Substructural Observed Communication Semantics

Ryan Kavanagh

EXPRESS/SOS 2020

Carnegie Mellon University

Contributions

An **observed communication semantics** for session-typed languages **with recursion** that are specified by **substructural operational semantics** (multiset rewriting systems).

Contributions

An **observed communication semantics** for session-typed languages **with recursion** that are specified by **substructural operational semantics** (multiset rewriting systems).

A notion of **fairness for multiset rewriting systems**.

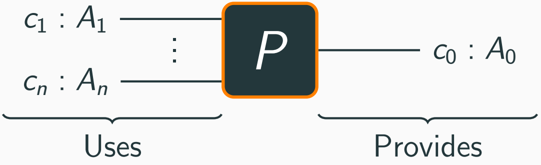
- Sufficient conditions for a fair scheduler
- Associated reasoning principles
- Various properties of fair traces

Communicating Processes

Where

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

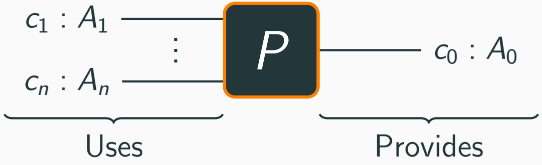
Communicating Processes



Where

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

Communicating Processes



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

What Can We Observe?

Key principle: We can only interact with processes through communication.

Corollary: Communications are the only semantically meaningful observables.

Observed Communication Semantics, Informally

Observed Communication Semantics, Informally

The **observation** of a process $c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ is the $(n+1)$-tuple

$$(\!| c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 |\!) = (c_0 : v_0, \ldots, c_n : v_n)$$

where $v_0, \ldots, v_n$ are the communications observed on the free channels $c_0, \ldots, c_n$.

Observed Communication Semantics, Informally

The **observation** of a process $c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ is the $(n+1)$-tuple

$$(\!| c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 |\!) = (c_0 : v_0, \ldots, c_n : v_n)$$

where $v_0, \ldots, v_n$ are the communications observed on the free channels $c_0, \ldots, c_n$.

Two processes $\Delta \vdash P :: a : A$ and $\Delta \vdash Q :: a : A$ are **observationally congruent** if for all $\Delta' \vdash C[\cdot] :: b : B$,

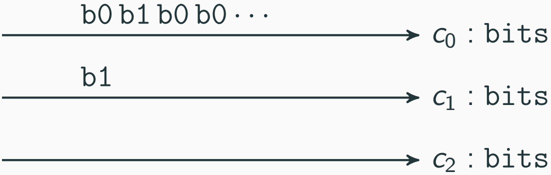
$$(\!|\Delta' \vdash C[P] :: b : B|\!) = (\!|\Delta' \vdash C[Q] :: b : B|\!).$$

Bit Stream Protocol

Bit stream protocol:

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

Example communications satisfying `bits`:

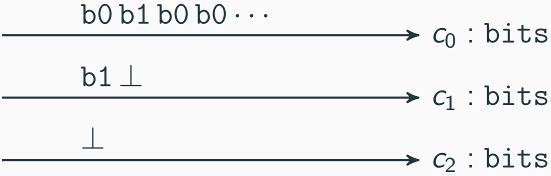


Bit Stream Protocol

Bit stream protocol:

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

Example communications satisfying `bits`:



Sending A Bit Stream

⊢ fix S in i.b1; i.b0; S :: i : bits

Sending A Bit Stream

$\vdash$ `fix S in i.b1; i.b0; S :: i : bits`

Sending A Bit Stream

⊢ fix S in i.b1; i.b0; S :: i : bits

Sending A Bit Stream

$\vdash$ `fix S in i.b1; i.b0; S :: i : bits`

Sending A Bit Stream

$\vdash$ `fix S in i.b1; i.b0; S :: i : bits`

Sending A Bit Stream

$\vdash$ `fix S in i.b1; i.b0; S :: i : bits`

Observation:

$$(\!|\vdash \mathtt{S} :: \mathtt{i} : \mathtt{bits}|\!) = (\mathtt{i} : (\mathtt{b1}, (\mathtt{b0}, (\mathtt{b1}, (\cdots)))))$$

