rightarrow \left[ \tau \right]$$
The Functional Layer

- Simply-typed $\lambda$-calculus with a fixed-point operator
- Typing judgment: $\Psi \vdash M : \tau$
- Standard denotational semantics:

$$\left[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \right] : \left[ \tau_1 \right] \times \cdots \times \left[ \tau_n \right] \rightarrow \left[ \tau \right]$$

- Includes quoted processes as a base type
The Functional Layer

- Simply-typed $\lambda$-calculus with a fixed-point operator
- Typing judgment: $\Psi \vdash M : \tau$
- Standard denotational semantics:

$$\left[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \right] : \left[ \tau_1 \right] \times \cdots \times \left[ \tau_n \right] \rightarrow \left[ \tau \right]$$

- Includes quoted processes as a base type

Processes can depend on functional values through contexts $\Psi$:

$$\Psi; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$$
The Functional Layer

- Simply-typed $\lambda$-calculus with a fixed-point operator
- Typing judgment: $\Psi \vdash M : \tau$
- Standard denotational semantics:

\[
\left[ x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau \right] : \left[ \tau_1 \right] \times \cdots \times \left[ \tau_n \right] \to \left[ \tau \right]
\]

- Includes quoted processes as a base type

Processes can depend on functional values through contexts $\Psi$:

\[
\Psi ; c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0
\]

Processes now denote continuous functions

\[
\left[ P \right] : \left[ \Psi \right] \to \left[ \left[ A_1 \right] \times \cdots \times \left[ A_n \right] \times \left[ A_0 \right] \right]
\]

\[
\to \left[ A_1 \right] \times \cdots \times \left[ A_n \right] \times \left[ A_0 \right]
\]
Soundness

Recall that processes $P$ and $Q$ are **denotationally equivalent** if $⟦P⟧ = ⟦Q⟧$. 

Theorem

If two processes are denotationally equivalent, then they are external communication equivalent and barbed congruent.
Recall that processes $P$ and $Q$ are **denotationally equivalent** if $\langle P \rangle = \langle Q \rangle$.

**Theorem**

*If two processes are denotationally equivalent, then they are external communication equivalent and barbed congruent.*
Contributions

We give Polarized SILL

1. An observed communication semantics
2. A communication-based testing equivalences framework
3. A communication-based denotational semantics

and we use these semantics to reason about processes.
Communication-based semantics elucidate the structure of session-typed languages and allow us to reason about programs written in these languages.
1. Modelling recursive types required new techniques for reasoning about parametrized fixed points of functors [MFPS’20]
Other Results

1. Modelling recursive types required new techniques for reasoning about parametrized fixed points of functors [MFPS’20]
2. A study of fairness for multiset rewriting systems [EXPRESS/SOS’20]
Other Results

1. Modelling recursive types required new techniques for reasoning about parametrized fixed points of functors [MFPS’20]
2. A study of fairness for multiset rewriting systems [EXPRESS/SOS’20]
3. A collection of case studies to which I apply these techniques
Future Work

1. Applications to richer protocols, e.g., dependent protocols
2. Applications to richer communication topologies, e.g., multicast
Acknowledgements
Communication-based semantics elucidate the structure of session-typed languages and allow us to reason about programs written in these languages.
Backup Slides
Relation to Deterministic Networks

My semantics generalizes Kahn’s 1974 semantics for deterministic networks to support:

1. session-typed communication instead of streams of values of simple type like integers or booleans
2. *bidirectional* communication instead of unidirectional streams of values

Generalizing Kahn-style semantics to handle non-determinism is difficult because of the Keller and Brock-Ackerman anomalies. Though execution in Polarized SILL is non-deterministic, its processes have deterministic input/output behaviour.
My semantics exists in a GoI construction $\mathcal{G}(\text{CPO})$:

- Objects are pairs $(A^+, A^-)$ of objects $A^+, A^-$ from $\text{CPO}$
- Morphisms $f : (A^+, A^-) \to (B^+, B^-)$ are morphisms $\hat{f} : A^+ \times B^- \to A^- \times B^+$ in $\text{CPO}$
- Composition $g \circ f$ is $\text{Tr}(\hat{g} \times \hat{f})$

Expressing my semantics in this construction:

\[
\left[ \Delta_1, \Delta_2 \vdash c \leftarrow P; \ Q :: d : D \right] \\
= \left[ \Delta_2, c : C \vdash Q :: d : D \right] \circ \left[ \Delta_1 \vdash P :: c : C \right]
\]
• Abramsky and Jagadeesan (1994) use this construction to give a type-free interpretation of classical linear logic where all types denote the same “universal domain”

• Abramsky, Haghverdi, and Scott (2002) use it to give an algebraic framework for Girard’s Geometry of Interaction

• I use it to give a semantics that captures the computational aspects of a programming language with recursion
Relation to Atkey’s Denotational Semantics

In Atkey’s denotational semantics for CP:

- Protocols denote sets of communications
- \([\vdash P :: \Gamma] \subset [\Gamma]\) is a relation containing the possible observed communications on its free channels, e.g.,

\[
\begin{align*}
[\vdash x \leftrightarrow y :: x : A, y : A^\perp] &= \{(a, a) \mid a \in [A]\} \\
[1] &= [1^\perp] = \{\ast\} \\
[\vdash x[] :: x : 1] &= \{(*)\} \\
[\vdash x(). P :: \Gamma, x : 1^\perp] &= \{(\gamma, \ast) \mid \gamma \in [\vdash P :: \Gamma]\} \\
[\vdash \nu x. (P | Q) :: \Gamma, \Delta] &= \{(\gamma, \delta) \mid (\gamma, a) \in [\vdash P :: \Gamma, x : A], \\
& \quad (\delta, a) \in [\vdash Q :: \Delta, x : A^\perp]\}
\end{align*}
\]