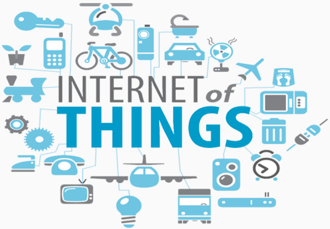
# Communication-Based Semantics for Recursive Session-Typed Processes

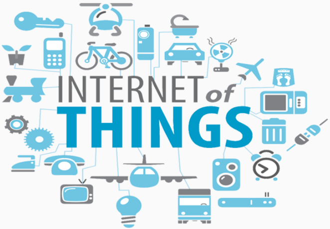
Ryan Kavanagh September 28, 2021

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

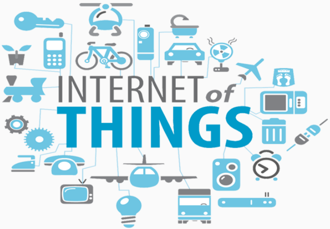
# Communicating Systems and Session Types

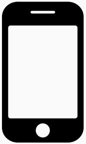


# Communicating Systems and Session Types

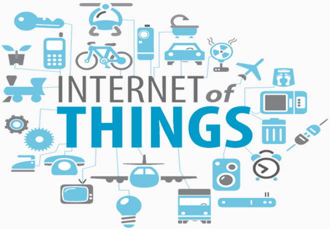


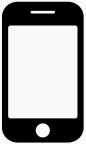
# Communicating Systems and Session Types



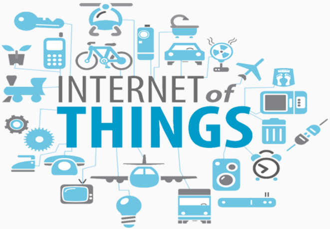



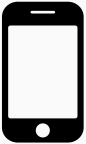
# Communicating Systems and Session Types



# Communicating Systems and Session Types



Programs written in session-typed programming languages are guaranteed to obey their protocols.

# Key Technique: Program Equivalence

*"Program equivalence is arguably one of the most interesting and at the same time important problems in formal verification."*[1]

[1] Lahiri et. al. "Program Equivalence", Dagstuhl Reports, Vol. 8, Issue 4, pp. 1–19, 2018.

# Some Existing Approaches to Equivalence

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

# Some Existing Approaches to Equivalence

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:

- Wadler's Classical Processes (CP): Atkey [2017] gives a relational semantics.
- Hypersequent CP: Kokke et al. [2019] give a denotational semantics using Brzozowski derivatives.
- 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:

- Wadler's Classical Processes (CP): Atkey [2017] gives a relational semantics.
- Hypersequent CP: Kokke et al. [2019] give a denotational semantics using Brzozowski derivatives.
- 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:

- Wadler's Classical Processes (CP): Atkey [2017] gives a relational semantics.
- Hypersequent CP: Kokke et al. [2019] give a denotational semantics using Brzozowski derivatives.
- 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.

# Reasoning About Programs in Rich 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*

# Reasoning About Programs in Rich Languages

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

# Back to Fundamentals

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.

# The Key Premise

Communication is the only observable phenomenon of processes!

# Thesis Statement

*Communication-based semantics elucidate the structure of session-typed languages and allow us to reason about programs written in these languages.*

# Our Laboratory: Polarized SILL

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**

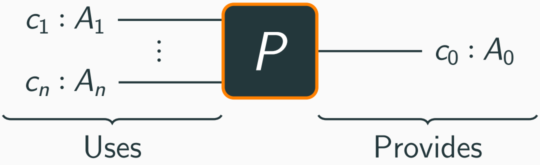
and we use these semantics to reason about processes.

# Polarized SILL


Where

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

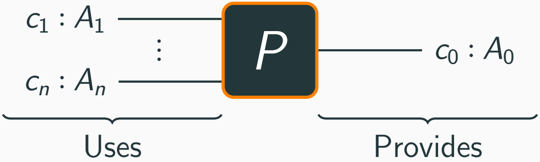
# Polarized SILL




Where

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

# Polarized SILL




Abbreviate as:

$$c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 \qquad (n \geq 0)$$

$$\Delta \vdash P :: c_0 : A_0$$

where $\Delta = c_1 : A_1, \ldots, c_n : A_n$.

