Session Types and Open Problems

Nobuko Yoshida
Betty Meeting 6th October 2016
The Kohei Honda Prize for Distributed Systems  Queen Mary, University of London

Posted with permission from QMUL on 17th Dec 2013. Original article written by Edmund Robinson.

This prize was instituted in 2013 and is awarded annually to one undergraduate student and one postgraduate student in recognition of their achievement in applying the highest quality scientific and engineering principles in the broad area of Distributed Systems. This is the area in which Dr Honda concentrated most of his teaching, and it is also the area in which he conducted his research. Its primary funding comes from a donation from his family, who wished to commemorate Dr Honda in this way. Additional funding has come from Dr Honda’s own ETAPS Award. This prize is sponsored by Springer Verlag, and awarded annually by the ETAPS committee in recognition of an individual’s research contribution. Dr Honda received the first such award posthumously, and the awarding panel expressed a wish that the funding be used to supplement this prize fund. The laudation for this award, written by Dr Honda’s colleague, Prof Vladimiro Sassone is included later.

About Dr Honda

Kohei Honda was born and lived the first part of his life in Japan. Like many scientists he was fascinated by the idea of finding basic explanatory theories, like the physicists looking for grand unified theories of the universe. Kohei, though, was passionately interested in finding the right basic explanatory theory for the process of computation. Most academics agree that the basic theory

Winners 2013

Ms Anna Pawlicka
2013 winner (Undergraduate) source: QMUL

Mr. Valmir Negacevshi
2013 winner (Postgraduate) source: QMUL
Programming languages are tools which offer frameworks of abstraction for such activities – promoting or limiting them

• Imperative
• Functional
• Logical
On Programs and Programming

- The most fundamental element of a PL in this context is a set of operations it is based on:
  - Imperative: assignment, jump.
  - Functional: \( \beta \)-reduction.
  - Logical: unification.

- Another element is how we can combine, or structure, these operations:
  - Imperative: sequential composition, if-then-else, while, procedures, modules, ...
  - Functional: application, product, union, recursion, modules, ...
Var $a$: array [MAX] of int;

Procedure sort ($l, r$: int);

Var $i, j, x$: int;

$i := l$; $j := r$;

$x := ([l+r] \div 2]$;  * Choose a pivot.

repeat

while $a[i] < x$ do $i := i+1$ end  \text{ Partition into two parts.}

while $a[j] > x$ do $j := j+1$ end

if $i < j$ then swap ($i, j$); $i := i+1$; $j := j-1$; end

until $i > j$

if $l < j$ then sort ($l, j$) \text{ Recursively sort two parts.}

if $l < r$ then sort ($i, r$);

end

Procedure swap ($i, j$: int)

Var $w$: int;


end
Quicksort in pure lambda:

```

```

Quicksort with combinators:

```
Y (λf. λx. (Isnil x)
(Concat (f Filter (λy. LTy (Cons (Carl))) (Cdr l)))
(Cons (Carl))
(f Filter (λy. Mey (Carl))) (Cdr l)))
```

```
I = λx.x  T = λx.(λy.x)  F = λx.(λy.y)  Y = (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))

Cons = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))))

Cons = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))))

Car = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))))

Cdr = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))))

Concat = Y (λc. λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))

Filter = Y (λc. λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

IstoBon = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))

Pred = Y (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

LT = Y (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

ME = Y (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))
```

```
Y (λf. λx. (Isnil x)
(Concat (f Filter (λy. LTy (Cons (Carl))) (Cdr l)))
(Cons (Carl))
(f Filter (λy. Mey (Carl))) (Cdr l)))
```

```
I = λx.x  T = λx.(λy.x)  F = λx.(λy.y)  Y = (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

Cons = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))))

Car = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))))

Cdr = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x)))))))))))))))))))))))))))

Concat = Y (λc. λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

Filter = Y (λc. λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

IstoBon = λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

Pred = Y (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

LT = Y (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))

ME = Y (λx.(λy.(λz.(λw.(λv.(λu.(λt.(λs.(λr.(λq.(λp.(λo.(λn.(λm.(λl.(λk.(λj.(λi.(λh.(λg.(λf.(λe.(λd.(λc.(λb.(λa.x))))))))))))))))))))))))
```
Quicksort in ML:

fun qs nil : int list = nil

| qs (x::r) = let val small =
|     filter (fn y => y < x) r
|     and large =
|     filter (fn y => y >= x) r
|     in qs small @ [x] @ qs large
|     end

fun filter p nil = nil

| filter p (x::r) =
|     if p x then x := filter p r
|     else filter p r
The Ti-calculus as a Descriptive Tool

\[ M :: = x \mid \lambda x.M \mid MN. \]

\[ P :: = \Sigma_0 P \mid \Pi Q \mid \sigma P \mid \Pi P \mid \theta. \]

with \( \Pi :: = x(\theta) \mid \theta x(\theta) \).
Examples of Representable Computation

