# Channel-Dependent Session Types

**Ryan Kavanagh**, Brigitte Pientka

NJ Programming Languages and Systems Seminar, May 2022

McGill University

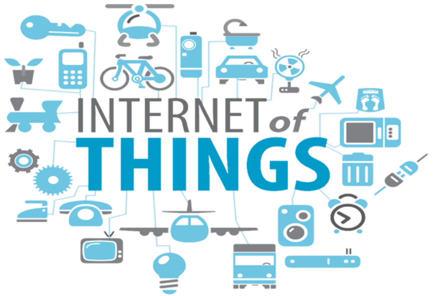
2022-05-19

Channel-Dependent Session Types

Channel-Dependent Session Types

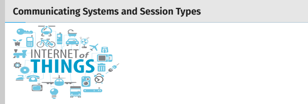
**Ryan Kavanagh**, Brigitte Pientka
NJ Programming Languages and Systems Seminar, May 2022
McGill University

# Communicating Systems and Session Types



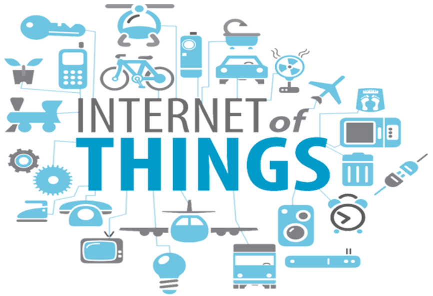





1. Communicating systems are ubiquitous
2. To work, every component must communicate with the others according to rules called protocols
3. Failure to do so can lead to vulnerabilities like Heartbleed
4. Caused by failure to implement TLS Heartbeat protocol extension.
5. Estimated cost to industry: over $500 million
6. Session-typed languages can help
7. Analogous to data types, but for communication
8. Today's talk: How to capture more expressive protocols

# Communicating Systems and Session Types



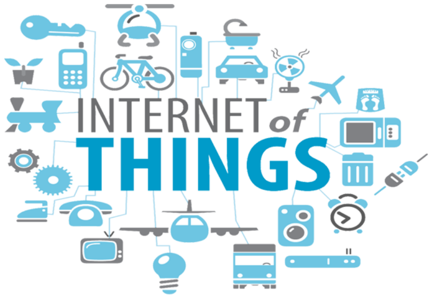


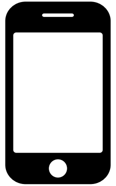




1. Communicating systems are ubiquitous
2. To work, every component must communicate with the others according to rules called protocols
3. Failure to do so can lead to vulnerabilities like Heartbleed
4. Caused by failure to implement TLS Heartbeat protocol extension.
5. Estimated cost to industry: over $500 million
6. Session-typed languages can help
7. Analogous to data types, but for communication
8. Today’s talk: How to capture more expressive protocols

# Communicating Systems and Session Types



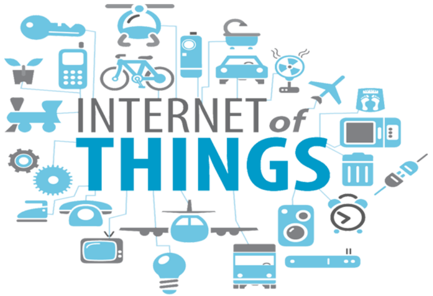



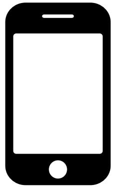





1. Communicating systems are ubiquitous
2. To work, every component must communicate with the others according to rules called protocols
3. Failure to do so can lead to vulnerabilities like Heartbleed
4. Caused by failure to implement TLS Heartbeat protocol extension.
5. Estimated cost to industry: over $500 million
6. Session-typed languages can help
7. Analogous to data types, but for communication
8. Today’s talk: How to capture more expressive protocols

# Communicating Systems and Session Types



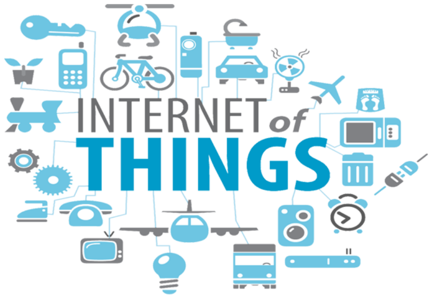


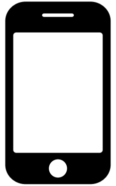


Communicating Systems and Session Types

1. Communicating systems are ubiquitous
2. To work, every component must communicate with the others according to rules called protocols
3. Failure to do so can lead to vulnerabilities like Heartbleed
4. Caused by failure to implement TLS Heartbeat protocol extension.
5. Estimated cost to industry: over $500 million
6. Session-typed languages can help
7. Analogous to data types, but for communication
8. Today's talk: How to capture more expressive protocols

# Communicating Systems and Session Types

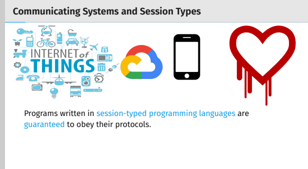


Programs written in session-typed programming languages are guaranteed to obey their protocols.

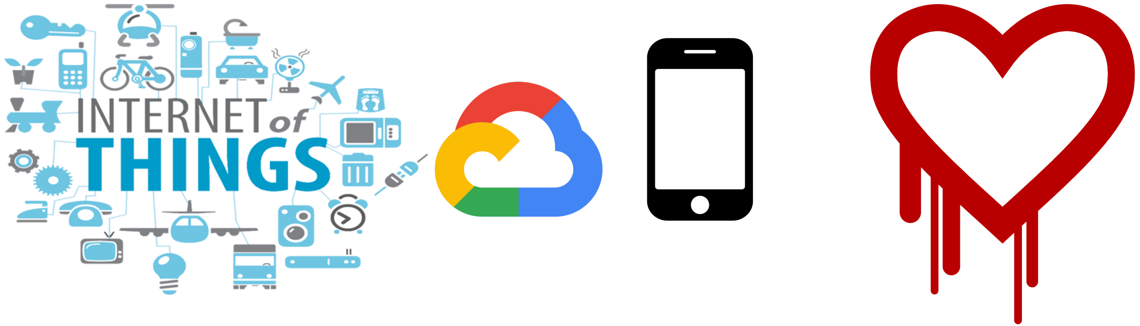






1. Communicating systems are ubiquitous
2. To work, every component must communicate with the others according to rules called protocols
3. Failure to do so can lead to vulnerabilities like Heartbleed
4. Caused by failure to implement TLS Heartbeat protocol extension.
5. Estimated cost to industry: over $500 million
6. Session-typed languages can help
7. Analogous to data types, but for communication
8. Today’s talk: How to capture more expressive protocols

# Communicating Systems and Session Types




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

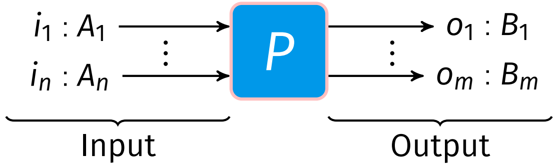
**Today:** How can we capture more expressive protocols?