# Bit Streams in SILL

Bit stream protocol:

$$\texttt{bits} = (\texttt{b0} : \texttt{bits}) \oplus (\texttt{b1} : \texttt{bits})$$

Example communications satisfying `bits`:

$$\xrightarrow{\qquad\texttt{b0}\,\texttt{b1}\,\texttt{b0}\,\texttt{b0}\cdots\qquad} c_0 : \texttt{bits}$$

$$\xrightarrow{\qquad\texttt{b1}\qquad} c_1 : \texttt{bits}$$

$$\xrightarrow{\qquad\qquad} c_2 : \texttt{bits}$$

# Bit Streams in SILL

Bit stream protocol:

$$\texttt{bits} = (\texttt{b0} : \texttt{bits}) \oplus (\texttt{b1} : \texttt{bits})$$

Example communications satisfying $\texttt{bits}$:

$$\xrightarrow{\texttt{b0}\,\texttt{b1}\,\texttt{b0}\,\texttt{b0}\cdots} c_0 : \texttt{bits}$$

$$\xrightarrow{\texttt{b1}\,\bot} c_1 : \texttt{bits}$$

$$\xrightarrow{\bot} c_2 : \texttt{bits}$$

# Flipping Bits


```
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 : bits |- F :: o : bits
o <- F <- i = case i { b0 => o.b1; o <- F <- i
                     | b1 => o.b0; o <- F <- i }
```

# Flipping Bits


```
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 : bits |- F :: o : bits
o <- F <- i = case i { b0 => o.b1; o <- F <- i
                     | b1 => o.b0; o <- F <- i }
```

# Flipping Bits


```
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 : bits |- F :: o : bits
o <- F <- i = case i { b0 => o.b1; o <- F <- i
                     | b1 => o.b0; o <- F <- i }
```

# Flipping Bits


```
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 : bits |- F :: o : bits
o <- F <- i = case i { b0 => o.b1; o <- F <- i
                     | b1 => o.b0; o <- F <- i }
```

# Flipping Bits


```
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 : bits |- F :: o : bits
o <- F <- i = case i { b0 => o.b1; o <- F <- i
                     | b1 => o.b0; o <- F <- i }
```

# Flipping Bits


```
i : bits |- F :: o : bits
o <- F <- i = case i { b0 => o.b1; o <- F <- i
                     | b1 => o.b0; o <- F <- i }
```

# Observed Communication Semantics

# Observed Communication Semantics

**Idea:** The meaning of a process is the communications we observe during its execution.

# Observed Communication Semantics

**Idea:** The meaning of a process is the communications we observe during its execution.

**Questions:**

1. What are observed communications?

# Observed Communication Semantics

**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?

# Session-Typed Communications

A session type specifies permitted communications.

# Session-Typed Communications

A session type specifies permitted communications.

Write $w \mathrel{\varepsilon} A$ to mean $w$ is a communication satisfying the session type $A$.

# Session-Typed Communications

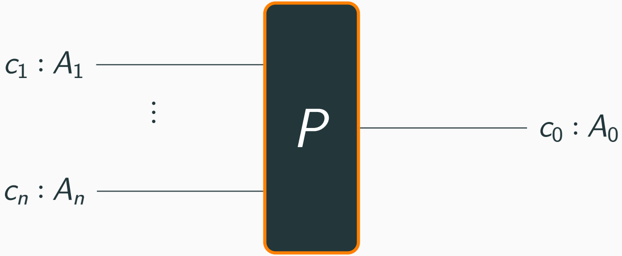
A session type specifies permitted communications.

Write $w \mathrel{\varepsilon} A$ to mean $w$ is a communication satisfying the session type $A$.

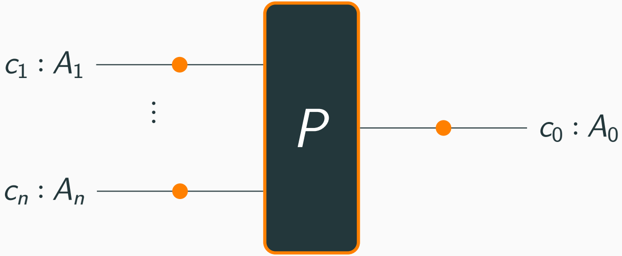
**Examples:**

- The **empty communication** $\perp \mathrel{\varepsilon} A$.
- **Bit stream communications** are $(\mathtt{b0}, w) \mathrel{\varepsilon} \mathtt{bits}$ and $(\mathtt{b1}, w) \mathrel{\varepsilon} \mathtt{bits}$ where $w \mathrel{\varepsilon} \mathtt{bits}$.

