Session Types and Higher-Order Concurrency

Jorge A. Pérez
University of Groningen, The Netherlands
www.jperez.nl

Joint work with

Alen Arslanagić, Dimitrios Kouzapas, Nobuko Yoshida, and Erik Voogd

(based on papers in Inf & Comp'19 and ECOOP'19)

UNIFYING
C•RRECTNESS FOR
C•MMUNICATING
S•FTWARE

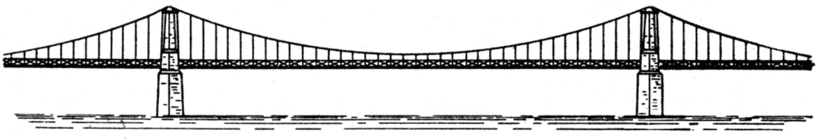
OWLS - February 24, 2021

Keywords and Slogans

Concurrency Theory, Message-Passing, Programming Languages, Verification

- **Type systems**
  Slogan: Well-typed programs can't go wrong (Milner)
- **Process calculi**
  Slogan: The π-calculus treats **processes** like the λ-calculus treats **functions**
- **Session types** for communication correctness
  Slogan: **What** and **when** should be sent through a channel
- **Relative expressiveness** of (typed) programming calculi

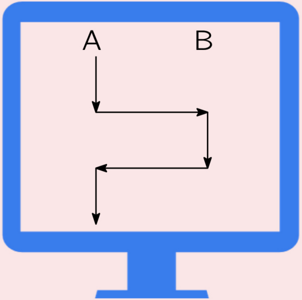
This Talk: Bridging Functions and Concurrency



- Bridges between functional and concurrent programming calculi
  $\rightarrow$ **Encodings** as formal compilers (language translations)
- Encodings informed by **session types**:
  - Protocols guide encoding definitions
  - Linearity is key to enforce optimizations
  - Encoding correctness based on prior work on typed equivalences [CONCUR'15]
- Type-based **extensions of known encodings**
- **New encodings** not available in untyped settings

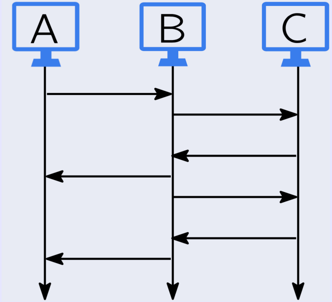
When is a Program Correct?

Sequential Programs



"Programs produce outputs that are consistent with their input"

Concurrent Programs



"Programs always respect their intended protocols"

Type Systems

Sequential Languages

- **Data type systems** classify values in a program
- Examples: Integers, strings of characters

Concurrent Languages

- **Behavioral type systems** classify protocols in a program
- Example: "first send username, then receive true/false, finally close"
- A typical bug: sending messages in the wrong order

Protocols as Session Types

Session types uniformly describe protocols in terms of

- communication actions (input and output)
- labeled choices (offers and selections)
- sequential composition
- recursion

Session protocols are attached to **interaction devices**:

- π-calculus names
- service endpoints
- Go channels
- $\cdots$

Sequentiality in types goes **hand-in-hand** with sequentiality in processes

Protocols as Session Types

$S ::=$	$!U;S$	**output** value of type U, continue as S
$\mid$	$?U;S$	**input** value of type U, continue as S
$\mid$	$\&\{l_i : S_i\}_{i \in I}$	**offer** a selection between $S_1, \ldots, S_n$
$\mid$	$\oplus\{l_i : S_i\}_{i \in I}$	**select** between $S_1, \ldots, S_n$
$\mid$	$\mu t.S \mid t$	**recursion**
$\mid$	$\mathtt{end}$	**terminated protocol**

(Labels $l_1, \ldots, l_n$ are pairwise different.)