1. Communicating systems are ubiquitous
2. To work, every component must communicate with the others according to rules called protocols
3. Failure to do so can lead to vulnerabilities like Heartbleed
4. Caused by failure to implement TLS Heartbeat protocol extension.
5. Estimated cost to industry: over $500 million
6. Session-typed languages can help
7. Analogous to data types, but for communication
8. Today's talk: How to capture more expressive protocols

# Processes and Session-Typed Channels

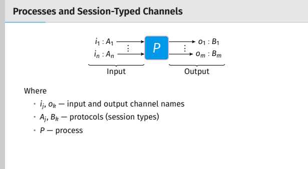



Where

- $i_j$, $o_k$ — input and output channel names
- $A_j$, $B_k$ — protocols (session types)
- $P$ — process

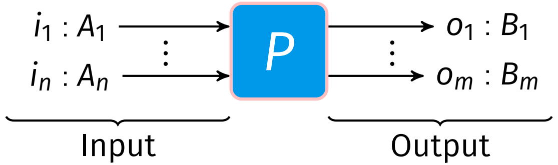






1. Think process as black boxes communicating over wires.
2. Wires are called “channels”; communication should respect a protocol.
3. The protocol specifies what kind of message can be transmitted next.
4. Protocols evolve over the course of communication to allow for different kinds of messages.
5. Make clear that channels and protocols are different.
6. In general, communication is bidirectional, but today, assume left to right.

# Processes and Session-Typed Channels



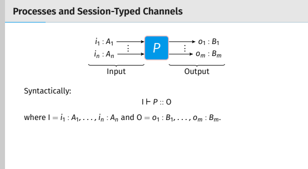

Syntactically:

$$\mathrm{I} \vdash P :: \mathrm{O}$$

where $\mathrm{I} = i_1 : A_1, \ldots, i_n : A_n$ and $\mathrm{O} = o_1 : B_1, \ldots, o_m : B_m$.







1. Think process as black boxes communicating over wires.
2. Wires are called “channels”; communication should respect a protocol.
3. The protocol specifies what kind of message can be transmitted next.
4. Protocols evolve over the course of communication to allow for different kinds of messages.
5. Make clear that channels and protocols are different.
6. In general, communication is bidirectional, but today, assume left to right.

# Bit Streams

Bit stream protocol:

$$\text{bits} = (\text{b0} \fatsemi \text{bits}) \oplus (\text{b1} \fatsemi \text{bits})$$

Example communications satisfying `bits`:

$$\xrightarrow{\text{b0 b1 b0 b0}\cdots} c : \text{bits}$$





Bit Streams
Bit stream protocol:
bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)
Example communications satisfying bits:
b0 b1 b0 b0 ··· c : bits

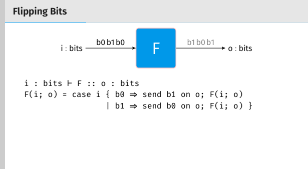
1. Recurring example throughout this talk — bit streams
2. We can also deal with more interesting features like queues and stacks or channel transmission, but bit streams are useful for illustrating key features.
3. Protocol specifies what communications can be sent on a channel.
4. A communication is a sequence of messages.
5. This is a recursive protocol.
6. Send a bit, and then say that the remainder of the communication will follow the bits protocol: protocols change

# Flipping Bits


```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```







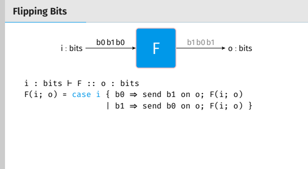
1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

# Flipping Bits


```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```

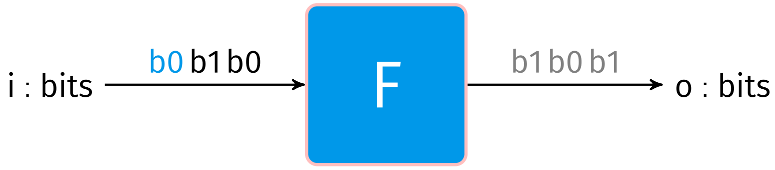






1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

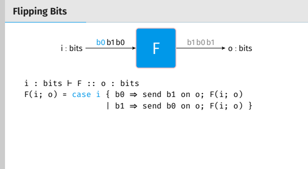
# Flipping Bits




```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```

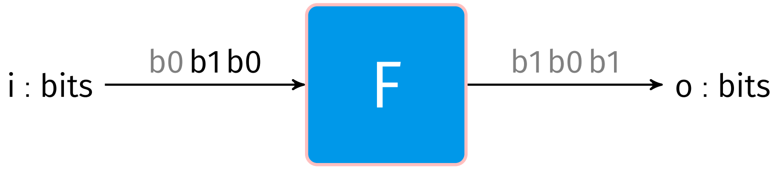






1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

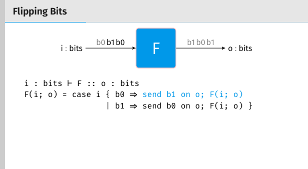
# Flipping Bits