- λ-calculus [MPW89, Milner89, Milner82, ...]
- Concurrent Object [Walker81]
- \( \omega \)-order term passing [Sagiv92]
- Various data structures [Milner92, ...]
- Proof Nets [Bellin and Scott93]
- Arbitrary ‘constant’ interaction [H95]
- Strategies on Games [HO95]
The Role of Types in TR-calculus.

- (classification) How can we classify name-passing interactive behaviours, i.e., behaviours representable in TR-calculus? What classes ("types") of behaviours can we find in the calculus?

- (safety) Is this program/system in the safe (or correct, relevant, ...) classes of behaviours? Can the safety be preserved compositionally?
Functional Types

- \( \text{Nat} \rightarrow \text{Nat} \):
  - Succ, Dec, Id,
  - ...

- \( \text{Nat} \rightarrow \text{Bool} \):
  - Iszero,
  - ...

- \( \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \):
  - And, Or, Not,
  - ...

- \( \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \):
  - Add, Sub, Mul,
  - Div, Eq?, ...

- \( \text{Nat} \rightarrow \text{Nat} \):
  - ...

- \( \text{Nat} \rightarrow \text{Nat} \):
  - ...

- \( \text{Bool} \):
  - True, False

with operation:

\[
\begin{cases}
  f : \text{Id} : \text{Nat} 
  
  e : \text{Nat} 
  
  d : \text{Nat} 

  f \cdot e \cdot d = f \cdot e : \text{Id}.

\end{cases}
\]

else undefined.
Process Types

- When it comes to processes, composition becomes:
  \[ 3 + 5 \]
  \[ \Rightarrow 8 \]

- Therefore we type processes:
  - divergence
  - deadlock
  - runtime error

- But some composition is dangerous!

  
  ![Diagram Showing Explosion]

  - The connection is prohibited.
Implementing ATM

1. Put the card in.
2. The ATM asks what you want.
   - Withdrawal?
   - Deposit?
   - Quit?
3. Select, for example, withdraw, then key in the amount.
5. Ok, take your money.
6. Overdraft! Try again.
Implementing ATM

ATMc(b) =

\[ \text{User: } \text{ATMc(b)} \]

\[ \text{Bank: } \text{ATMc(b)} \]

\[ \text{User: } \text{ATMc(b)} \]

ENCODING
Session Type Mobility Group

www.mrg.doc.ic.ac.uk
Selected Publications 2015/2016


- **[CC’16]** Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.

- **[FASE’16]** Raymond Hu, NY: Hybrid Session Verification through Endpoint API Generation.

- **[TACAS’16]** Julien Lange, NY: Characteristic Formulae for Session Types.


- **[POPL’16]** Dominic Orchard, NY: Effects as sessions, sessions as effects.

- **[FSTTCS’15]** Romain Demangeon, NY: On the Expressiveness of Multiparty Session Types.

- **[OOPSLA’15]** Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, NY: Protocol-Based Verification of Message-Passing Parallel Programs.


- **[CONCUR’15]** Laura Bocchi, Julien Lange, NY: Meeting Deadlines Together.

- **[CONCUR’15]** Marco Carbone, Fabrizio Montesi, Carsten Schürmann, NY: Multiparty Session Types as Coherence Proofs.

- **[CC’15]** Nicholas Ng, Jose G.F. Coutinho, NY: Protocols by Default: Safe MPI Code Generation based on Session Types.


- **[POPL’15]** Julien Lange, Emilio Tuosto, NY: From communicating machines to graphical choreographies.
Selected Publications 2015/2016