Flipping Bits


```
i : bits ⊢ fix F in case i
                  { b0 => o.b1; F
                  | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                    { b0 => o.b1; F
                    | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                 { b0 => o.b1; F
                 | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                 { b0 => o.b1; F
                 | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                { b0 => o.b1; F
                | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                 { b0 => o.b1; F
                 | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                  { b0 => o.b1; F
                  | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                    { b0 => o.b1; F
                    | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                 { b0 => o.b1; F
                 | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                { b0 => o.b1; F
                | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                   { b0 => o.b1; F
                   | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                    { b0 => o.b1; F
                    | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                 { b0 => o.b1; F
                 | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                 { b0 => o.b1; F
                 | b1 => o.b0; F } :: o : bits
```

Flipping Bits


```
i : bits ⊢ fix F in case i
                   { b0 => o.b1; F
                   | b1 => o.b0; F } :: o : bits
```

Observation:

$$(\!|\, \texttt{i} : \texttt{bits} \vdash \texttt{F} :: \texttt{o} : \texttt{bits} \,|\!) = (\texttt{i} : \bot_{\texttt{bits}}, \texttt{o} : \bot_{\texttt{bits}})$$

Composing Processes

We can compose processes

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

$$\texttt{i} : \texttt{bits} \vdash \texttt{F} :: \texttt{o} : \texttt{bits}$$

to get the process $\vdash \texttt{i} : \texttt{bits} \leftarrow \texttt{S}; \texttt{F} :: \texttt{o} : \texttt{bits}$

Composing Processes

We can compose processes

$$\vdash \mathtt{S} :: \mathtt{i} : \mathtt{bits}$$
$$\mathtt{i} : \mathtt{bits} \vdash \mathtt{F} :: \mathtt{o} : \mathtt{bits}$$

to get the process $\vdash \mathtt{i} : \mathtt{bits} \leftarrow \mathtt{S}; \mathtt{F} :: \mathtt{o} : \mathtt{bits}$

Composing Processes

We can compose processes

$$\begin{array}{rl} & \vdash \texttt{S} :: \texttt{i} : \texttt{bits} \\ \texttt{i} : \texttt{bits} & \vdash \texttt{F} :: \texttt{o} : \texttt{bits} \end{array}$$

to get the process $\vdash \texttt{i} : \texttt{bits} \leftarrow \texttt{S}; \texttt{F} :: \texttt{o} : \texttt{bits}$

Observation:

$$(\!|\vdash \texttt{i} : \texttt{bits} \leftarrow \texttt{S}; \texttt{F} :: \texttt{o} : \texttt{bits}|\!) = (\texttt{o} : (\texttt{b1}, (\texttt{b0}, (\texttt{b1}, (\cdots)))))$$

Buffering and Equivalence

Consider an alternate implementation BuffF of F that buffers input and processes bits two at a time.

$$
\begin{aligned}
&(\!| \vdash \mathtt{i} : \mathtt{bits} \leftarrow \mathtt{S}; \mathtt{BuffF} :: \mathtt{o} : \mathtt{bits} |\!) \\
&= (\mathtt{o} : (\mathtt{b1}, (\mathtt{b0}, (\mathtt{b1}, (\cdots))))) \\
&= (\!| \vdash \mathtt{i} : \mathtt{bits} \leftarrow \mathtt{S}; \mathtt{F} :: \mathtt{o} : \mathtt{bits} |\!)
\end{aligned}
$$

Buffering and Equivalence

Consider an alternate implementation `BuffF` of `F` that buffers input and processes bits two at a time.

$$\begin{aligned}
&(\!|\vdash \mathtt{i} : \mathtt{bits} \leftarrow \mathtt{S}; \mathtt{BuffF} :: \mathtt{o} : \mathtt{bits}|\!) \\
&= (\mathtt{o} : (\mathtt{b1}, (\mathtt{b0}, (\mathtt{b1}, (\cdots))))) \\
&= (\!|\vdash \mathtt{i} : \mathtt{bits} \leftarrow \mathtt{S}; \mathtt{F} :: \mathtt{o} : \mathtt{bits}|\!)
\end{aligned}$$

Let `O` send just one bit: $\vdash \mathtt{i.b1}; \mathtt{fix\ w\ in\ w} :: \mathtt{i} : \mathtt{bits}$.

$$\begin{aligned}
&(\!|\vdash \mathtt{i} : \mathtt{bits} \leftarrow \mathtt{O}; \mathtt{BuffF} :: \mathtt{o} : \mathtt{bits}|\!) \\
&= (\mathtt{o} : \bot_{\mathtt{bits}}) \\
&\neq (\mathtt{o} : (\mathtt{b0}, \bot_{\mathtt{bits}})) \\
&= (\!|\vdash \mathtt{i} : \mathtt{bits} \leftarrow \mathtt{O}; \mathtt{F} :: \mathtt{o} : \mathtt{bits}|\!).
\end{aligned}$$

`BuffF` and `F` are **not observationally congruent**!