```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```

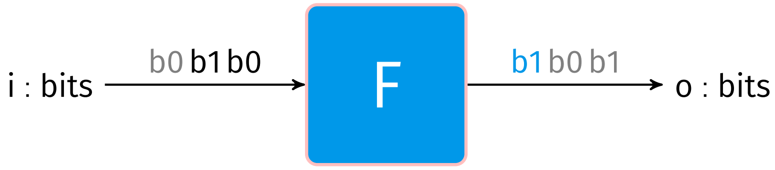






1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

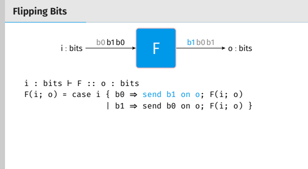
# Flipping Bits




```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```

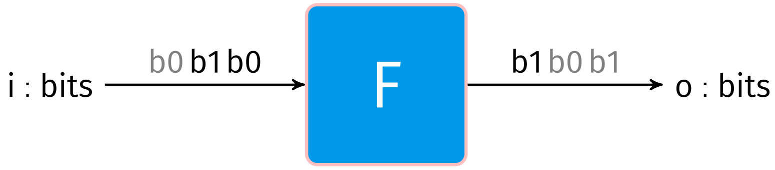






1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

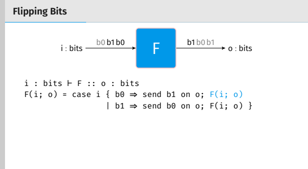
# Flipping Bits




```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```







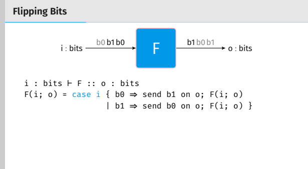
1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

# Flipping Bits


```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```







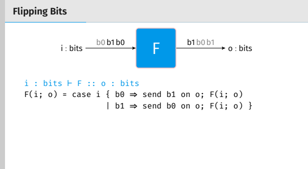

1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

# Flipping Bits


```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```







1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

# Flipping Bits

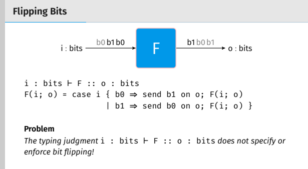

```
i : bits ⊢ F :: o : bits
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```

**Problem**

*The typing judgment* `i : bits ⊢ F :: o : bits` *does not specify or enforce bit flipping!*







1. The bit flipping process F uses the input channel *i* that satisfies bits, and provides an output channel *o* that satisfies bits.
2. *i* and *o* are channel names; bits is the protocol
3. We can think of the typing judgment as a spec for *F*.
4. The typing judgment isn't very precise: the identity function satisfies the same specification.
5. EMPHASIZE MULTITUDE OF DIFFERENT PROCESSES
6. Want to make typing judgments capture more precise invariants relating input and output.
7. Treat session types as processes that can observe communications to produce more precise specifications.

# Extending Session Types With Observing Processes

| | |
|---|---|
| $A, B \coloneqq \cdots$ | other session types |
| $\mid (l \mathbin{⨾} A) \oplus (r \mathbin{⨾} B)$ | labelled choice |





Extending Session Types With Observing Processes
$A, B \coloneqq \cdots$ other session types
$\mid (l \mathbin{⨾} A) \oplus (r \mathbin{⨾} B)$ labelled choice

1. Only giving syntax for binary choice, but assume it for any finite arity
2. Use definitional equality to capture type-level computation

# Extending Session Types With Observing Processes

| | |
|---|---|
| $A, B \coloneqq \cdots$ | other session types |
| $\mid (l \mathbin{⨟} A) \oplus (r \mathbin{⨟} B)$ | labelled choice |
| $\mid \mathrm{CASE}\ c\ \{l \Rightarrow A \mid r \Rightarrow B\}$ | label observation |





Extending Session Types With Observing Processes
A, B ≔ ··· other session types
| (l ⨟ A) ⊕ (r ⨟ B) labelled choice
| CASE c {l ⇒ A | r ⇒ B} label observation

1. Only giving syntax for binary choice, but assume it for any finite arity
2. Use definitional equality to capture type-level computation

# Extending Session Types With Observing Processes

$$
\begin{aligned}
A, B &\coloneqq \cdots && \text{other session types} \\
&\mid (l \fatsemi A) \oplus (r \fatsemi B) && \text{labelled choice} \\
&\mid \text{CASE } c\ \{l \Rightarrow A \mid r \Rightarrow B\} && \text{label observation}
\end{aligned}
$$

**Operational Intuition**

CASE $c$ $\{l \Rightarrow A \mid r \Rightarrow B\} \equiv A$ if $l$ observed on channel $c$

CASE $c$ $\{l \Rightarrow A \mid r \Rightarrow B\} \equiv B$ if $r$ observed on channel $c$





Extending Session Types With Observing Processes
$A, B \coloneqq \cdots$ other session types
$\mid (l \fatsemi A) \oplus (r \fatsemi B)$ labelled choice
$\mid$ CASE $c$ $\{l \Rightarrow A \mid r \Rightarrow B\}$ label observation
Operational Intuition
CASE $c$ $\{l \Rightarrow A \mid r \Rightarrow B\} \equiv A$ if $l$ observed on channel $c$
CASE $c$ $\{l \Rightarrow A \mid r \Rightarrow B\} \equiv B$ if $r$ observed on channel $c$

1. Only giving syntax for binary choice, but assume it for any finite arity
2. Use definitional equality to capture type-level computation

# Revisiting Bit Flipping

Bit stream protocol:

$$\text{bits} = (\text{b0} \fatsemi \text{bits}) \oplus (\text{b1} \fatsemi \text{bits})$$





Revisiting Bit Flipping
Bit stream protocol:
bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\text{bits} = (\text{b0} \fatsemi \text{bits}) \oplus (\text{b1} \fatsemi \text{bits})$$

Bit flipping protocol, assuming $i : \text{bits}$:

$$\begin{aligned} \text{bitsFlip} = \text{CASE } i \ \{ & \text{b0} \Rightarrow (\text{b1} \fatsemi \text{bitsFlip}) \\ \mid\ & \text{b1} \Rightarrow (\text{b0} \fatsemi \text{bitsFlip}) \ \} \end{aligned}$$





Revisiting Bit Flipping
Bit stream protocol:
bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)
Bit flipping protocol, assuming i : bits:
bitsFlip = CASE i {b0 ⇒ (b1 ⨟ bitsFlip) | b1 ⇒ (b0 ⨟ bitsFlip) }

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\text{bits} = (\text{b0} ⨟ \text{bits}) \oplus (\text{b1} ⨟ \text{bits})$$

Bit flipping protocol, assuming $i : \text{bits}$:

$$\begin{aligned}\text{bitsFlip} = \text{CASE } i \; \{ & \text{b0} \Rightarrow (\text{b1} ⨟ \text{bitsFlip}) \\ \mid\ & \text{b1} \Rightarrow (\text{b0} ⨟ \text{bitsFlip}) \, \}\end{aligned}$$