Notice:

- U stands for basic values (e.g. `int`) but also sessions S (aka delegation)
- Sequential communication patterns (no built-in concurrency)

Session-Based Concurrency

Two phases:

I. Services advertise their session protocols along **channel names**.
Agreements are realized by their point-to-point interaction,
in an **unrestricted** and **non-deterministic** way.

II. After agreement, services establish a session using **session names**.
Intra-session interactions follow the intended protocol,
in a **linear** and **deterministic** way.

Notice:

- 'Linear' and 'unrestricted' in the sense of Girard's **linear logic**.

Challenge

- Many behavioral type systems!
- Correctness via various **behavioral properties**
  - Protocol fidelity, comm. safety, deadlock-freedom
- Different type systems, properties and insights
- A program can be both correct *and* incorrect!

Relative Expressiveness

Connect behavioral type systems
by relating the **concurrent languages** on which they operate

Goals

✓ **Encodability result**:
A correct compiler between two concurrent languages

✗ **Separation result**:
A proof that a correct compiler does not exist

Highlights:

⇒ A **general**, **rigorous**, **flexible**, and **practical** approach

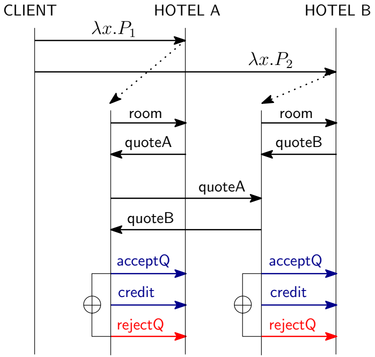
Higher-Order Concurrency

- Process languages in which values may contain processes
- Natural bridge between the λ-calculus and process calculi
- Key example: the higher-order π-calculus

A celebrated result, by Sangiorgi (1992)

- Process passing is **representable** using name passing
- Encoding is **fully abstract** wrt barbed congruence (contextual equivalence)
- Highlights **significance** of the π-calculus
- Enables **transfer** reasoning techniques

Higher-Order **Session** Concurrency



Two alternative sources:

- Higher-order π-calculi
  + **session communication** (establishment, input/output, labeled choice)
- Session π-calculi
  + passing of **abstractions** $\lambda x.P$ (functions from names to processes)

Higher-order π-calculus with sessions (HOπ)

n, m	$::= a, b \mid s, \overline{s}$	names: shared and linear
u, w	$::= n \mid x, y, z$	name identifiers
V, W	$::= u \mid \lambda x.\, P \mid x, y, z$	values: names, **abstractions**
P, Q	$::= u?(x).P \mid u!\langle V\rangle.P$	input / output
	$\mid\ u \triangleright \{l_i : P_i\}_{i\in I} \mid u \triangleleft l.P$	labeled choice
	$\mid\ X \mid \mu X.P$	recursion
	$\mid\ P \mid Q \mid (\nu\, n)P \mid 0$	parallel, restriction, inaction
	$\mid\ V\, u$	**name application**

Reduction Semantics: Key Rules

$$(\lambda x.\, P)\, u \longrightarrow P\{u/x\}$$

$$n!\langle V\rangle.P \mid \overline{n}?(x).Q \longrightarrow P \mid Q\{V/x\}$$

$$n \triangleleft l_j.Q \mid \overline{n} \triangleright \{l_i : P_i\}_{i\in I} \longrightarrow Q \mid P_j \quad (j \in I)$$

Example: Two Different Clients in HOπ

$$\mathsf{Client}_1 \triangleq (\nu\, h_1, h_2)(s_1!\Big\langle \lambda x.\, P_{xy}\{^{h_1}/_{y}\}\Big\rangle.s_2!\Big\langle \lambda x.\, P_{xy}\{^{h_2}/_{y}\}\Big\rangle.\mathbf{0} \mid \overline{h_1}?(x).\overline{h_2}?(y).\mathtt{if}\ x \leq y\ \mathtt{then}\ (\overline{h_1} \triangleleft \mathsf{accept}.\overline{h_2} \triangleleft \mathsf{reject}.\mathbf{0}\ \mathtt{else}\ \ \overline{h_1} \triangleleft \mathsf{reject}.\overline{h_2} \triangleleft \mathsf{accept}.\mathbf{0}))$$