Other Protocols

- External choice: $\&\{l : A_l\}_{l \in L}$
- Channel transmission: $A \otimes B$ and $A \multimap B$
- Synchronization: $\uparrow A$ and $\downarrow A$
- Functional value transmission: $\tau \wedge A$ and $\tau \supset A$

Observed Communication Semantics, More Formally

Substructural Operational Semantics

Languages are specified by a substructural operational semantics (a multiset rewriting system).

Substructural Operational Semantics

Languages are specified by a substructural operational semantics (a multiset rewriting system).

Two kinds of judgments in the multisets:

1. $\mathrm{proc}(c, P)$: process P provides channel c;

Substructural Operational Semantics

Languages are specified by a substructural operational semantics (a multiset rewriting system).

Two kinds of judgments in the multisets:

1. $\mathrm{proc}(c, P)$: process P provides channel c;
2. $\mathrm{msg}(c, m; c \leftarrow d)$: channel c carries a message m with continuation channel d:

$$\mathrm{msg}(c_1, m_1; c_1 \leftarrow c_2), \mathrm{msg}(c_2, m_2; c_2 \leftarrow c_3),$$
$$\mathrm{msg}(c_3, m_3; c_3 \leftarrow c_4), \ldots$$

Substructural Operational Semantics

Languages are specified by a substructural operational semantics (a multiset rewriting system).

~~Two~~ Three kinds of judgments in the multisets:

1. $\text{proc}(c, P)$: process P provides channel c;
2. $\text{msg}(c, m; c \leftarrow d)$: channel c carries a message m with continuation channel d:

$$\text{msg}(c_1, m_1; c_1 \leftarrow c_2), \text{msg}(c_2, m_2; c_2 \leftarrow c_3),$$
$$\text{msg}(c_3, m_3; c_3 \leftarrow c_4), \ldots$$

3. $\text{type}(c : A)$: channel c has type A

Example Multiset Rewrite Rules

Unfolding recursive processes:

$$\text{proc}(\mathtt{c}, \mathtt{fix\ p\ in\ P}) \rightarrow \text{proc}(\mathtt{c}, [\mathtt{fix\ p\ in\ P}/\mathtt{p}]\,\mathtt{P})$$

Example Multiset Rewrite Rules

Unfolding recursive processes:

$$\text{proc}(\mathtt{c}, \mathtt{fix\ p\ in\ P}) \to \text{proc}(\mathtt{c}, [\mathtt{fix\ p\ in\ P}/\mathtt{p}]\,\mathtt{P})$$

Sending labels:

$$\text{proc}(\mathtt{c}, \mathtt{c}.\mathtt{k}; \mathtt{P}), \text{type}(\mathtt{c} : \oplus\{\mathtt{l} : \mathtt{A_l}\}_{\mathtt{l} \in \mathtt{L}}) \to$$
$$\exists\, \mathtt{d}.\text{msg}(\mathtt{c}, \mathtt{c}.\mathtt{k}; \mathtt{c} \leftarrow \mathtt{d}), \text{proc}(\mathtt{d}, [\mathtt{d}/\mathtt{c}]\mathtt{P}), \text{type}(\mathtt{d} : \mathtt{A_k})$$

Example Multiset Rewrite Rules

Unfolding recursive processes:

$$\mathsf{proc}(\mathtt{c}, \mathtt{fix\ p\ in\ P}) \to \mathsf{proc}(\mathtt{c}, [\mathtt{fix\ p\ in\ P}/\mathtt{p}]\,\mathtt{P})$$

