Message-Observing Sessions

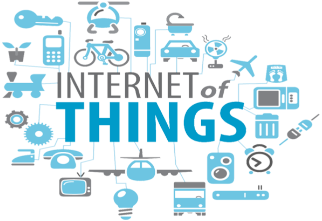
Ryan Kavanagh and Brigitte Pientka

OOPSLA 2024

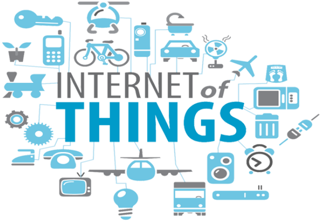
Université du Québec à Montréal and McGill University

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

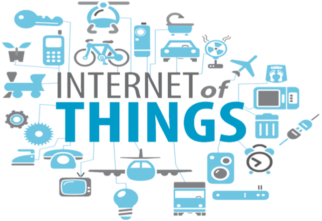
Communicating systems are ubiquitous



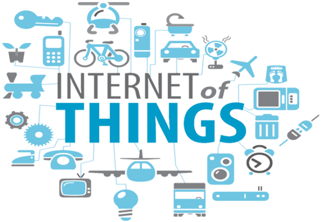
Communicating systems are ubiquitous



Communicating systems are ubiquitous



Communicating systems are ubiquitous

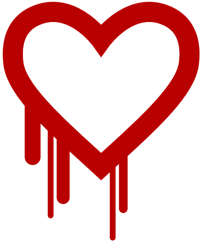


Miscommunication is costly

- Incorrectly implemented communication protocols are costly

Miscommunication is costly

- Incorrectly implemented communication protocols are costly



Miscommunication is costly

- Incorrectly implemented communication protocols are costly

aws

Miscommunication is costly

- Incorrectly implemented communication protocols are costly

aws yahoo!

Miscommunication is costly

- Incorrectly implemented communication protocols are costly

Miscommunication is costly

- Incorrectly implemented communication protocols are costly

aws

yahoo!

Canada Revenue
Agency

Agence du revenu
du Canada

Miscommunication is costly

- Incorrectly implemented communication protocols are costly

Canada Revenue
Agency

Agence du revenu
du Canada

- Want to statically guarantee our programs communicate correctly

Miscommunication is costly

- Incorrectly implemented communication protocols are costly

aws yahoo!

Canada Revenue Agency Agence du revenu du Canada

- Want to statically guarantee our programs communicate correctly
- Want to precisely specify the desired communication behaviours

Key technology: Session types

Session types encode communication protocols as types.

Key technology: Session types

Session types encode communication protocols as types.

- Multiparty session types (MPST) specify systems top-down
  - \+ Specify rich interactions involving multiple processes
  - − Typically not compositional, a closed-world approach

Key technology: Session types

Session types encode communication protocols as types.

- Multiparty session types (MPST) specify systems top-down
  - \+ Specify rich interactions involving multiple processes
  - − Typically not compositional, a closed-world approach
- Binary session types specify systems bottom-up
  - − Only specify local interactions between pairs of processes
  - \+ Compositional, with many expressive varieties

Key technology: Session types

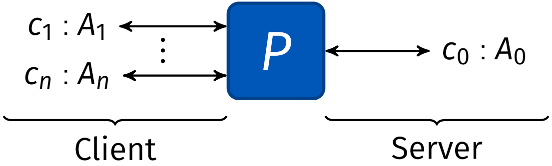
Session types encode communication protocols as types.

- Multiparty session types (MPST) specify systems top-down
  - \+ Specify rich interactions involving multiple processes
  - − Typically not compositional, a closed-world approach
- Binary session types specify systems bottom-up
  - − Only specify local interactions between pairs of processes
  - \+ Compositional, with many expressive varieties

Research Problem

Can we have the best of both worlds:
the ability to specify global properties *and* compositionality?

Specifying communication-based concurrency



$c_1 : A_1$

$\vdots$

$c_n : A_n$

P

$c_0 : A_0$

Client

Server

where

- P — process
- c_i — name of bidirectional communication channel
- A_i — protocol (session type) on channel c_i

Write $\mathrm{P}\,[\mathrm{c}_1 : \mathrm{A}_1, \ldots, \mathrm{c}_n : \mathrm{A}_n]\,(\mathrm{c}_0 : \mathrm{A}_0)$ to syntactically specify P

Bit streams

Bit stream protocol:

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

Example communications satisfying `bits`:

$$\xrightarrow{\quad \mathsf{b0}\,\mathsf{b1}\,\mathsf{b0}\,\mathsf{b0} \cdots \quad} \mathsf{c} : \mathsf{bits}$$