$$P_{xy} \triangleq x!\langle \mathsf{room}\rangle.x?(quote).y!\langle quote\rangle.y \triangleright \left\{ \begin{array}{l} \mathsf{accept} : x \triangleleft \mathsf{accept}.x!\langle \mathsf{credit}\rangle.\mathbf{0}\ , \\ \mathsf{reject} : x \triangleleft \mathsf{reject}.\mathbf{0} \end{array} \right\}$$

$$\mathsf{Client}_2 \triangleq (\nu\, h)(s_1!\Big\langle \lambda x.\, Q_1\{^{h}/_{y}\}\Big\rangle.s_2!\Big\langle \lambda x.\, Q_2\{^{\overline{h}}/_{y}\}\Big\rangle.\mathbf{0})$$

$$Q_1 \triangleq x!\langle \mathsf{room}\rangle.x?(quote_1).y!\langle quote_1\rangle.y?(quote_2).R_x$$

$$Q_2 \triangleq x!\langle \mathsf{room}\rangle.x?(quote_1).y?(quote_2).y!\langle quote_1\rangle.R_x$$

$$R_x \triangleq \mathtt{if}\ \ quote_1 \leq quote_2\ \mathtt{then}\ (x \triangleleft \mathsf{accept}.x!\langle \mathsf{credit}\rangle.\mathbf{0}\ \mathtt{else}\ \ x \triangleleft \mathsf{reject}.\mathbf{0})$$

Session Types for HOπ

C	$::=$	$S \mid \langle S \rangle \mid \langle L \rangle$	first-order types
L	$::=$	$C {\rightarrow} \diamond \mid C {\multimap} \diamond$	functional types (shared / linear)
U	$::=$	$C \mid L$	value types
S	$::=$	$!\langle U \rangle; S$	output
	$\mid$	$?(U); S$	input
	$\mid$	$\oplus\{l_i : S_i\}_{i \in I}$	selection
	$\mid$	$\&\{l_i : S_i\}_{i \in I}$	branching
	$\mid$	$\mu \mathsf{t}.S \mid \mathsf{t} \mid \mathtt{end}$	recursive and terminated type

Judgements for values and processes:

$$\Gamma; \Lambda; \Delta \vdash V \triangleright U \qquad \Gamma; \Lambda; \Delta \vdash P \triangleright \diamond$$

Example: Typing a Client

$$\mathsf{Client}_1 \triangleq (\nu\, h_1, h_2)(s_1!\big\langle \lambda x.\, P_{xy}\{^{h_1}/_{y}\}\big\rangle.s_2!\big\langle \lambda x.\, P_{xy}\{^{h_2}/_{y}\}\big\rangle.0 \mid \overline{h_1}?(x).\overline{h_2}?(y).\mathtt{if}\ x \leq y\ \mathtt{then}\ (\overline{h_1} \triangleleft \mathsf{accept}.\overline{h_2} \triangleleft \mathsf{reject}.0\ \mathtt{else}\ \ \overline{h_1} \triangleleft \mathsf{reject}.\overline{h_2} \triangleleft \mathsf{accept}.0))$$

$$P_{xy} \triangleq x!\langle \mathsf{room}\rangle.x?(quote).y!\langle quote\rangle.y \triangleright \left\{ \begin{array}{l} \mathsf{accept} : x \triangleleft \mathsf{accept}.x!\langle \mathsf{credit}\rangle.0\ , \\ \mathsf{reject} : x \triangleleft \mathsf{reject}.0 \end{array} \right\}$$

A session type (with base types quote, room, and credit):

$$U \quad = \quad !\langle \mathsf{room}\rangle; ?(\mathsf{quote}); \oplus\{\mathsf{accept} : !\langle \mathsf{credit}\rangle; \mathsf{end}, \mathsf{reject} : \mathsf{end}\}$$