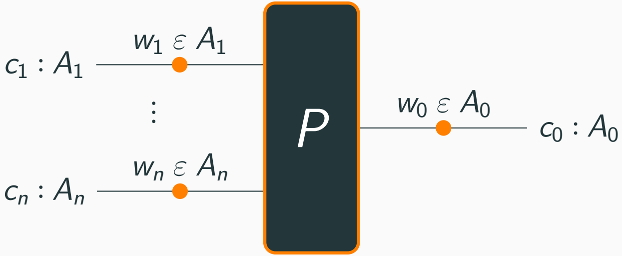
# Observing Communications



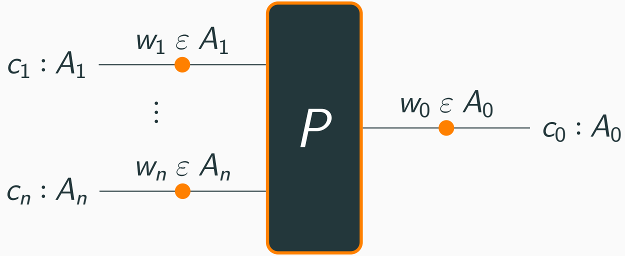
# Observing Communications



# Observing Communications



# Observing Communications




$$\langle c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 \rangle_{c_0, \ldots, c_n} = (c_0 : w_0, \ldots, c_n : w_n).$$

# Example Observed Communications

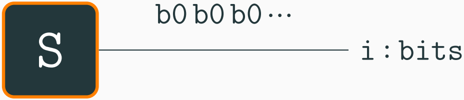
Consider the process `S` sending a stream of zero bits:

```
⊢ S :: i : bits
i <- S = i.b0; i <- S
```

# Example Observed Communications

Consider the process S sending a stream of zero bits:

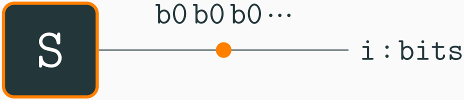
```
⊢ S :: i : bits
i <- S = i.b0; i <- S
```



# Example Observed Communications

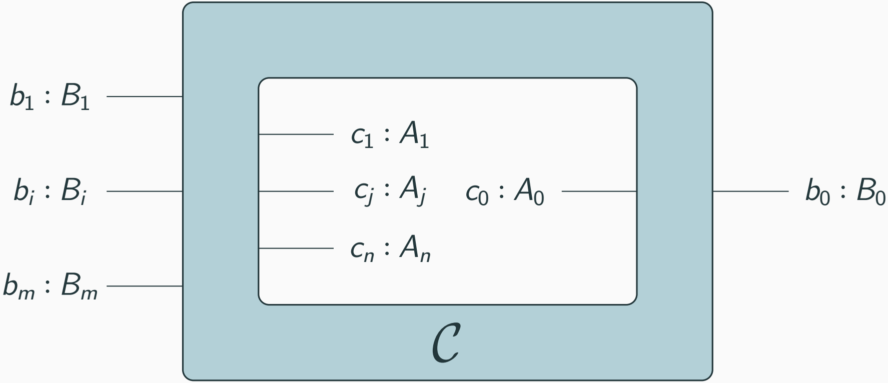
Consider the process S sending a stream of zero bits:

```
⊢ S :: i : bits
i <- S = i.b0; i <- S
```

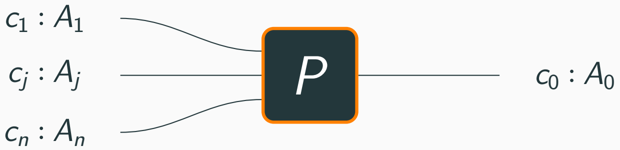



$$\langle\!\langle \vdash \mathtt{S} :: \mathtt{i} : \mathtt{bits} \rangle\!\rangle_{\mathtt{i}} = (\mathtt{i} : (\mathtt{b0}, (\mathtt{b0}, (\mathtt{b0}, \ldots))))$$

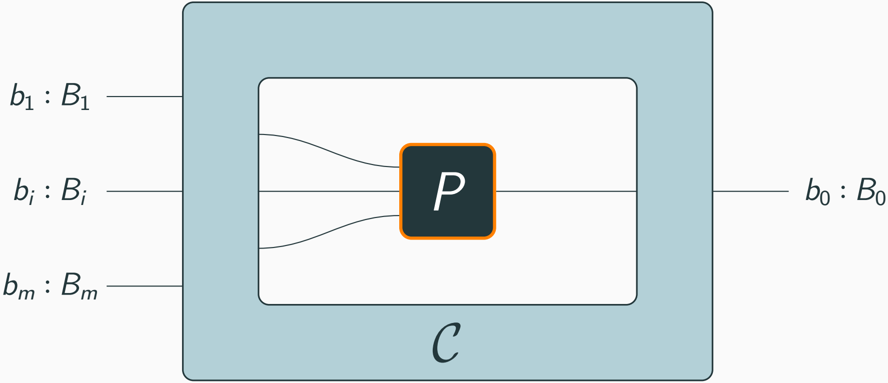
# Observing Communication Between Processes