Sending labels:

$$\mathsf{proc}(\mathtt{c}, \mathtt{c.k}; \mathtt{P}), \mathsf{type}(\mathtt{c} : \oplus\{\mathtt{l} : \mathtt{A_l}\}_{\mathtt{l} \in \mathtt{L}}) \to$$
$$\exists\, \mathtt{d}.\mathsf{msg}(\mathtt{c}, \mathtt{c.k}; \mathtt{c} \leftarrow \mathtt{d}), \mathsf{proc}(\mathtt{d}, [\mathtt{d}/\mathtt{c}]\mathtt{P}), \mathsf{type}(\mathtt{d} : \mathtt{A_k})$$

Receiving labels:

$$\mathsf{msg}(\mathtt{c}, \mathtt{c.k}; \mathtt{c} \leftarrow \mathtt{d}), \mathsf{proc}(\mathtt{e}, \mathtt{case\ c}\ \{\mathtt{l} \Rightarrow \mathtt{P_l}\}) \to \mathsf{proc}(\mathtt{e}, [\mathtt{d}/\mathtt{c}]\mathtt{P_k})$$

Executions

An **execution** of a process $c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ is a maximal trace starting from

$$\text{type}(c_0 : A_0), \ldots, \text{type}(c_n : A_n), \text{proc}(c_0, P).$$

Example Execution: Sending Streams

proc(o, fix S in o.b1; o.b0; S)

Example Execution: Sending Streams

proc(o, fix S in o.b1; o.b0; S)

Example Execution: Sending Streams

$\mathrm{proc}(\mathtt{o}, \mathtt{fix\ S\ in\ o.b1; o.b0; S})$

$\rightarrow \mathrm{proc}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o.b0; fix\ S\ in\ o.b1; o.b0; S})$

Example Execution: Sending Streams

$\mathsf{proc}(\mathtt{o}, \mathtt{fix\ S\ in\ o.b1; o.b0; S})$

$\rightarrow \mathsf{proc}(\mathtt{o}, \mathtt{o.b1; o.b0; fix\ S\ in\ o.b1; o.b0; S})$

$\rightarrow \mathsf{msg}(\mathtt{o}, \mathtt{o.b1; o} \leftarrow \mathtt{o_1}), \mathsf{proc}(\mathtt{o_1}, \mathtt{o_1.b0; fix\ S\ in\ o_1.b1; o_1.b0; S})$

Example Execution: Sending Streams

$\text{proc}(\texttt{o}, \texttt{fix S in o.b1; o.b0; S})$

$\rightarrow \text{proc}(\texttt{o}, \texttt{o.b1; o.b0; fix S in o.b1; o.b0; S})$

$\rightarrow \text{msg}(\texttt{o}, \texttt{o.b1; o} \leftarrow \texttt{o}_1), \text{proc}(\texttt{o}_1, \texttt{o}_1\texttt{.b0; fix S in o}_1\texttt{.b1; o}_1\texttt{.b0; S})$

Example Execution: Sending Streams

$$\mathsf{proc}(o, \mathtt{fix}\ S\ \mathtt{in}\ o.b1; o.b0; S)$$

$$\to \mathsf{proc}(o, o.b1; o.b0; \mathtt{fix}\ S\ \mathtt{in}\ o.b1; o.b0; S)$$

$$\to \mathsf{msg}(o, o.b1; o \leftarrow o_1), \mathsf{proc}(o_1, o_1.b0; \mathtt{fix}\ S\ \mathtt{in}\ o_1.b1; o_1.b0; S)$$

$$\to \mathsf{msg}(o, o.b1; o \leftarrow o_1), \mathsf{msg}(o_1, o_1.b0; o_1 \leftarrow o_2),$$
$$\mathsf{proc}(o_2, \mathtt{fix}\ S\ \mathtt{in}\ o_2.b1; o_2.b0; S)$$

Example Execution: Sending Streams

$\mathsf{proc}(\mathtt{o}, \mathtt{fix\ S\ in\ o.b1; o.b0; S})$

$\rightarrow \mathsf{proc}(\mathtt{o}, \mathtt{o.b1; o.b0; fix\ S\ in\ o.b1; o.b0; S})$

$\rightarrow \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{proc}(\mathtt{o}_1, \mathtt{o}_1\mathtt{.b0}; \mathtt{fix\ S\ in\ } \mathtt{o}_1\mathtt{.b1}; \mathtt{o}_1\mathtt{.b0}; \mathtt{S})$

$\rightarrow \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{msg}(\mathtt{o}_1, \mathtt{o}_1\mathtt{.b0}; \mathtt{o}_1 \leftarrow \mathtt{o}_2),$