Typing judgments:

$$\emptyset; \emptyset; y : !\langle \mathsf{quote}\rangle; \&\{\mathsf{accept} : \mathsf{end}, \mathsf{reject} : \mathsf{end}\} \vdash \lambda x.\, P_{xy} \triangleright U \multimap \diamond$$

$$\emptyset; \emptyset; s_1 : !\langle U \multimap \diamond\rangle; \mathsf{end} \cdot s_2 : !\langle U \multimap \diamond\rangle; \mathsf{end} \vdash \mathsf{Client}_1 \triangleright \diamond$$

Key Questions

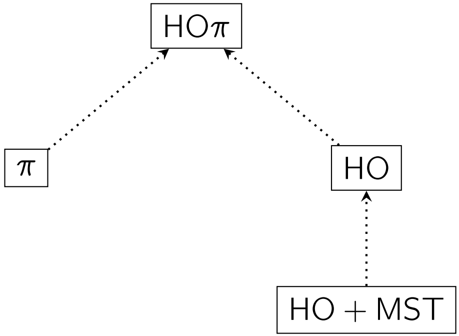
At the level of **processes**, two mechanisms: name passing and abstraction passing (first- and higher-order concurrency).

- Are both mechanisms fundamental?
- Can one of them be represented using the other?

At the level of **types**:

- To what extent the structure of session types play a role?

Sub-languages of HOπ



- HO isolates **higher-order** features: only abstraction passing, no name passing
- π isolates **first-order** features: only name passing, no abstraction passing
- HO + **MST** is as HO but without sequentiality in session types

Sub-calculi of HOπ

n, m	$::=$	$a, b \mid s, \overline{s}$	names: shared and linear
u, w	$::=$	$n \mid x, y, z$	name identifiers
V, W	$::=$	$\boxed{u} \mid \boxed{x, y, z} \mid \boxed{\lambda x.\, P}$	values: names, abstractions
P, Q	$::=$	$u?(x).P \mid u!\langle V\rangle.P$	input / output
	$\mid$	$u \triangleright \{l_i : P_i\}_{i \in I} \mid u \triangleleft l.P$	labeled choice
	$\mid$	$\boxed{X \mid \mu X.P}$	recursion
	$\mid$	$P \mid Q \mid (\nu\, n)P \mid 0$	parallel, restriction, inaction
	$\mid$	$\boxed{V\, u}$	name application

- HO lacks shaded constructs
- π lacks boxed constructs

Session Types for HO and π

$$
\begin{array}{llll}
C & ::= & S \mid \langle S \rangle \mid \langle L \rangle & \text{first-order types} \\
L & ::= & C \rightarrow \diamond \mid C \multimap \diamond & \text{shared / linear functional types} \\
U & ::= & \boxed{C} \mid \boxed{L} & \text{value types} \\
S & ::= & !\langle U \rangle; S & \text{output} \\
 & \mid & ?(U); S & \text{input} \\
 & \mid & \oplus\{l_i : S_i\}_{i \in I} & \text{selection} \\
 & \mid & \&\{l_i : S_i\}_{i \in I} & \text{branching} \\
 & \mid & \mu \mathtt{t}.S \mid \mathtt{t} \mid \mathtt{end} & \text{recursive and terminated type}
\end{array}
$$

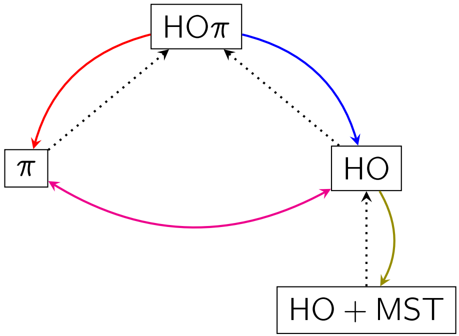
- Types for HO lack shaded constructs
- Types for π lack boxed constructs