```
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\text{bits} = (\text{b0} \fatsemi \text{bits}) \oplus (\text{b1} \fatsemi \text{bits})$$

Bit flipping protocol, assuming $i : \text{bits}$:

$$\begin{aligned} \text{bitsFlip} = \text{CASE } i \;\{ & \text{b0} \Rightarrow (\text{b1} \fatsemi \text{bitsFlip}) \\ \mid\; & \text{b1} \Rightarrow (\text{b0} \fatsemi \text{bitsFlip}) \;\} \end{aligned}$$

```
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





Revisiting Bit Flipping
Bit stream protocol:
bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)
Bit flipping protocol, assuming i : bits:
bitsFlip = CASE i {b0 ⇒ (b1 ⨟ bitsFlip)
| b1 ⇒ (b0 ⨟ bitsFlip) }
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
| b1 ⇒ send b0 on o; F(i; o) }

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\text{bits} = (\text{b0} \fatsemi \text{bits}) \oplus (\text{b1} \fatsemi \text{bits})$$

Bit flipping protocol, assuming $i : \text{bits}$:

$$\begin{aligned}\text{bitsFlip} = \text{CASE } i \; \{ & \text{b0} \Rightarrow (\text{b1} \fatsemi \text{bitsFlip}) \\ | \; & \text{b1} \Rightarrow (\text{b0} \fatsemi \text{bitsFlip}) \; \}\end{aligned}$$

```
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\mathrm{bits} = (\mathrm{b0} \fatsemi \mathrm{bits}) \oplus (\mathrm{b1} \fatsemi \mathrm{bits})$$

Bit flipping protocol, assuming $i : \mathrm{bits}$:

$$\begin{aligned} \mathrm{bitsFlip} = \mathrm{CASE}\ i\ \{ & \mathrm{b0} \Rightarrow (\mathrm{b1} \fatsemi \mathrm{bitsFlip}) \\ \mid\ & \mathrm{b1} \Rightarrow (\mathrm{b0} \fatsemi \mathrm{bitsFlip})\ \} \end{aligned}$$

```
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





Revisiting Bit Flipping
Bit stream protocol:
bits = (b0 ⨾ bits) ⊕ (b1 ⨾ bits)
Bit flipping protocol, assuming i : bits:
bitsFlip = CASE i {b0 ⇒ (b1 ⨾ bitsFlip) | b1 ⇒ (b0 ⨾ bitsFlip) }
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o) | b1 ⇒ send b0 on o; F(i; o) }

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\mathsf{bits} = (\mathsf{b0} \fatsemi \mathsf{bits}) \oplus (\mathsf{b1} \fatsemi \mathsf{bits})$$

Bit flipping protocol, assuming $i : \mathsf{bits}$:

$$\begin{aligned}\mathsf{bitsFlip} = \mathsf{CASE}\ i\ \{ & \mathsf{b0} \Rightarrow (\mathsf{b1} \fatsemi \mathsf{bitsFlip}) \\ \mid\ & \mathsf{b1} \Rightarrow (\mathsf{b0} \fatsemi \mathsf{bitsFlip})\ \}\end{aligned}$$

```
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





Revisiting Bit Flipping
Bit stream protocol:
bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)
Bit flipping protocol, assuming i : bits:
bitsFlip = CASE i {b0 ⇒ (b1 ⨟ bitsFlip)
| b1 ⇒ (b0 ⨟ bitsFlip) }
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
| b1 ⇒ send b0 on o; F(i; o) }

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\text{bits} = (\text{b0} \fatsemi \text{bits}) \oplus (\text{b1} \fatsemi \text{bits})$$

Bit flipping protocol, assuming $i : \text{bits}$:

$$\begin{aligned}\text{bitsFlip} = \text{CASE}\ i\ \{&\text{b0} \Rightarrow (\text{b1} \fatsemi \text{bitsFlip}) \\ |\ &\text{b1} \Rightarrow (\text{b0} \fatsemi \text{bitsFlip})\ \}\end{aligned}$$

```
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





Revisiting Bit Flipping

Bit stream protocol:

bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)

Bit flipping protocol, assuming i : bits:

bitsFlip = CASE i {b0 ⇒ (b1 ⨟ bitsFlip) | b1 ⇒ (b0 ⨟ bitsFlip) }

i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o) | b1 ⇒ send b0 on o; F(i; o) }

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\mathsf{bits} = (\mathsf{b0} \fatsemi \mathsf{bits}) \oplus (\mathsf{b1} \fatsemi \mathsf{bits})$$

Bit flipping protocol, assuming $i : \mathsf{bits}$:

$$\begin{aligned}\mathsf{bitsFlip} = \mathsf{CASE}\ i\ \{&\mathsf{b0} \Rightarrow (\mathsf{b1} \fatsemi \mathsf{bitsFlip}) \\ \mid\ &\mathsf{b1} \Rightarrow (\mathsf{b0} \fatsemi \mathsf{bitsFlip})\ \}\end{aligned}$$

```
i : bits ⊢ F :: o : (b1 ⨟ bitsFlip)
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





Revisiting Bit Flipping
Bit stream protocol:
bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)
Bit flipping protocol, assuming i : bits:
bitsFlip = CASE i {b0 ⇒ (b1 ⨟ bitsFlip)
| b1 ⇒ (b0 ⨟ bitsFlip) }
i : bits ⊢ F :: o : (b1 ⨟ bitsFlip)
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
| b1 ⇒ send b0 on o; F(i; o) }

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\text{bits} = (\text{b0} \fatsemi \text{bits}) \oplus (\text{b1} \fatsemi \text{bits})$$