$\mathsf{proc}(\mathtt{o}_2, \mathtt{fix\ S\ in\ } \mathtt{o}_2\mathtt{.b1}; \mathtt{o}_2\mathtt{.b0}; \mathtt{S})$

Example Execution: Sending Streams

$$\mathsf{proc}(\mathtt{o}, \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o.b1}; \mathtt{o.b0}; \mathtt{S})$$

$$\to \mathsf{proc}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o.b0}; \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o.b1}; \mathtt{o.b0}; \mathtt{S})$$

$$\to \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{proc}(\mathtt{o}_1, \mathtt{o}_1\mathtt{.b0}; \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o}_1\mathtt{.b1}; \mathtt{o}_1\mathtt{.b0}; \mathtt{S})$$

$$\to \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{msg}(\mathtt{o}_1, \mathtt{o}_1\mathtt{.b0}; \mathtt{o}_1 \leftarrow \mathtt{o}_2),$$
$$\mathsf{proc}(\mathtt{o}_2, \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o}_2\mathtt{.b1}; \mathtt{o}_2\mathtt{.b0}; \mathtt{S})$$

$$\to \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{msg}(\mathtt{o}_1, \mathtt{o}_1\mathtt{.b0}; \mathtt{o}_1 \leftarrow \mathtt{o}_2),$$
$$\mathsf{proc}(\mathtt{o}_2, \mathtt{o}_2\mathtt{.b1}; \mathtt{o}_2\mathtt{.b0}; \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o}_2\mathtt{.b1}; \mathtt{o}_2\mathtt{.b0}; \mathtt{S})$$

Example Execution: Sending Streams

$$\mathsf{proc}(\mathtt{o}, \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o.b1}; \mathtt{o.b0}; \mathtt{S})$$

$$\to \mathsf{proc}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o.b0}; \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o.b1}; \mathtt{o.b0}; \mathtt{S})$$

$$\to \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{proc}(\mathtt{o}_1, \mathtt{o}_1.\mathtt{b0}; \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o}_1.\mathtt{b1}; \mathtt{o}_1.\mathtt{b0}; \mathtt{S})$$

$$\to \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{msg}(\mathtt{o}_1, \mathtt{o}_1.\mathtt{b0}; \mathtt{o}_1 \leftarrow \mathtt{o}_2), \quad \mathsf{proc}(\mathtt{o}_2, \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o}_2.\mathtt{b1}; \mathtt{o}_2.\mathtt{b0}; \mathtt{S})$$

$$\to \mathsf{msg}(\mathtt{o}, \mathtt{o.b1}; \mathtt{o} \leftarrow \mathtt{o}_1), \mathsf{msg}(\mathtt{o}_1, \mathtt{o}_1.\mathtt{b0}; \mathtt{o}_1 \leftarrow \mathtt{o}_2), \quad \mathsf{proc}(\mathtt{o}_2, \mathtt{o}_2.\mathtt{b1}; \mathtt{o}_2.\mathtt{b0}; \mathtt{fix}\ \mathtt{S}\ \mathtt{in}\ \mathtt{o}_2.\mathtt{b1}; \mathtt{o}_2.\mathtt{b0}; \mathtt{S})$$

$$\to \cdots$$

Session-Typed Communications

A **session-typed communication** is a (potentially infinite) tree v generated by the grammar:

$$v ::= \bot_A \mid (k, v) \mid (v, v') \mid \cdots$$

They are associated to session types by rules, e.g.,

$$\frac{}{\bot_{\oplus\{l:A_l\}_{l\in L}} \;\varepsilon\; \oplus\{l : A_l\}_{l\in L}} \qquad \frac{v_k \;\varepsilon\; A_k}{(k, v_k) \;\varepsilon\; \oplus\{l : A_l\}_{l\in L}}$$

Session-Typed Communications

A **session-typed communication** is a (potentially infinite) tree v generated by the grammar:

$$v ::= \bot_A \mid (k, v) \mid (v, v') \mid \cdots$$

They are associated to session types by rules, e.g.,

$$\frac{}{\bot_{\oplus\{l:A_l\}_{l\in L}} \;\varepsilon\; \oplus\{l : A_l\}_{l\in L}} \qquad \frac{v_k \;\varepsilon\; A_k}{(k, v_k) \;\varepsilon\; \oplus\{l : A_l\}_{l\in L}}$$

Given a trace T, the judgment $T \rightsquigarrow v \;\varepsilon\; A \,/\, c$ means we observed a session-typed communication v of type A on channel c in T.