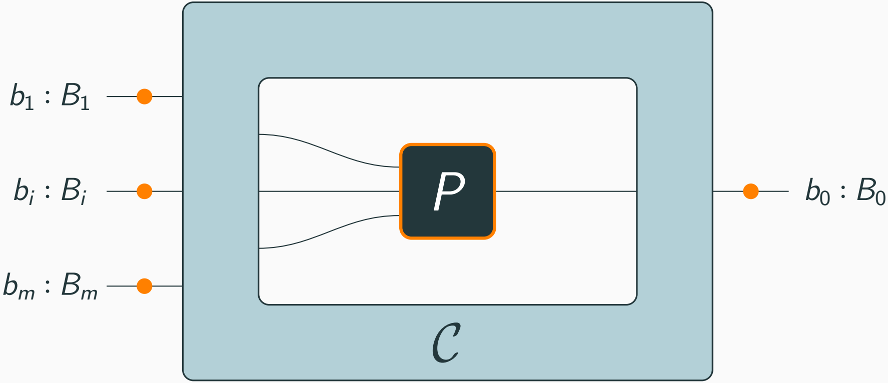
# Observing Communication Between Processes



# Observing Communication Between Processes

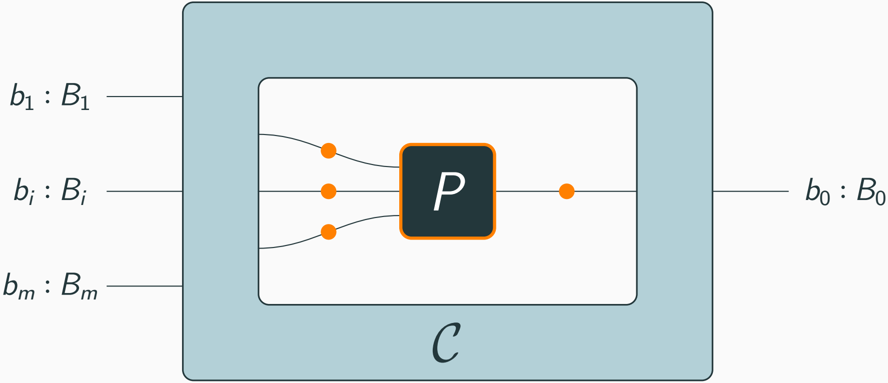


# Observing Communication Between Processes




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

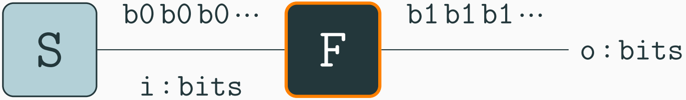
# Observing Communication Between Processes



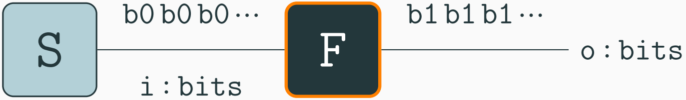

$$\langle b_1 : B_1, \ldots, b_m : B_m \vdash \mathcal{C}[P] :: b_0 : B_0 \rangle_{b_0,\ldots,b_m} = (b_0 : w_0, \ldots, b_m : w_m)$$

$$\langle b_1 : B_1, \ldots, b_m : B_m \vdash \mathcal{C}[P] :: b_0 : B_0 \rangle_{c_0,\ldots,c_n} = (c_0 : w_0', \ldots, c_n : w_n')$$

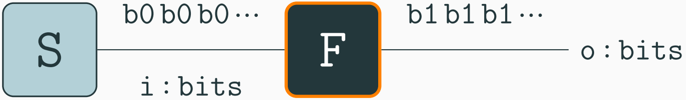
# More Example Observed Communications



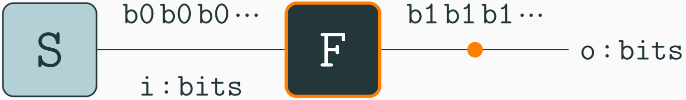
# More Example Observed Communications



# More Example Observed Communications

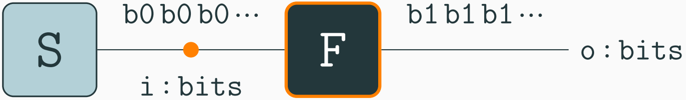


# More Example Observed Communications




$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_\circ = (\mathtt{o} : (\mathtt{b1}, (\mathtt{b1}, (\mathtt{b1}, \ldots))))$$