Minimal Session Types for HO

$$
\begin{array}{llll}
C & ::= & S \mid \langle U \rangle & \text{value types} \\
U & ::= & \widetilde{C} \rightarrow \diamond \mid \widetilde{C} \multimap \diamond & \text{functional types} \\
S & ::= & \boxed{!\langle \widetilde{U} \rangle; \mathtt{end}} & \text{output} \\
 & \mid & \boxed{?(\widetilde{U}); \mathtt{end}} & \text{input} \\
 & \mid & \oplus\{l_i : S_i\}_{i \in I} & \text{selection} \\
 & \mid & \&\{l_i : S_i\}_{i \in I} & \text{branching} \\
 & \mid & \mu \mathtt{t}.S \mid \mathtt{t} \mid \mathtt{end} & \text{recursive and terminated type}
\end{array}
$$

- \- Sequentiality in types
- \+ Polyadic communication

Expressivity Results for HOπ



HOπ and its sub-calculi are **equally expressive**

- Encoding HOπ **into** π
  Refines Sangiorgi's with session types
- Encoding HOπ **into** HO
  New encoding, even in untyped settings

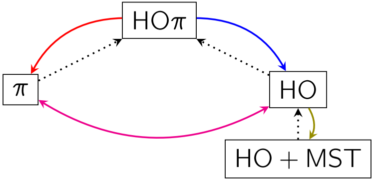
Minimal Session Types for HO

- Session types explained in terms of themselves
- Closer to types in actual PLs

HOπ encodes its **extensions**

- Higher-order abstractions
- Polyadic communication
- Their super-calculus

This Talk

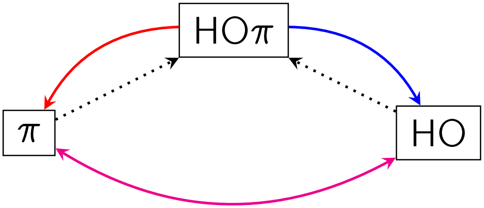


- Encoding HOπ into π and HO
- Minimal session types for HO

Further Results

- The notion of **precise encodings**
- New typed equivalence for HOπ
- Encoding extensions of HOπ into HOπ
- Negative result, using **minimal encodings**:
  session names can't encode shared communication
- Comparing HO and π, using **tight encodings**

Two **Precise** Encodings



Recall:

- π lacks higher-order features (abstraction passing, application)
- HO lacks first-order features (name passing and recursion)

Approach

- Abstract definition of **precise encoding** (translation + correctness criteria)
- **Instantiate the definition** with typed calculi, typed semantics, equivalences

Encoding #1: HOπ into π

Sangiorgi's encoding refined using **linearity**.

Translating processes:

$$\llbracket u!\big\langle \lambda x.\, Q \big\rangle.P \rrbracket \triangleq \begin{cases} (\nu\, a)(u!\langle a\rangle.(\llbracket P\rrbracket \mid \quad a?(y).y?(x).\llbracket Q\rrbracket)\,) & \text{if } Q \text{ is linear} \\ (\nu\, a)(u!\langle a\rangle.(\llbracket P\rrbracket \mid *\, a?(y).y?(x).\llbracket Q\rrbracket)\,) & \text{otherwise} \end{cases}$$

$$\llbracket u?(x).P \rrbracket \triangleq u?(x).\llbracket P\rrbracket$$

$$\llbracket x\, u \rrbracket \triangleq (\nu\, s)(x!\langle s\rangle.\overline{s}!\langle u\rangle.0)$$

$$\llbracket (\lambda x.\, P)\, u \rrbracket \triangleq (\nu\, s)(s?(x).\llbracket P\rrbracket \mid \overline{s}!\langle u\rangle.0)$$