It is coinductively defined using the union of multisets in T (without repetitions).

From Traces to Observed Communications

The **observation** of a process $c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0$ is the $(n+1)$-tuple

$$(\!| c_1 : A_1, \ldots, c_n : A_n \vdash P :: c_0 : A_0 |\!) = (c_0 : v_0, \ldots, c_n : v_n)$$

where $T \rightsquigarrow v_i \,\varepsilon\, A_i \,/\, c_i$ for $0 \leq i \leq n$.

Problems. Are observations unique? Does it make sense for it to be unique? What about unfair executions?

Unfair Executions: Pathological Example

Let $L = \oplus\{l : L\}$, and let Ω and B respectively be

$$\begin{aligned} &\vdash \texttt{fix w in w} &&\texttt{:: a : A} \\ \texttt{a : A} &\vdash \texttt{fix p in c.l; p} &&\texttt{:: c : L} \end{aligned}$$

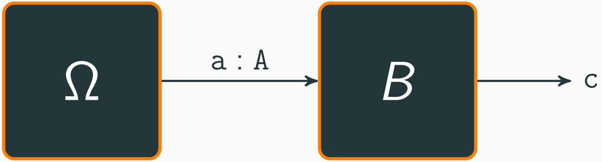
Unfair Executions: Pathological Example

Let $L = \oplus\{l : L\}$, and let Ω and B respectively be

```
      ⊢ fix w in w       :: a : A
a : A ⊢ fix p in c.l; p :: c : L
```

We can compose them to get:

```
⊢ a : A ← Ω ; B :: c : L
```



An observation is a tuple $(\texttt{c} : v)$ where $v \mathrel{\varepsilon} L$.

Fairness for Multiset Rewriting Systems

Multiset Rewriting Systems

A **multiset rewrite rule** r is a pair of multisets $F(\vec{x})$ and $G(\vec{x}, \vec{n})$ of first-order atomic formulas.

$$r : \forall \vec{x}. F(\vec{x}) \rightarrow \exists \vec{n}. G(\vec{x}, \vec{n}).$$

Multiset Rewriting Systems

A **multiset rewrite rule** r is a pair of multisets $F(\vec{x})$ and $G(\vec{x}, \vec{n})$ of first-order atomic formulas.

$$r : \forall \vec{x}. F(\vec{x}) \rightarrow \exists \vec{n}. G(\vec{x}, \vec{n}).$$

A **multiset rewriting system** is a set of multiset rewrite rules.

Multiset Rewriting Systems

A **multiset rewrite rule** r is a pair of multisets $F(\vec{x})$ and $G(\vec{x}, \vec{n})$ of first-order atomic formulas.

$$r : \forall \vec{x}. F(\vec{x}) \to \exists \vec{n}. G(\vec{x}, \vec{n}).$$

A **multiset rewriting system** is a set of multiset rewrite rules.

Given some $\vec{c}$, a **rule instantiation**

$$r(\vec{c}) : F(\vec{c}) \to \exists \vec{n}. G(\vec{c}, \vec{n})$$

is **applicable** to M if $M = F(\vec{c}), M'$.

Multiset Rewriting Systems

A **multiset rewrite rule** r is a pair of multisets $F(\vec{x})$ and $G(\vec{x}, \vec{n})$ of first-order atomic formulas.

$$r : \forall \vec{x}. F(\vec{x}) \rightarrow \exists \vec{n}. G(\vec{x}, \vec{n}).$$

A **multiset rewriting system** is a set of multiset rewrite rules.

Given some $\vec{c}$, a **rule instantiation**

$$r(\vec{c}) : F(\vec{c}) \rightarrow \exists \vec{n}. G(\vec{c}, \vec{n})$$

is **applicable** to M if $M = F(\vec{c}), M'$.

The **result** of applying $r(\vec{c})$ to M is $G(\vec{c}, \vec{d}), M'$, where $\vec{d}$ are fresh constants. Write $M \xrightarrow{(r;(\vec{c},\vec{d}))} G(\vec{c}, \vec{d}), M'$.