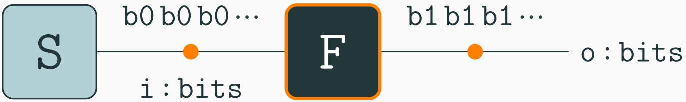
# More Example Observed Communications




$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_{\mathtt{o}} = (\mathtt{o} : (\mathtt{b1}, (\mathtt{b1}, (\mathtt{b1}, \ldots))))$$

$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_{\mathtt{i}} = (\mathtt{i} : (\mathtt{b0}, (\mathtt{b0}, (\mathtt{b0}, \ldots))))$$

# More Example Observed Communications




$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_{\mathtt{o}} = (\mathtt{o} : (\mathtt{b1}, (\mathtt{b1}, (\mathtt{b1}, \ldots))))$$

$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_{\mathtt{i}} = (\mathtt{i} : (\mathtt{b0}, (\mathtt{b0}, (\mathtt{b0}, \ldots))))$$

$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_{\mathtt{i},\mathtt{o}} = (\mathtt{i} : (\mathtt{b0}, \ldots), \mathtt{o} : (\mathtt{b1}, \ldots))$$

# Fairness

# Fairness

# Fairness

# Fairness


$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_{\mathtt{i},\mathtt{o}} = (\mathtt{i} : (\mathtt{b0}, (\mathtt{b0}, \ldots)), \mathtt{o} : \bot)$$

# Fairness


$$\langle\!\langle \vdash \mathcal{C}[\mathtt{F}] :: \mathtt{o} : \mathtt{bits} \rangle\!\rangle_{\mathtt{i},\mathtt{o}} = (\mathtt{i} : (\mathtt{b0}, (\mathtt{b0}, \ldots)), \mathtt{o} : \bot)$$

**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**

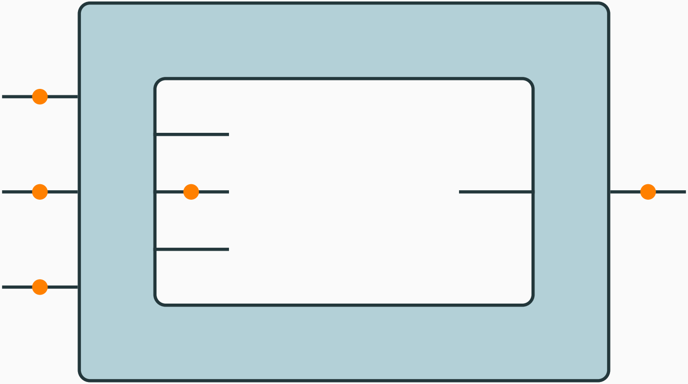
and we use these semantics to reason about processes.

# Communication-Based Testing Equivalences

# Testing Equivalences

**Main Idea:** Two processes are equivalent if we cannot observe any differences through experimentation.

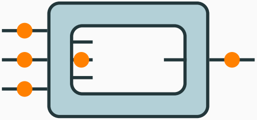
# Communication Experiments

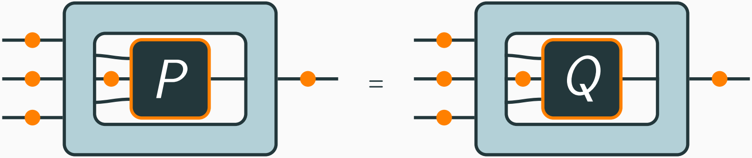


# Performing Experiments

Processes 

and 


are **equivalent** according to  if

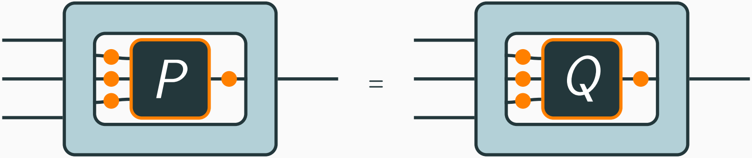


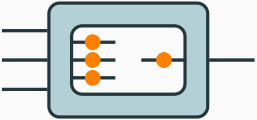
# Internal Communication Equivalence

Processes 

and 


are **internally communication equivalent** if



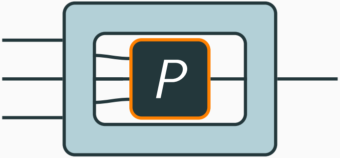

for all 

# Congruence Relations

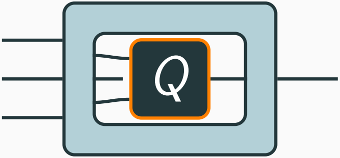
An equivalence relation ≡ is a **congruence** if