Specifying bit flipping

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

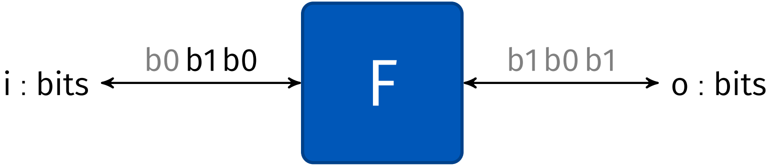
Specifying bit flipping

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

Specifying bit flipping

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

Specifying bit flipping

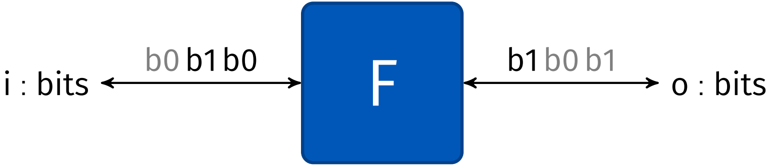


```
F [i : bits] (o : bits) =
          case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```

Specifying bit flipping

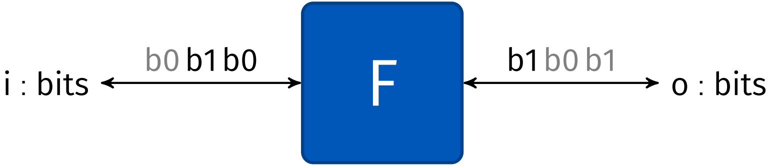
```
F [i : bits] (o : bits) =
          case i { b0 ⇒ send b1 on o; F(i; o)
                 | b1 ⇒ send b0 on o; F(i; o) }
```

Specifying bit flipping



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

Specifying bit flipping



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

Specifying bit flipping

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

Problem
The specification `F [i : bits] (o : bits)` does not specify or enforce bit flipping!

Today’s contribution: **Most**

Most is a language for compositionally specifying communication protocols that depend on communications occurring on other channels.

Today's contribution: **Most**

Most is a language for compositionally specifying communication protocols that depend on communications occurring on other channels.

Today's contribution: Most

Most is a language for compositionally specifying communication protocols that depend on communications occurring on other channels.

Today's contribution: Most

Most is a language for compositionally specifying communication protocols that depend on communications occurring on other channels.

Three ingredients:

1. syntax for specifying protocols with type-level computation
2. semantic framework for explaining protocols as sequences of allowed communications while taking dependency into account
3. static typechecking to ensure that processes satisfy their protocols

Most’s Syntax

Introducing message-observing session types

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

Introducing message-observing session types

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

Introducing message-observing session types

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

Operational Intuition

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

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

Introducing message-observing session types

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

Operational Intuition

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

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

See paper for how to observe termination and channel transmission!

Revisiting bit flipping

Bit stream protocol:

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

Revisiting bit flipping

Bit stream protocol:

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

Bit flipping protocol relative to a channel $\mathtt{i}$: bits:

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

Revisiting bit flipping

Bit stream protocol:

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

Bit flipping protocol relative to a channel $\mathtt{i} : \mathsf{bits}$:

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

Revisiting bit flipping

Bit stream protocol:

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

Bit flipping protocol relative to a channel $\mathtt{i} : \mathsf{bits}$:

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

Revisiting bit flipping

Bit stream protocol:

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

Bit flipping protocol relative to a channel `i` : bits:

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

Revisiting bit flipping

Bit stream protocol:

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

Bit flipping protocol relative to a channel $\mathtt{i} : \mathsf{bits}$:

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

Revisiting bit flipping

Bit stream protocol:

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

Bit flipping protocol relative to a channel `i` : bits:

$$\begin{aligned} \mathsf{bitsFlip}(i) = \mathsf{CASE}\ \mathtt{i}\ \{ & \mathsf{b0} \Rightarrow (\mathsf{b1} \parr \mathsf{bitsFlip}(\mathsf{i})) \\ | \ & \mathsf{b1} \Rightarrow (\mathsf{b0} \parr \mathsf{bitsFlip}(\mathsf{i}))\ \} \end{aligned}$$

```
F [i : bits] (o : bitsFlip(i)) = ···
```

Revisiting bit flipping

Bit stream protocol:

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

Bit flipping protocol relative to a channel `i` : bits:

$$\begin{aligned}\mathsf{bitsFlip}(i) = \mathsf{CASE}\ \mathtt{i}\ \{ & \mathsf{b0} \Rightarrow (\mathsf{b1} \mathbin{⨟} \mathsf{bitsFlip}(\mathsf{i})) \\ | \ & \mathsf{b1} \Rightarrow (\mathsf{b0} \mathbin{⨟} \mathsf{bitsFlip}(\mathsf{i}))\ \}\end{aligned}$$