Translating types:

$$\langle\!\langle !\langle S{\multimap}\diamond\rangle;\, T \rangle\!\rangle \triangleq !\Big\langle \langle ?(\langle\!\langle S\rangle\!\rangle); \mathsf{end}\rangle \Big\rangle; \langle\!\langle T\rangle\!\rangle$$

$$\langle\!\langle ?(S{\multimap}\diamond);\, T \rangle\!\rangle \triangleq ?\Big(\langle ?(\langle\!\langle S\rangle\!\rangle); \mathsf{end}\rangle \Big); \langle\!\langle T\rangle\!\rangle$$

Example: Encoding #1 at Work

$$\mathsf{Client}_1 \triangleq (\nu\, h_1, h_2)(s_1!\big\langle \lambda x.\, P_{xy}\{^{h_1}\!/y\}\big\rangle.s_2!\big\langle \lambda x.\, P_{xy}\{^{h_2}\!/y\}\big\rangle.0 \mid$$
$$\overline{h_1}?(x).\overline{h_2}?(y).\mathtt{if}\ x \leq y\ \mathtt{then}$$
$$(\overline{h_1} \triangleleft \mathsf{accept}.\overline{h_2} \triangleleft \mathsf{reject}.0\ \mathtt{else}\ \ \overline{h_1} \triangleleft \mathsf{reject}.\overline{h_2} \triangleleft \mathsf{accept}.0))$$

$$P_{xy} \triangleq x!\langle \mathsf{room}\rangle.x?(quote).y!\langle quote\rangle.y \triangleright \left\{ \begin{array}{l} \mathsf{accept} : x \triangleleft \mathsf{accept}.x!\langle \mathsf{credit}\rangle.0\ , \\ \mathsf{reject} : x \triangleleft \mathsf{reject}.0 \end{array} \right\}$$

$$[\![\mathsf{Client}_1]\!] = (\nu\, h_1, h_2)\Big((\nu\, a_1)(s_1!\langle a_1\rangle.(\nu\, a_2)(s_2!\langle a_2\rangle.$$
$$(0 \mid a_2?(y).y?(x).[\![P_{xy}\{^{h_2}\!/y\}]\!])) \mid a_1?(y).y?(x).[\![P_{xy}\{^{h_1}\!/y\}]\!]) \mid$$
$$\overline{h_1}?(x).\overline{h_2}?(y).\mathtt{if}\ x \leq y\ \mathtt{then}$$
$$(\overline{h_1} \triangleleft \mathsf{accept}.\overline{h_2} \triangleleft \mathsf{reject}.0\ \mathtt{else}\ \ \overline{h_1} \triangleleft \mathsf{reject}.\overline{h_2} \triangleleft \mathsf{accept}.0)\Big)$$

where $[\![P_{xy}]\!] = P_{xy}$, for it does not involve higher-order communication.

Encoding #2: HOπ into HO

Two Challenges

1. Encoding **name passing into abstraction passing**
   No completely satisfactory encoding known in the literature
2. Encoding recursion $\mu X.P$ using session names
   How to model **infinite behavior** using only **linear names**?

Encoding HOπ into HO: Challenge 1 of 2

How to encode the output of a name b along channel a?

Idea: "Pack" b into an abstraction

The receiver "unpacks" b following a protocol on a fresh session:

$$\begin{aligned} [\![a!\langle b\rangle.P]\!] &= a!\langle \lambda z.\ z?(x).(x\, b)\rangle.[\![P]\!] \\ [\![a?(x).Q]\!] &= a?(y).(\nu\, s)(y\, s \mid \overline{s}!\langle \lambda x.\ [\![Q]\!]\rangle.0) \end{aligned}$$

Type preservation: Input/outputs are preserved!