- **[ECOOP’16]** Alceste Scala, NY: Lightweight Session Programming in Scala
- **[CC’16]** Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
- **[FASE’16]** Raymond Hu, NY: Hybrid Session Verification through Endpoint API Generation.
- **[TACAS’16]** Julien Lange, NY: Characteristic Formulae for Session Types.
- **[POPL’16]** Dominic Orchard, NY: Effects as sessions, sessions as effects.
- **[FSTTCS’15]** Romain Demangeon, NY: On the Expressiveness of Multiparty Session Types.
- **[OOPSLA’15]** Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, NY: Protocol-Based Verification of Message-Passing Parallel Programs.
- **[CONCUR’15]** Laura Bocchi, Julien Lange, Nobuko Yoshida: Meeting Deadlines Together.
- **[CONCUR’15]** Marco Carbone, Fabrizio Montesi, Carsten Schürmann, NY: Multiparty Session Types as Coherence Proofs.
- **[CC’15]** Nicholas Ng, Jose G.F. Coutinho, NY: Protocols by Default: Safe MPI Code Generation based on Session Types.
- **[POPL’15]** Julien Lange, Emilio Tuosto, NY: From communicating machines to graphical choreographies.
• **TCS’16**: Monitoring Networks through Multiparty Session Types. Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, Nobuko Yoshida

• **LMCS’16**: Multiparty Session Actors. Rumyana Neykova, Nobuko Yoshida

• **FMSD’15**: Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, Nobuko Yoshida

• **TGC’13**: The Scribble Protocol Language. Nobuko Yoshida, Raymond Hu, Rumyana Neykova, Nicholas Ng
Session Types Overview

- Global session type
  \[ G = A \rightarrow B : \langle U_1 \rangle \cdot B \rightarrow C : \langle U_2 \rangle \cdot C \rightarrow A : \langle U_3 \rangle \]

- Local session type
  - Slice of global protocol relevant to one role
  - Mechanically derived from a global protocol
  \[ T_A = !\langle B, U_1 \rangle \cdot ?\langle C, U_3 \rangle \]

- Process language
  - Execution model of I/O actions by session participants
  - Mechanically derived from a global protocol
  \[ P_A = a[A](x) \cdot x!\langle B, u_1 \rangle \cdot x?(C, y) \]

- (Static) type checking for communication safety and progress
The Scribble Protocol Language

Scribble: adapts and extends MPST as an engineering language for describing multiparty message passing protocols

```plaintext
global protocol MyProtocol(role A, role B, role C) {
    m1(int) from A to B;
    rec X {
        choice at B {
            m2(String) from B to C;
            continue X;
        } or {
            m3() from B to C;
        }
    }
}
```

Communication Model

- asynchronous, reliable, role-to-role ordering
- Scribble sessions can be conducted over any transport that supports this model
Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data. However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

Describe 
Scribble is a language for describing multiparty protocols from a global, or endpoint neutral, perspective.

Verify 
Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that protocols described using the language are sound, and do not suffer from deadlocks or livelocks.

Project 
Endpoint projection is the term used for identifying the responsibility of a particular role (or endpoint) within a protocol.

Implement 
Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b) using session type APIs to clearly describe the behaviour, and (c) statically verify the code against the projection.

Monitor 
Use the endpoint projection for roles defined within a Scribble protocol, to monitor the activity of a particular endpoint, to ensure it correctly implements the expected behaviour.
module examples;

global protocol HelloWorld(role Me, role World) {
  hello() from Me to World;
  choice at World {
    goodMorning1() from World to Me;
  } or {
    goodMorning1() from World to Me;
  }
}
Open Problems

1. Behavioural Theories and Session Types
2. Relationship with Other Frameworks
   - Linear Logic
   - Communicating Automata
   - Petri Nets
3. Outreach
   - Industry
   - Developers
   - Education
Interactions with Industries

Strange Loop
SEPTEMBER 15-17 2016 / PEABODY OPERA HOUSE / ST. LOUIS, MO

Adam Bowen @adammbowen · Sep 15
I didn’t even know that session types existed an hour ago, but thanks to Nobuko Yoshida’s great talk at #pwlconf, I want to learn more.

Nobuko Yoshida
Imperial College, London

DoC researcher to speak at Golang UK conference
by Vicky Kapogianni
20 July 2016

DoC researcher to speak at industry-focused Golang UK conference on results of concurrency research

@nicholaschwng rocking on @GolangUKconf about static deadlock detection in #golang #gouk16

The Golang UK Conference
Interactions with Industries

**F#unctional Londoners Meetup Group**

6 days ago · 6:30 PM

Session Types with Fahd Abdeljallal

43 Members

Synopsis: Session types are a formalism to codify the structure of a communication, using types to specify the communication protocol used. This formalism provides the... [LEARN MORE]

---

**Distributed Systems vs. Compositionality**

Dr. Roland Kuhn
@rolandkuhn — CTO of Actyx

**Current State**

- behaviors can be composed both sequentially and concurrently
- effects are not yet tracked
- Scribble generator for Scala not yet there
- theoretical work at Imperial College, London (Prof. Nobuko Yoshida & Alceste Scalas)
Dynamic Monitoring