```
F [i : bits] (o : bitsFlip(i)) = ···
```

This technique can specify any stream transducer or multiplexer!

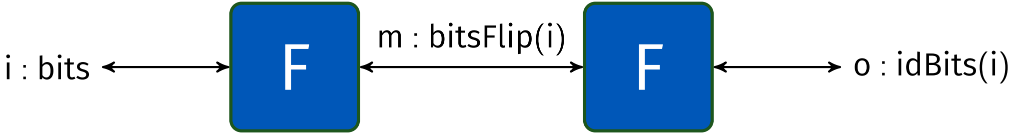
Insight #1: Type-level concurrent computation

First Key Insight

We can specify dependent communication protocols with a restricted form of type-level concurrent computation.

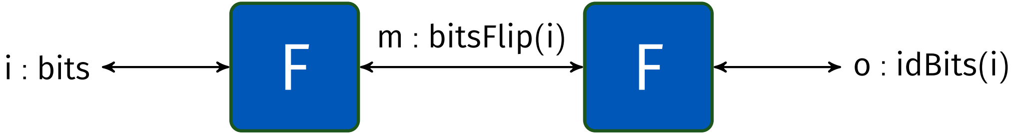
Composing processes

We want to be able to specify process compositions:



Composing processes

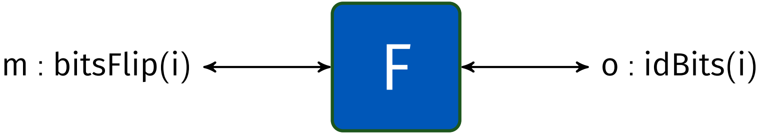
We want to be able to specify process compositions:



We want to reason about the whole in terms of the specifications of the parts.

Insight #2: Tracking ambient channels to achieve compositionality

The specification of the right process depends on an ambient i : bits:



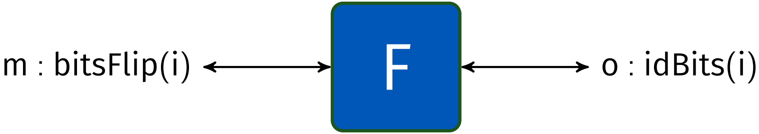
m : bitsFlip(i)

F

o : idBits(i)

Insight #2: Tracking ambient channels to achieve compositionality

The specification of the right process depends on an ambient i : bits:



We extend our specifications to track assumptions about ambient channels:

```
F {i : bits} [m : bitsFlip(i)] (o : idBits(i)) = ...
```

Rely-guarantee perspective ensures compositionality!

Most’s Semantics

Denotational semantics

A process denotes a set of traces of messages sent or received on channels. For example, flipping bits received on *m* onto a channel *o*:

$$\begin{aligned}[\![\mathtt{F}(\mathtt{m};\mathtt{o})]\!] = \{ & \text{recv b0 on m} :: \text{send b1 on o} :: \cdots, \\ & \text{recv b1 on m} :: \text{send b0 on o} :: \cdots, \ldots \}\end{aligned}$$

Denotational semantics

A process denotes a set of traces of messages sent or received on channels. For example, flipping bits received on *m* onto a channel *o*:

$$\begin{aligned}[\![\mathtt{F}(\mathtt{m};\mathtt{o})]\!] = \{&\mathsf{recv}\ \mathsf{b0}\ \mathsf{on}\ \mathsf{m} :: \mathsf{send}\ \mathsf{b1}\ \mathsf{on}\ \mathsf{o} :: \cdots,\\ &\mathsf{recv}\ \mathsf{b1}\ \mathsf{on}\ \mathsf{m} :: \mathsf{send}\ \mathsf{b0}\ \mathsf{on}\ \mathsf{o} :: \cdots, \ldots\}\end{aligned}$$

A specification denotes a set of allowed traces, interleaved with constraints on ambient channels. For example,

$$\begin{aligned}&[\![\{\mathsf{i} : \mathsf{bits}\}\ [\mathsf{m} : \mathsf{bitsFlip}(\mathsf{i})]\ (\mathsf{o} : \mathsf{idBits}(\mathsf{i}))]\!]\\ &= \{\ \mathsf{rely}\ \mathsf{b0}\ \mathsf{on}\ \mathsf{i} :: \mathsf{recv}\ \mathsf{b1}\ \mathsf{on}\ \mathsf{m} :: \mathsf{send}\ \mathsf{b0}\ \mathsf{on}\ \mathsf{o} :: \cdots, \ldots\ \}\end{aligned}$$

Typechecking Most

Constraint generation and checking

Typechecking P against a specification generates constraints $\mathcal{T}$ on the what communications may appear on ambient channels:

$$P \Vdash \{\mathit{AmbientCtx}\}\ [\mathit{ClientCtx}]\ (a : A)\ /\!/\ \mathcal{T}$$

Constraint generation and checking

Typechecking P against a specification generates constraints $\mathcal{T}$ on the what communications may appear on ambient channels:

$$P \Vdash \{AmbientCtx\}\ [ClientCtx]\ (a : A)\ /\!/\ \mathcal{T}$$

When typechecking process compositions, we check that each process satisfies the constraints that the other imposes on its channels.

Soundness

Theorem

Our typechecking algorithm is semantically sound:
If P typechecks against a specification, then its traces are among those allowed by that specification.

Related Work

Not the first dependent session type system, but crucially different:

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

Multi-party session types provide a rich notion of process specification, but are quite complex and not compositional.

Future Work

- Introduce sharing to specify and verify shared services (databases, shared data structures, etc.)
- Develop an elegant subtyping relation to allow composition along channels with different types
- Mechanize the system and extract a verified compiler
- Characterize the expressiveness gap between Most and MPST

Takeaways

Concurrent type-level computation lets us compositionally specify protocols that vary based on ambient communications.

Most provides a significant step towards capturing message-dependent protocols and providing more precise specifications.

I am recruiting students! If this work sounds interesting, please come talk to me!