Example: Encoding Name-Passing in HO

With name-passing, we can have the following reduction:

$$n!\langle m\rangle.P \mid \overline{n}?(x).Q \longrightarrow P \mid Q\{{}^{m}\!/\!_{x}\}$$

No name-passing in HO! Using the encoding, we have:

$$\begin{aligned}
[\![n!\langle m\rangle.P \mid \overline{n}?(x).Q]\!] = {} & n!\Big\langle \lambda z.\ z?(x).(x\ m)\Big\rangle.[\![P]\!] \mid n?(y).(\nu\, s)(y\ s \mid \overline{s}!\langle \lambda x.\,[\![Q]\!]\rangle) \\
\longrightarrow {} & [\![P]\!] \mid (\nu\, s)(\lambda z.\ z?(x).(x\ m)\ s \mid \overline{s}!\langle \lambda x.\,[\![Q]\!]\rangle) \\
\longrightarrow {} & [\![P]\!] \mid (\nu\, s)(s?(x).(x\ m) \mid \overline{s}!\langle \lambda x.\,[\![Q]\!]\rangle) \\
\longrightarrow {} & [\![P]\!] \mid (\lambda x.\,[\![Q]\!])\ m \\
\longrightarrow {} & [\![P]\!] \mid [\![Q]\!]\{{}^{m}\!/\!_{x}\}
\end{aligned}$$

Encoding HOπ into HO: Challenge 2 of 2

Key Idea for Translating $\mu X.P$

- Treat P as an **abstraction** with variables instead of names
- Having no linear names, this abstraction can be **duplicated**; its (recursive) type captures infinite behavior

Formally

- Map $\| \cdot \|_.$ converts free session names into name variables.
- Below, $\tilde{n} = \mathtt{fn}(P)$:

$$\llbracket \mu X.P \rrbracket_f \triangleq (\nu\, s)(\overline{s}!\Big\langle \lambda(\|\tilde{n}\|, y).\ y?(z_X).\Big\| \llbracket P \rrbracket_{f,\{X \to \tilde{n}\}} \Big\|_\emptyset \Big\rangle.0 \mid s?(z_X).\llbracket P \rrbracket_{f,\{X \to \tilde{n}\}})$$

$$\llbracket X \rrbracket_f \triangleq (\nu\, s)(z_X\,(\tilde{n}, s) \mid \overline{s}!\langle z_X \rangle.0) \quad (\tilde{n} = f(X))$$

- Moreover: $\langle\!\langle \Gamma \cdot X : \Delta \rangle\!\rangle = \langle\!\langle \Gamma \rangle\!\rangle \cdot z_X : (\widetilde{S}, \mu \mathrm{t}.?((\widetilde{S}, \mathrm{t}) {\to} \diamond); \mathtt{end}) {\to} \diamond$.

Session Types: The Reality

- Sequential composition not supported in types for actual PLs.
- Channel types declare payload types and channel directions, not structure.
  - In Go:

    ```
    ch := make(chan int)
    ```

  - In CloudHaskell:

    ```
    (s,r) <- newChan::Process (SendPort Int, ReceivePort Int)
    ```

- Programmers must enforce sequentiality themselves $\rightsquigarrow$ Error-prone

On Sequentiality in Processes: Trios in Concert

A beautiful result, by Parrow (1996)

- π-calculus processes **decomposed** as a collection of **trios processes** with at most 3 nested prefixes
- P and its decomposition $\mathcal{D}(P)$ are tightly related, up to weak bisimilarity:

$$P \approx \mathcal{D}(P)$$

- Untyped setting: No constraints on name usage
- Replication instead of recursion
- No higher-order communication nor choices

Bridge the gap!

Can we dispense with sequential composition in session types?

- **Yes!** Sequentiality in types can be codified by sequentiality in processes. Key inspiration from Parrow's decomposition approach.
- Only sequential composition **in processes** is truly indispensable.

Key Ideas

A process P typed with standard session types $S_1, \ldots, S_n$:

- Sequencing in $S_1, \ldots, S_n$ is codified by $\mathcal{D}(P)$, the decomposition of P.
- Each S_i is decomposed into $\mathcal{G}(S_i)$, a **list** of minimal session types.
- Roughly: If $\Gamma \vdash P$ then $\mathcal{G}(\Gamma) \vdash \mathcal{D}(P)$.