Bit flipping protocol, assuming $i : \text{bits}$:

$$\begin{aligned} \text{bitsFlip} = \text{CASE } i\ \{ & \text{b0} \Rightarrow (\text{b1} \fatsemi \text{bitsFlip}) \\ \mid\ & \text{b1} \Rightarrow (\text{b0} \fatsemi \text{bitsFlip})\ \} \end{aligned}$$

```
i : bits ⊢ F :: o : (b1 ⨟ bitsFlip)
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





Revisiting Bit Flipping
Bit stream protocol:
bits = (b0 ⨟ bits) ⊕ (b1 ⨟ bits)
Bit flipping protocol, assuming i : bits:
bitsFlip = CASE i {b0 ⇒ (b1 ⨟ bitsFlip) | b1 ⇒ (b0 ⨟ bitsFlip) }
i : bits ⊢ F :: o : (b1 ⨟ bitsFlip)
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o) | b1 ⇒ send b0 on o; F(i; o) }

1. bitsflip uses unary labelled choice

# Revisiting Bit Flipping

Bit stream protocol:

$$\mathrm{bits} = (\mathrm{b0} \parr \mathrm{bits}) \oplus (\mathrm{b1} \parr \mathrm{bits})$$

Bit flipping protocol, assuming $i : \mathrm{bits}$:

$$\begin{aligned}\mathrm{bitsFlip} = \mathrm{CASE}\ i\ \{ & \mathrm{b0} \Rightarrow (\mathrm{b1} \parr \mathrm{bitsFlip}) \\ \mid\ & \mathrm{b1} \Rightarrow (\mathrm{b0} \parr \mathrm{bitsFlip})\ \}\end{aligned}$$

```
i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```





Revisiting Bit Flipping

Bit stream protocol:

bits = (b0 ⅋ bits) ⊕ (b1 ⅋ bits)

Bit flipping protocol, assuming i : bits:

bitsFlip = CASE i {b0 ⇒ (b1 ⅋ bitsFlip) | b1 ⇒ (b0 ⅋ bitsFlip) }

i : bits ⊢ F :: o : bitsFlip
F(i; o) = case i { b0 ⇒ send b1 on o; F(i; o) | b1 ⇒ send b0 on o; F(i; o) }

1. bitsflip uses unary labelled choice

# The Meaning of Specifications



The Meaning of Specifications

# What Do Process Specifications And Types Mean?

Typing judgments $I \vdash P :: O$ specify $P$'s communication behaviour.





What Do Process Specifications And Types Mean?
Typing judgments $I \vdash P :: O$ specify $P$'s communication behaviour.

1. Given inputs allowed by I, *P* may produce outputs allowed by O.
2. We have a good understanding of this when everything is static, but what happens with type-level computation?
3. Unsatisfying to just have syntax.
4. Need to first answer: what is the meaning of a type.
5. Classical session types are static and denote a set of allowed communications (OCS).
6. With CDST, think of types and specifications as programs whose executions generate all communications they allow.

# What Do Process Specifications And Types Mean?

Typing judgments $I \vdash P :: O$ specify $P$'s communication behaviour.

What does this specification mean when types involve computation?





What Do Process Specifications And Types Mean?
Typing judgments $I \vdash P :: O$ specify $P$'s communication behaviour.
What does this specification mean when types involve computation?

1. Given inputs allowed by I, $P$ may produce outputs allowed by O.
2. We have a good understanding of this when everything is static, but what happens with type-level computation?
3. Unsatisfying to just have syntax.
4. Need to first answer: what is the meaning of a type.
5. Classical session types are static and denote a set of allowed communications (OCS).
6. With CDST, think of types and specifications as programs whose executions generate all communications they allow.

# What Do Process Specifications And Types Mean?

Typing judgments $I \vdash P :: O$ specify $P$'s communication behaviour.

What does this specification mean when types involve computation?

What do types with computation even mean?





What Do Process Specifications And Types Mean?
Typing judgments $I \vdash P :: O$ specify $P$'s communication behaviour.
What does this specification mean when types involve computation?
What do types with computation even mean?

1. Given inputs allowed by I, $P$ may produce outputs allowed by O.
2. We have a good understanding of this when everything is static, but what happens with type-level computation?
3. Unsatisfying to just have syntax.
4. Need to first answer: what is the meaning of a type.
5. Classical session types are static and denote a set of allowed communications (OCS).
6. With CDST, think of types and specifications as programs whose executions generate all communications they allow.

# What Do Process Specifications And Types Mean?

Typing judgments $\mathrm{I} \vdash P :: \mathrm{O}$ specify *P*'s communication behaviour.

What does this specification mean when types involve computation?

What do types with computation even mean?

| **Classical Session Type** | **Channel-Dependent Type** |
| --- | --- |
| Set of allowed communications | Program that computes allowed communications |





What Do Process Specifications And Types Mean?
Typing judgments $\mathrm{I} \vdash P :: \mathrm{O}$ specify *P*'s communication behaviour.
What does this specification mean when types involve computation?
What do types with computation even mean?
Classical Session Type Set of allowed communications
Channel-Dependent Type Program that computes allowed communications

1. Given inputs allowed by I, *P* may produce outputs allowed by O.
2. We have a good understanding of this when everything is static, but what happens with type-level computation?
3. Unsatisfying to just have syntax.
4. Need to first answer: what is the meaning of a type.
5. Classical session types are static and denote a set of allowed communications (OCS).
6. With CDST, think of types and specifications as programs whose executions generate all communications they allow.

# Session Types Are Non-Deterministic Processes

## Core Ideas

1. A session type is a non-deterministic process that asynchronously broadcasts communications.
2. The communications it allows are those it can broadcast.





Session Types Are Non-Deterministic Processes
Core Ideas
1. A session type is a non-deterministic process that asynchronously broadcasts communications.
2. The communications it allows are those it can broadcast.

1. Make the idea that types/specs are programs that compute allowed communications a bit more explicit.
2. Interpret types as processes that observe and generate communications
3. Broadcast means processes send messages to everybody
4. Asynchronously means that processes don’t need to synchronize to send: just send, and others will receive when they are ready

# Session Types Are Non-Deterministic Processes

**Core Ideas**

1. A session type is a non-deterministic process that asynchronously broadcasts communications.
2. The communications it allows are those it can broadcast.

A specification $I \vdash P :: O$ means that $P$ must

1. accept all communications I can broadcast
2. send only communications O can broadcast given those so far broadcast by I, O





Session Types Are Non-Deterministic Processes
Core Ideas
1. A session type is a non-deterministic process that asynchronously broadcasts communications.
2. The communications it allows are those it can broadcast.
A specification I ⊢ P :: O means that P must
1. accept all communications I can broadcast
2. send only communications O can broadcast given those so far broadcast by I, O

1. Make the idea that types/specs are programs that compute allowed communications a bit more explicit.
2. Interpret types as processes that observe and generate communications
3. Broadcast means processes send messages to everybody
4. Asynchronously means that processes don't need to synchronize to send: just send, and others will receive when they are ready

# Design Choices and Challenges



Design Choices and Challenges

# Dependency Condition

Type-level dependency can only restrict output.
It never restricts input.