≡

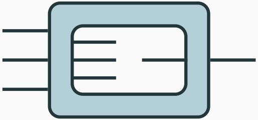

implies




≡




for all



# Not A Congruence

**Theorem**

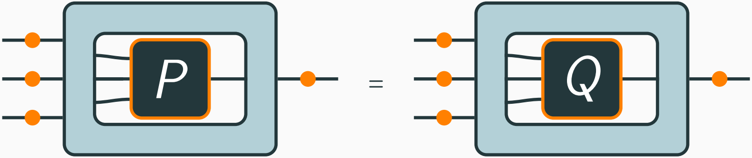
*Internal communication equivalence is not a congruence relation.*

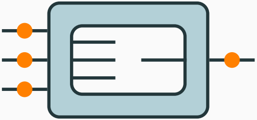
# External Communication Equivalence

Processes 

and 


are **externally communication equivalent** if




for all 

# Properties of External Communication Equivalence

**Theorem**

*External communication equivalence is a congruence relation.*

# Properties of External Communication Equivalence

**Theorem**

*External communication equivalence is a congruence relation.*

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

# Properties of External Communication Equivalence

**Theorem**

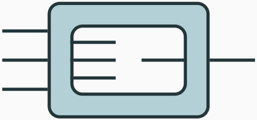
*External communication equivalence is a congruence relation.*

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

**Theorem**

*Processes are external communication equivalent if and only if they are barbed congruent.*

## New Semantics, Same Refrain

"Processes are equivalent if [. . . ] for all  "

# Contributions

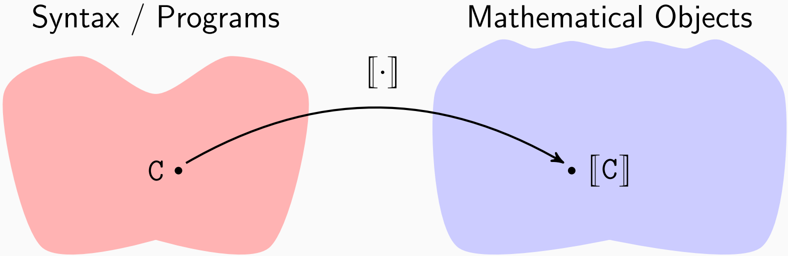
We give Polarized SILL

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.

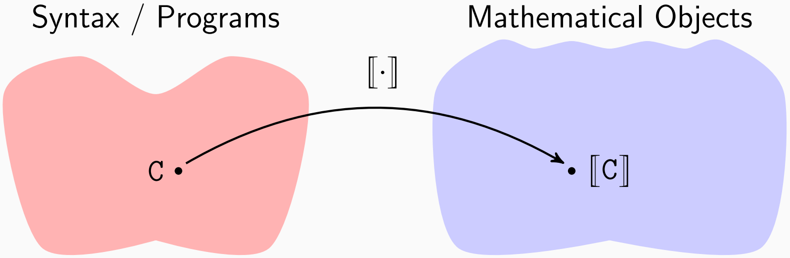
# Denotational Semantics

# The Denotational Approach




**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 $\mathtt{C}$ and $\mathtt{C}'$ are **semantically equivalent** if $[\![\mathtt{C}]\!] = [\![\mathtt{C}']\!]$.

# Denoting Protocols and Processes

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

# Monotonicity

**Significance:** “New” input does not affect “old” output.

If

$$c_1 : \mathtt{bits} \xrightarrow{\mathtt{b0}\ \mathtt{b0}} \boxed{P} \xrightarrow{\mathtt{b1}\ \mathtt{b1}} c_0 : \mathtt{bits}$$

then never

$$c_1 : \mathtt{bits} \xrightarrow{\mathtt{b0}\ \mathtt{b0}\ \mathtt{b0}} \boxed{P} \xrightarrow{\mathtt{b0}\ \mathtt{b0}} c_0 : \mathtt{bits}$$