Example: Decomposing Processes

$C_{\mathtt{Pay}} \mid Q_{\mathtt{Pay}}$

$\mathcal{D}(C_{\mathtt{Pay}} \mid Q_{\mathtt{Pay}}) = \overline{c_1}!\langle\rangle \mid c_1?().\overline{c_2}!\langle\rangle.\overline{c_6}!\langle\rangle$

$C_{\mathtt{Pay}}$

$\overline{u}!\langle 36\rangle.$
$\overline{u}!\langle bank\rangle.$
$\overline{u}?(status).$
0

$\mathcal{D}(C_{\mathtt{Pay}})$

$c_6?().\overline{u_1}!\langle 36\rangle.\overline{c_7}!\langle\rangle \mid$
$c_7?().\overline{u_2}!\langle bank\rangle.\overline{c_8}!\langle\rangle \mid$
$c_8?().\overline{u_3}?(status).\overline{c_9}!\langle status\rangle \mid$
$c_9?(status).0$

$Q_{\mathtt{Pay}}$

$u?(a).$
$u?(b).$
$u!\langle a < 42\rangle.$
0

$\mathcal{D}(Q_{\mathtt{Pay}})$

$c_2?().u_1?(a).\ \overline{c_3}!\langle a\rangle \mid$
$c_3?(a).u_2?(b).\ \overline{c_4}!\langle a, b\rangle \mid$
$c_4?(a, b).u_3!\langle a < 42\rangle.\ \overline{c_5}!\langle b\rangle \mid$
$c_5?(b).0$

Example: Decomposing Session Types

$$Q_{\text{Pay}} = u?(a).u?(b).u!\langle a < 42 \rangle.0$$

$$u : ?\mathsf{Int}; ?\mathsf{Str}; !\mathsf{Bool}; \mathsf{end}$$

$$\downarrow$$

$$\mathcal{D}(Q_{\text{Pay}})$$

$$c_2?().u_1?(a).\overline{c_3}!\langle a \rangle \parallel c_3?(a).u_2?(b).\overline{c_4}!\langle a, b \rangle \parallel c_4?(a,b).u_3!\langle a < 42 \rangle.\overline{c_5}!\langle b \rangle$$

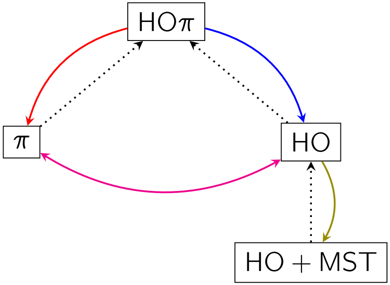
$u_1 : ?\mathsf{Int}; \mathsf{end}$
$c_2 : ?(); \mathsf{end}$

$u_2 : ?\mathsf{Str}; \mathsf{end}$
$c_3 : ?(\mathsf{Int}); \mathsf{end}$

$u_3 : !\mathsf{Bool}; \mathsf{end}$
$c_4 : ?(\mathsf{Int}, \mathsf{Bool}); \mathsf{end}$

Concluding Remarks

- Expressivity results for HO process calculi with session types
- Different calculi with functional and concurrent features, tightly connected
- Session types guide encodings, and induce strong forms of correctness
- Session types explained in terms of themselves
- More results in Inf & Comp'19 / ECOOP'19.



Session Types and Higher-Order Concurrency

Jorge A. Pérez
University of Groningen, The Netherlands
www.jperez.nl

Joint work with

Alen Arslanagić, Dimitrios Kouzapas, Nobuko Yoshida, and Erik Voogd

(based on papers in Inf & Comp'19 and ECOOP'19)

UNIFYING CORRECTNESS FOR COMMUNICATING SOFTWARE

OWLS - February 24, 2021