Über Fairness

A **trace** is a sequence of multisets related by rule applications:

$$M_0 \xrightarrow{(r_1;(\vec{c}_1,\vec{d}_1))} M_1 \xrightarrow{(r_2;(\vec{c}_2,\vec{d}_2))} M_2 \xrightarrow{(r_3;(\vec{c}_3,\vec{d}_3))} \cdots$$

where at each step the $\vec{d}_i$ are globally fresh.

Über Fairness

A **trace** is a sequence of multisets related by rule applications:

$$M_0 \xrightarrow{(r_1;(\vec{c}_1,\vec{d}_1))} M_1 \xrightarrow{(r_2;(\vec{c}_2,\vec{d}_2))} M_2 \xrightarrow{(r_3;(\vec{c}_3,\vec{d}_3))} \cdots$$

where at each step the $\vec{d}_i$ are globally fresh.

It is **(über) fair** if for $r \in \mathcal{R}$, $\vec{c}$, and i, if $r(\vec{c})$ is applicable to M_i, then there exists a $j > i$ such that $r(\vec{c}) \equiv r_j(\vec{c}_j)$.

Interference-Freedom

An MRS is **interference-free** on M_0 if, where $r_1(\vec{c}_1), \ldots, r_n(\vec{c}_n)$ are the rule instantiations applicable to M_0, then all possible application orderings are valid and result in the same multiset.

Interference-Freedom

An MRS is **interference-free** on M_0 if, where $r_1(\vec{c}_1), \ldots, r_n(\vec{c}_n)$ are the rule instantiations applicable to M_0, then all possible application orderings are valid and result in the same multiset.

It is **interference-free from** M_0 if for each trace from M_0, it is interference-free on each M_i in the trace.

Non-Overlapping MRSs

Proposition

Consider rules $r_i : \forall \vec{x}_i . F_i(\vec{x}_i) \to \exists \vec{n}_i . G_i(\vec{x}_i, \vec{n}_i)$.
If $F_1(\vec{c}_1), \dots, F_n(\vec{c}_n)$ *are "non-overlapping"* M*, then the rules are interference-free on* M.

Example

The rules for session-typed processes are non-overlapping on every multiset in a process trace.

Interference-Freedom: Fair Scheduler

Proposition

If an MRS is interference-free on M_0, then there exists a fair maximal trace from M_0.

Permutations

Consider a trace T (indexed by $i \in I$)

$$M_0 \xrightarrow{(r_1;(\vec{c}_1,\vec{d}_1))} M_1 \xrightarrow{(r_2;(\vec{c}_2,\vec{d}_2))} M_2 \xrightarrow{(r_3;(\vec{c}_3,\vec{d}_3))} \cdots$$

Given a permutation σ of I, a sequence $\sigma \cdot T$

$$M_0 \xrightarrow{(r_{\sigma(1)};(\vec{c}_{\sigma(1)},\vec{d}_{\sigma(1)}))} M_1' \xrightarrow{(r_{\sigma(2)};(\vec{c}_{\sigma(2)},\vec{d}_{\sigma(2)}))} M_2' \xrightarrow{(r_{\sigma(3)};(\vec{c}_{\sigma(3)},\vec{d}_{\sigma(3)}))} \cdots$$

is called a **permutation of** T if it is also a trace.

Permutation and Fairness

Theorem

If an MRS is interference-free from M, T is a fair trace from M, and $\sigma \cdot T$ is a permutation of T, then $\sigma \cdot T$ is also fair.

Permutation and Fairness

Theorem

If an MRS is interference-free from M, T is a fair trace from M, and $\sigma \cdot T$ is a permutation of T, then $\sigma \cdot T$ is also fair.

Theorem

If an MRS is interference-free from M, then all fair traces from M are permutations of each other.

Permutation and Fairness

Theorem

If an MRS is interference-free from M, T is a fair trace from M, and $\sigma \cdot T$ is a permutation of T, then $\sigma \cdot T$ is also fair.

Theorem

If an MRS is interference-free from M, then all fair traces from M are permutations of each other.

Corollary

All fair executions of a session-typed process are permutations of each other.