# Continuity

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

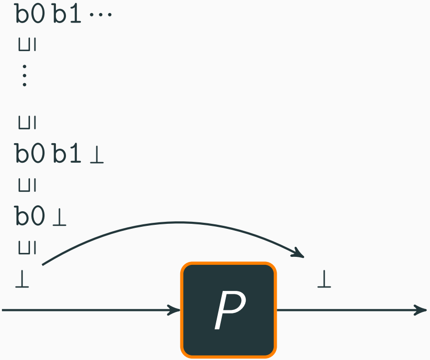
# Continuity

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

$$
\begin{array}{l}
\mathtt{b0}\,\mathtt{b1}\cdots \\
\sqsubseteq \\
\vdots \\
\sqsubseteq \\
\mathtt{b0}\,\mathtt{b1}\,\bot \\
\sqsubseteq \\
\mathtt{b0}\,\bot \\
\sqsubseteq \\
\bot
\end{array}
$$

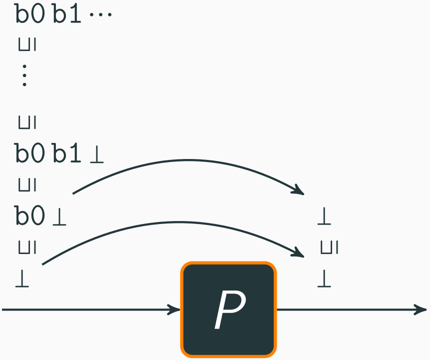
# Continuity

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



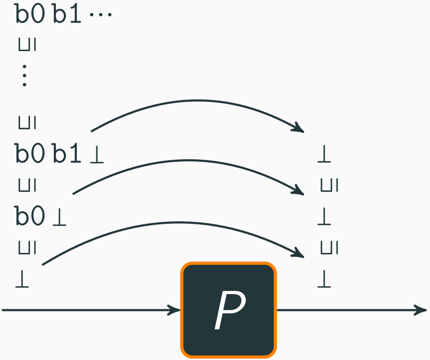
# Continuity

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



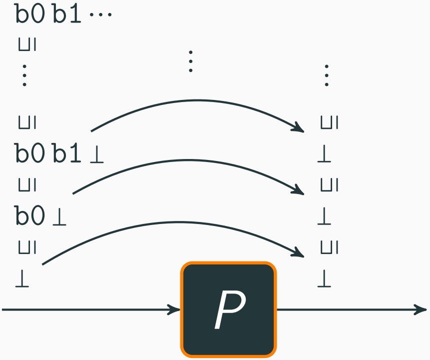
# Continuity

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



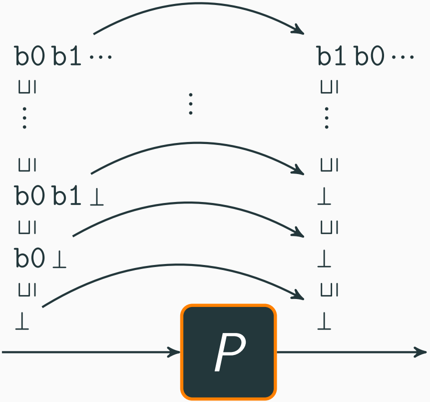
# Continuity

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



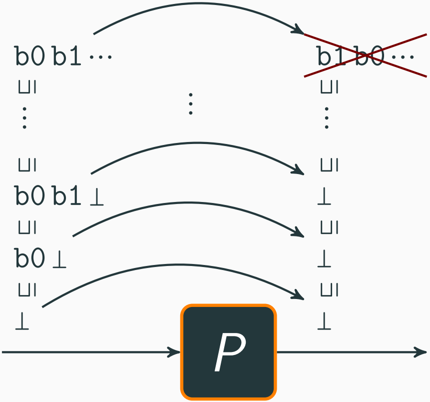
# Continuity

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



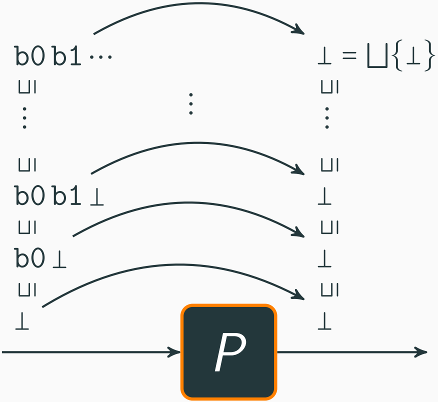
# Continuity

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



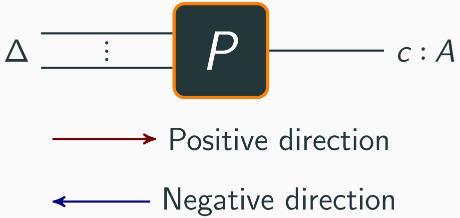
# Continuity

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

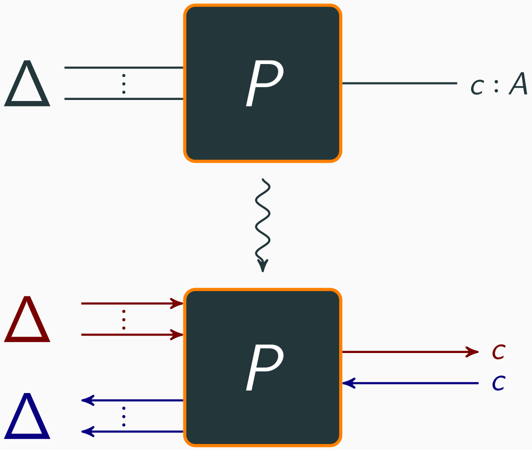


# Polarity of Communication

The **polarity** of a protocol is the direction in which its messages flow on channels.

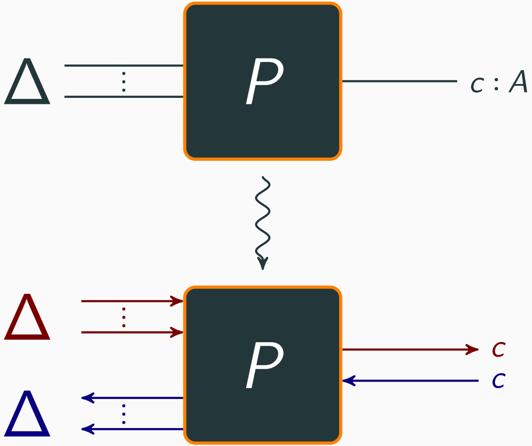


# Splitting Channels




$\Delta \vdash P :: c : A$

# Splitting Channels




$$\Delta \vdash P :: c : A$$

$$[\![P]\!] : \text{“}\Delta \times c\text{”} \to \text{“}\Delta \times c\text{”}$$

# The Bidirectional Version

A **protocol** $A$ denotes the domains

- $\llbracket A \rrbracket$ of negative (right-to-left) communications, and
- $\llbracket A \rrbracket$ 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

$$\llbracket P \rrbracket : \llbracket A_1 \rrbracket \times \cdots \times \llbracket A_n \rrbracket \times \llbracket A_0 \rrbracket \to \\ \to \llbracket A_1 \rrbracket \times \cdots \times \llbracket A_n \rrbracket \times \llbracket A_0 \rrbracket$$

# Decomposing Communications

A **protocol** $A$ denotes a decomposition function

$$\langle A \rangle : [\![A]\!] \to {\color{red}[\![A]\!]} \times {\color{blue}[\![A]\!]}$$

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

- ${\color{red}[\![A]\!]}$ of positive (left-to-right) communications,
- ${\color{blue}[\![A]\!]}$ of negative (right-to-left) communications.

# Processes "Fill In" Partial Communications

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