Dependency Condition
Type-level dependency can only restrict output.
It never restricts input.

1. Makes no sense to use local information to restrict what you receive.
2. Postel's Law: conservative in what you send, liberal in what you accept.
3. This condition has important ramifications on process composition.

## Dependency Condition

Type-level dependency can only restrict output.
It never restricts input.

**Example (Permitted)**

$$i : (l \mathbin{⨟} A) \oplus (r \mathbin{⨟} B), j : \ldots \vdash P :: o : \text{CASE}\ i\ \{l \Rightarrow \ldots \mid r \Rightarrow \ldots\}$$



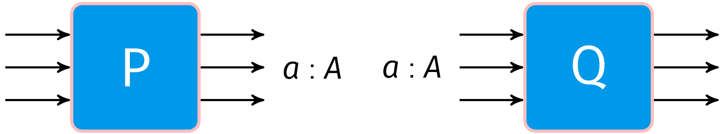


Dependency Condition
Type-level dependency can only restrict output.
It never restricts input.
Example (Permitted)
$i : (l \mathbin{⨟} A) \oplus (r \mathbin{⨟} B), j : \ldots \vdash P :: o : \text{CASE}\ i\ \{l \Rightarrow \ldots \mid r \Rightarrow \ldots\}$

1. Makes no sense to use local information to restrict what you receive.
2. Postel’s Law: conservative in what you send, liberal in what you accept.
3. This condition has important ramifications on process composition.

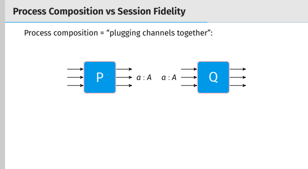
# Process Composition vs Session Fidelity

Process composition = “plugging channels together”:





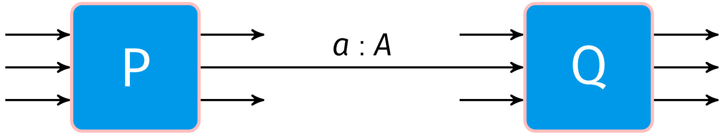


1. Explain process composition: parallel composition, and then hide channel from external view.
2. Session fidelity is the property that a process is never sent a communication it cannot handle.
3. Often ensured in part by only composing channels of equal / dual type.
4. Requiring channels of equal type means that we cannot compose channels if one has dependency.
5. If we cannot compose processes, then what’s the point?
6. Want to be able to determine this statically.

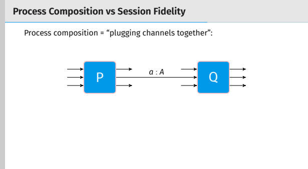
# Process Composition vs Session Fidelity

Process composition = “plugging channels together”:





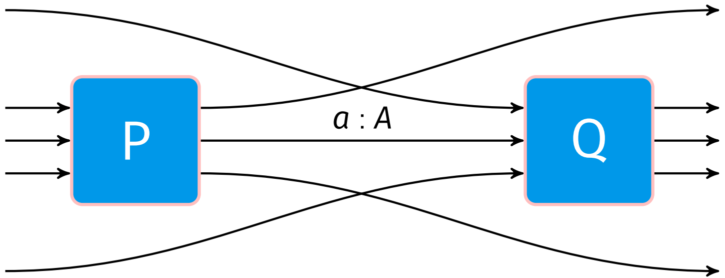



1. Explain process composition: parallel composition, and then hide channel from external view.
2. Session fidelity is the property that a process is never sent a communication it cannot handle.
3. Often ensured in part by only composing channels of equal / dual type.
4. Requiring channels of equal type means that we cannot compose channels if one has dependency.
5. If we cannot compose processes, then what’s the point?
6. Want to be able to determine this statically.

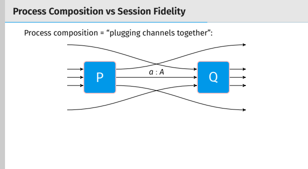
# Process Composition vs Session Fidelity

Process composition = “plugging channels together”:





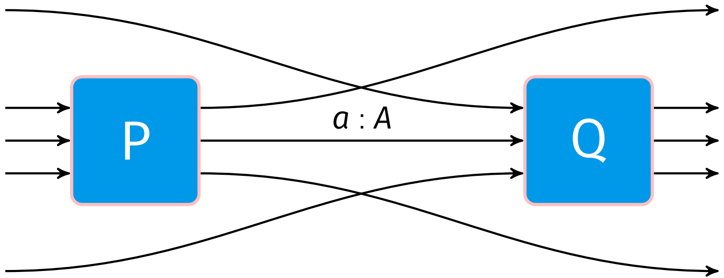



1. Explain process composition: parallel composition, and then hide channel from external view.
2. Session fidelity is the property that a process is never sent a communication it cannot handle.
3. Often ensured in part by only composing channels of equal / dual type.
4. Requiring channels of equal type means that we cannot compose channels if one has dependency.
5. If we cannot compose processes, then what’s the point?
6. Want to be able to determine this statically.

# Process Composition vs Session Fidelity

Process composition = “plugging channels together”:

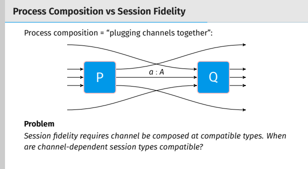



**Problem**

*Session fidelity requires channel be composed at compatible types. When are channel-dependent session types compatible?*








1. Explain process composition: parallel composition, and then hide channel from external view.
2. Session fidelity is the property that a process is never sent a communication it cannot handle.
3. Often ensured in part by only composing channels of equal / dual type.
4. Requiring channels of equal type means that we cannot compose channels if one has dependency.
5. If we cannot compose processes, then what’s the point?
6. Want to be able to determine this statically.

# Session Sorting

Type-level dependency restricts what processes send.

**Definition (Session Sorting)**

Write $A <: S$ when the type $A$ is a restriction of the (non-dependent) session type $S$.





Session Sorting
Type-level dependency restricts what processes send.
Definition (Session Sorting)
Write $A <: S$ when the type $A$ is a restriction of the (non-dependent) session type $S$.

1. Session sorting is an abstraction akin to subtyping or dependency erasure
2. Those familiar with session subtyping: sorting is basically an extension of Gay and Hole style subtyping to handle case constructs.
3. Provides a dependency free upper bound on what $A$ allows.
4. Call the non-dependent $S$ the sort of $A$.
5. Can compose channel of type $A$ with one of type $S$ when $A : S$.
6. Inspiration for treatment of composition from Griffith's thesis.