Union-Equivalence

Two traces

$$M_0 \xrightarrow{(r_1;(\vec{c}_1,\vec{d}_1))} M_1 \xrightarrow{(r_2;(\vec{c}_2,\vec{d}_2))} M_2 \xrightarrow{(r_3;(\vec{c}_3,\vec{d}_3))} \cdots$$

$$N_0 \xrightarrow{(r'_1;(\vec{c}'_1,\vec{d}'_1))} N_1 \xrightarrow{(r'_2;(\vec{c}'_2,\vec{d}'_2))} N_2 \xrightarrow{(r'_3;(\vec{c}'_3,\vec{d}'_3))} \cdots$$

are **union-equivalent** if $\bigcup \mathrm{supp}(M_i) = \bigcup \mathrm{supp}(N_i)$.

Union-Equivalence

Two traces

$$M_0 \xrightarrow{(r_1;(\vec{c}_1,\vec{d}_1))} M_1 \xrightarrow{(r_2;(\vec{c}_2,\vec{d}_2))} M_2 \xrightarrow{(r_3;(\vec{c}_3,\vec{d}_3))} \cdots$$

$$N_0 \xrightarrow{(r'_1;(\vec{c}'_1,\vec{d}'_1))} N_1 \xrightarrow{(r'_2;(\vec{c}'_2,\vec{d}'_2))} N_2 \xrightarrow{(r'_3;(\vec{c}'_3,\vec{d}'_3))} \cdots$$

are **union-equivalent** if $\bigcup \mathrm{supp}(M_i) = \bigcup \mathrm{supp}(N_i)$.

Theorem

If an MRS is interference-free from M, then all fair traces from M are union-equivalent.

Union-Equivalence

Two traces

$$M_0 \xrightarrow{(r_1;(\vec{c}_1,\vec{d}_1))} M_1 \xrightarrow{(r_2;(\vec{c}_2,\vec{d}_2))} M_2 \xrightarrow{(r_3;(\vec{c}_3,\vec{d}_3))} \cdots$$

$$N_0 \xrightarrow{(r'_1;(\vec{c}'_1,\vec{d}'_1))} N_1 \xrightarrow{(r'_2;(\vec{c}'_2,\vec{d}'_2))} N_2 \xrightarrow{(r'_3;(\vec{c}'_3,\vec{d}'_3))} \cdots$$

are **union-equivalent** if $\bigcup \mathrm{supp}(M_i) = \bigcup \mathrm{supp}(N_i)$.

Theorem

If an MRS is interference-free from M, then all fair traces from M are union-equivalent.

Corollary

All fair executions of a session-typed process are union-equivalent.

Unique Observations for Processes

Recall that observations are defined in terms of the union of the supports of the multisets in a process execution.

If we restrict our attention fair executions,
then **every process has a unique observation**!

Contributions

An **observed communication semantics** for session-typed languages **with recursion** that are specified by **substructural operational semantics** (multiset rewriting systems).

A notion of **fairness for multiset rewriting systems**.

- Sufficient conditions for a fair scheduler
- Associated reasoning principles
- Various properties of fair traces

Future Work

This work is still in its early stages!

- Relate the operational observation to a domain-theoretic denotational semantics
- Relate to existing notions of operational observation and equivalence: barbed congruence, bisimulation, etc.

Contributions

An **observed communication semantics** for session-typed languages **with recursion** that are specified by **substructural operational semantics** (multiset rewriting systems).

A notion of **fairness for multiset rewriting systems**.

- Sufficient conditions for a fair scheduler
- Associated reasoning principles
- Various properties of fair traces

Buffered Bit Flipping

```
i : bits ⊢ fix F in case i
           { b0 => case i { b0 => o.b1; o.b1; F
                          | b1 => o.b1; o.b0; F }
           | b1 => case i { b0 => o.b0; o.b1; F
                          | b1 => o.b0; o.b0; F }
         :: o : bits
```

[back]

Related Work i

- Robert Atkey
  Observed Communication Semantics for Classical Processes
  ESOP 2017, LNCS 10201, pp. 56–82, 2017.
- Iliano Cervesato, Nancy Durgin, Patrick Lincoln et. al
  A Comparison Between Strand Spaces and Multiset Rewriting for Security Protocol Analysis
  Journal of Computer Security **13**(2), pp. 265-316, 2005.

Related Work ii

Wen Kokke, Fabrizio Montesi & Marco Peressotti
Better Late Than Never: A Fully-Abstract Semantics for Classical Processes
PACMPL **4**(POPL):24, 2019.