$$\begin{aligned}[\![P]\!] : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \times [\![A_0]\!] \to \\ \to [\![A_1]\!] \times \cdots [\![A_n]\!] \times [\![A_0]\!]\end{aligned}$$

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

# 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:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash M : \tau]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$$

# The Functional Layer

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

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash M : \tau]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$$

# The Functional Layer

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

$$\llbracket x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash M : \tau \rrbracket : \llbracket \tau_1 \rrbracket \times \cdots \times \llbracket \tau_n \rrbracket \to \llbracket \tau \rrbracket$$

- 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:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash M : \tau]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$$

- 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:

$$[\![x_1 : \tau_1, \ldots, x_n : \tau_n \Vdash M : \tau]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$$

- 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

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

# Soundness

Recall that processes $P$ and $Q$ are **denotationally equivalent** if $\llbracket P \rrbracket = \llbracket Q \rrbracket$.

# 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.*

# 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.

# Thesis Statement

*Communication-based semantics elucidate
the structure of session-typed languages and allow us
to reason about programs written in these languages.*

# Other Results

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

# Thesis Statement

*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.

# Relation to the Geometry of Interaction I

My semantics exists in a GoI construction $\mathcal{G}(\mathbf{CPO})$:

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

Expressing my semantics in this construction:

$$\begin{aligned}&[\![\Delta_1, \Delta_2 \vdash c \leftarrow P;\ \ Q :: d : D]\!] \\ &= [\![\Delta_2, c : C \vdash Q :: d : D]\!] \circ [\![\Delta_1 \vdash P :: c : C]\!]\end{aligned}$$

# Relation to the Geometry of Interaction II

- 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.,

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