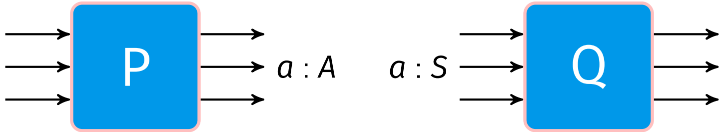
# Session Sorting

Type-level dependency restricts what processes send.

**Definition (Session Sorting)**

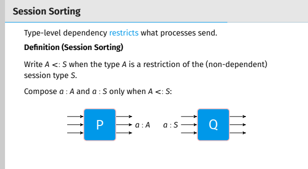
Write $A <: S$ when the type $A$ is a restriction of the (non-dependent) session type $S$.

Compose $a : A$ and $a : S$ only when $A <: S$:







1. Session sorting is an abstraction akin to subtyping or dependency erasure
2. Those familiar with session subtyping: sorting is basically an extension of Gay and Hole style subtyping to handle case constructs.
3. Provides a dependency free upper bound on what $A$ allows.
4. Call the non-dependent $S$ the sort of $A$.
5. Can compose channel of type $A$ with one of type $S$ when $A : S$.
6. Inspiration for treatment of composition from Griffith's thesis.

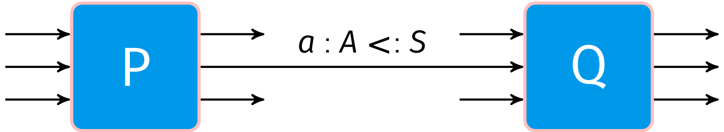
# Session Sorting

Type-level dependency restricts what processes send.

**Definition (Session Sorting)**

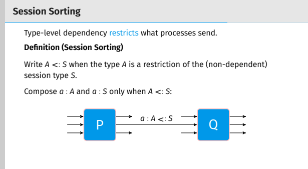
Write $A <: S$ when the type $A$ is a restriction of the (non-dependent) session type $S$.

Compose $a : A$ and $a : S$ only when $A <: S$:







1. Session sorting is an abstraction akin to subtyping or dependency erasure
2. Those familiar with session subtyping: sorting is basically an extension of Gay and Hole style subtyping to handle case constructs.
3. Provides a dependency free upper bound on what $A$ allows.
4. Call the non-dependent $S$ the sort of $A$.
5. Can compose channel of type $A$ with one of type $S$ when $A : S$.
6. Inspiration for treatment of composition from Griffith's thesis.

# Process Composition with Session Sorting

Recall the bit stream and bit flipping protocols:

$$\begin{aligned} \mathsf{bits} &= (\mathsf{b0} \fatsemi \mathsf{bits}) \oplus (\mathsf{b1} \fatsemi \mathsf{bits}) \\ \mathsf{bitsFlip} &= \mathsf{CASE}\ i\ \{\mathsf{b0} \Rightarrow (\mathsf{b1} \fatsemi \mathsf{bitsFlip}) \\ &\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow (\mathsf{b0} \fatsemi \mathsf{bitsFlip})\ \} \end{aligned}$$





Process Composition with Session Sorting
Recall the bit stream and bit flipping protocols:
bits = (b0 ⨾ bits) ⊕ (b1 ⨾ bits)
bitsFlip = CASE i {b0 ⇒ (b1 ⨾ bitsFlip)
| b1 ⇒ (b0 ⨾ bitsFlip) }

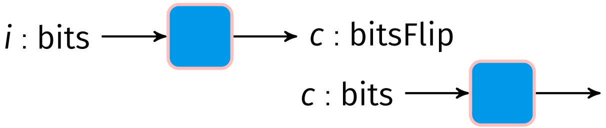
1. Sorting for cases analogous to typing for case statements in functional languages.
2. bitsFlip is of sort bits because each branch is of sort bits: it is a restriction of the sort bits
3. Lots of other cool design challenges with composition that I'd be happy to talk about offline.

# Process Composition with Session Sorting

Recall the bit stream and bit flipping protocols:

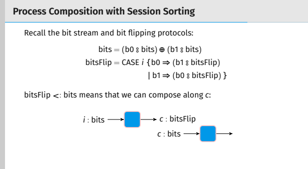
$$\begin{aligned} \mathsf{bits} &= (\mathsf{b0} \fatsemi \mathsf{bits}) \oplus (\mathsf{b1} \fatsemi \mathsf{bits}) \\ \mathsf{bitsFlip} &= \mathsf{CASE}\ i\ \{\mathsf{b0} \Rightarrow (\mathsf{b1} \fatsemi \mathsf{bitsFlip}) \\ &\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow (\mathsf{b0} \fatsemi \mathsf{bitsFlip})\ \} \end{aligned}$$

$\mathsf{bitsFlip} <: \mathsf{bits}$ means that we can compose along $c$:







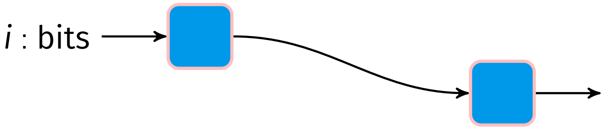
1. Sorting for cases analogous to typing for case statements in functional languages.
2. bitsFlip is of sort bits because each branch is of sort bits: it is a restriction of the sort bits
3. Lots of other cool design challenges with composition that I'd be happy to talk about offline.

# Process Composition with Session Sorting

Recall the bit stream and bit flipping protocols:

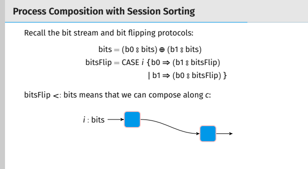
$$
\begin{aligned}
\mathsf{bits} &= (\mathsf{b0} \fatsemi \mathsf{bits}) \oplus (\mathsf{b1} \fatsemi \mathsf{bits}) \\
\mathsf{bitsFlip} &= \mathsf{CASE}\ i\ \{\mathsf{b0} \Rightarrow (\mathsf{b1} \fatsemi \mathsf{bitsFlip}) \\
&\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow (\mathsf{b0} \fatsemi \mathsf{bitsFlip})\ \}
\end{aligned}
$$

$\mathsf{bitsFlip} <: \mathsf{bits}$ means that we can compose along $c$:








1. Sorting for cases analogous to typing for case statements in functional languages.
2. bitsFlip is of sort bits because each branch is of sort bits: it is a restriction of the sort bits
3. Lots of other cool design challenges with composition that I'd be happy to talk about offline.

# Related Work

- *Value-dependent session types*: session types depend on transmitted values.





Related Work

- *Value-dependent session types*: session types depend on transmitted values.

1. VDST: depend on values from same channel. Invariants captured by sending proof terms. T/C/P 2011
2. LDST: TV19. Treat labels as first class objects. Types do a case analysis on labels sent on same channel.
3. LDST: Original motivation was to disentangle communication from introducing and eliminating values.
4. VDST/LDST: dependency only on same channel.
5. MPST: can globally specify interactions. Very rich but very complex. Typically closed world, hard to extend with new processes.
6. Stolze, Miculan, Di Gianantonio 2021 worked on extending MPST with new process composition.

- *Value-dependent session types*: session types depend on transmitted values.
- *Label-dependent session types*: session types depend on transmitted labels.





Related Work

- *Value-dependent session types*: session types depend on transmitted values.
- *Label-dependent session types*: session types depend on transmitted labels.

1. VDST: depend on values from same channel. Invariants captured by sending proof terms. T/C/P 2011
2. LDST: TV19. Treat labels as first class objects. Types do a case analysis on labels sent on same channel.
3. LDST: Original motivation was to disentangle communication from introducing and eliminating values.
4. VDST/LDST: dependency only on same channel.
5. MPST: can globally specify interactions. Very rich but very complex. Typically closed world, hard to extend with new processes.
6. Stolze, Miculan, Di Gianantonio 2021 worked on extending MPST with new process composition.

## Related Work

- *Value-dependent session types*: session types depend on transmitted values.
- *Label-dependent session types*: session types depend on transmitted labels.
- *Multi-party session types* provide a rich notion of process specification, but are quite complex.





Related Work

- *Value-dependent session types*: session types depend on transmitted values.
- *Label-dependent session types*: session types depend on transmitted labels.
- *Multi-party session types* provide a rich notion of process specification, but are quite complex.

1. VDST: depend on values from same channel. Invariants captured by sending proof terms. T/C/P 2011
2. LDST: TV19. Treat labels as first class objects. Types do a case analysis on labels sent on same channel.
3. LDST: Original motivation was to disentangle communication from introducing and eliminating values.
4. VDST/LDST: dependency only on same channel.
5. MPST: can globally specify interactions. Very rich but very complex. Typically closed world, hard to extend with new processes.
6. Stolze, Miculan, Di Gianantonio 2021 worked on extending MPST with new process composition.

# Future Work

- Adapt the type system to guarantee deadlock freedom
- Integrate with other forms of dependency like value- and label-dependency
- Find a logical interpretation
- Prove subject reduction
- Implement channel-dependent session types!





Future Work

- Adapt the type system to guarantee deadlock freedom
- Integrate with other forms of dependency like value- and label-dependency
- Find a logical interpretation
- Prove subject reduction
- Implement channel-dependent session types!

# Thank You

**Take away**
Channel-dependent session types use restricted type-level concurrent computation to capture more precise communication invariants.

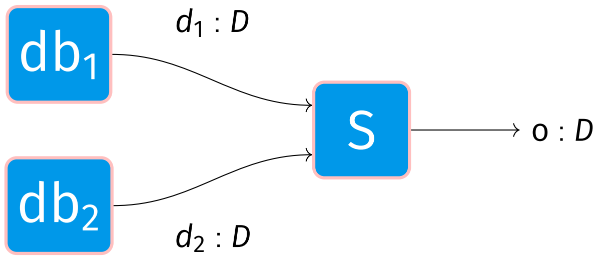




Thank You

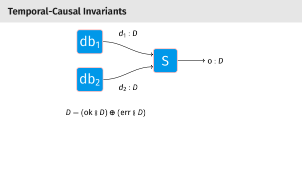
Take away
Channel-dependent session types use restricted type-level concurrent computation to capture more precise communication invariants.

# Temporal-Causal Invariants



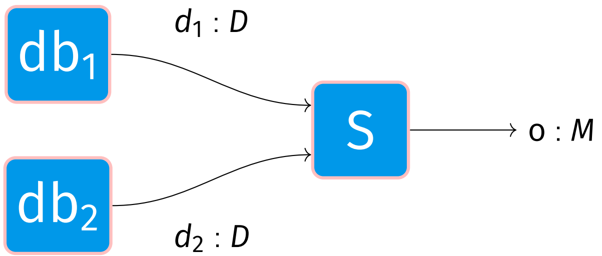

$$D = (\mathsf{ok} \mathbin{⅋} D) \oplus (\mathsf{err} \mathbin{⅋} D)$$





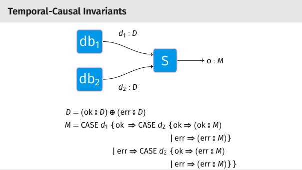
1. Backup slide
2. The bit flipping example captures information flow.
3. We can also use type-level computation to describe temporal and causal invariants.
4. Want to observe ok on $o$ only if both databases successfully committed their data.
5. Particularly useful in bidirectional settings where we can delegate communication: lets us specify how our delegates communicate.

# Temporal-Causal Invariants




$$
\begin{aligned}
D &= (\mathsf{ok} \fatsemi D) \oplus (\mathsf{err} \fatsemi D) \\
M &= \mathsf{CASE}\ d_1\ \{\mathsf{ok} \Rightarrow \mathsf{CASE}\ d_2\ \{\mathsf{ok} \Rightarrow (\mathsf{ok} \fatsemi M) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{err} \Rightarrow (\mathsf{err} \fatsemi M)\} \\
&\qquad\qquad\quad \mid \mathsf{err} \Rightarrow \mathsf{CASE}\ d_2\ \{\mathsf{ok} \Rightarrow (\mathsf{err} \fatsemi M) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{err} \Rightarrow (\mathsf{err} \fatsemi M)\}\}
\end{aligned}
$$





1. Backup slide
2. The bit flipping example captures information flow.
3. We can also use type-level computation to describe temporal and causal invariants.
4. Want to observe ok on *o* only if both databases successfully committed their data.
5. Particularly useful in bidirectional settings where we can delegate communication: lets us specify how our delegates communicate.