[RV’13, COORDINATION’14, FMSD’15]

Projection

Global Type

Local Type

Local Type

Local Type

Program Alice

Program Bob

Program Carol
Type Checking
[ECOOP’16, OOPSLA’15, POPL’16]

Global Type

Projection

Local Type

Type Checking
Program Alice

Local Type

Type Checking
Program Bob

Local Type

Type Checking
Program Carol
Code Generation [CC’15, FASE’16]

Global Type

Projection

Local Type

Generation

Program Alice

Local Type

Generation

Program Bob

Local Type

Generation

Program Carol
Synthesis

[ICALP’13, POPL’15, CONCUR’15, TACAS’16, CC’16]
GLOBALLY GOVERNED
SESSION SEMANTICS

Dimitrios Kouzapas
Glasgow

Nobuko Yoshida
Imperial College London
Typed Semantics in $\Pi$ 1991 →

IO-subtyping, Linear types, Secure Information Flow, ...

\[ \Gamma \vdash p \approx q \]

- Correctness of Encoding
- Limit environments $\Rightarrow$ Equate more processes
- Compositional
Multiparty Session Types

- Participants agreed with global protocols
- Many Multiparty Sessions can interleave for a single point application

with each message clearly identifiable as belonging to a specific session
Multiparty Session Bisimulations

Standard Multiparty Session Bisimulations $\approx_s$

\[
\Gamma \vdash P \triangleright \Delta
\]

Shared Env

Session channels

Env

Governed Multiparty Session Bisimulations $\approx_g$

\[
E, \Gamma \vdash P \triangleright \Delta
\]

Global Type Env

a mapping from session to global types

\[S_1: G_1, S_2: G_2, \ldots, S_n: G_n\]
Governed Bisimulations

- Compositional?
- Coincides with Contextual Equiv?
- What is a difference between $\approx_s$ and $\approx_g$?
- Under what condition $\approx_s$ and $\approx_g$ can coincide?
- Applications?
Theorem 3

- If for all $E$, $\Gamma \vdash P_1 \circ \Delta_1 \approx g P_2 \circ \Delta_2$ then $\Gamma \vdash P_1 \circ \Delta_1 \approx_s P_2 \circ \Delta_2$

- If $\Gamma \vdash P_1 \circ \Delta_1 \approx_s P_2 \circ \Delta_2$ then for all $E$, $\Gamma \vdash P_1 \circ \Delta_1 \approx g P_2 \circ \Delta_2$
Theorem 4 (coincidence) \[\text{no interleaved sessions}\]

Assume \(P_1\) and \(P_2\) are simple. If there exists \(E\), \(\Gamma \vdash P_1 \circ \Delta_1 \approx_g P_2 \circ \Delta_2\), then \(\Gamma \vdash P_1 \circ \Delta_1 \approx_s P_2 \circ \Delta_2\).
Theorem 4 (coincidence) no interleaved sessions

Assume $P_1$ and $P_2$ are simple. If there exists

\[ E, \Gamma \vdash P_1 \circ \Delta_1 \approx_{s} P_2 \circ \Delta_2, \text{ then } \]

\[ \Gamma \vdash P_1 \circ \Delta_1 \approx_{s} P_2 \circ \Delta_2 \]
Bridging functions and concurrency

- First 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
- Several other results in the paper


**CONCUR’15**: Characteristic Bisimulations for Higher-Order Session Processes. Dimitrios Kouzapas, Jorge A. Pérez, Nobuko Yoshida
Multiparty Compatibility in Communicating Automata

Synthesis and Characterisation of Multiparty Session Types

Nobuko Yoshida

Pierre-Malo Denielou

ICALP'13
mut! Title; ? Quote; ![ok: ! Add; ? Date, retry: t ]
mut? Title; ! Quote; ? ![ok: ? Add; ! Date, retry: t ]
Communicating Automata [1980s]
1. Deterministic
2. No-Mixed State
3. Compatible

[Gouda et al 1986] Two compatible machines without mixed states which are deterministic satisfy deadlock-freedom.
From Communicating Machines to Graphical Choreographies [POPL’15, CONCUR’15]

[ESOP’10,ESOP’12,CONCUR’12,CONCUR’14]
Contributions

- Static deadlock detection tool *dingo-hunter*
- Deadlock detection based on session types
- **Infer** session types as Communicating Automata
- **Synthesise** global session graphs from CA
Go and Concurrency

- Developed by Google for multi-core programming
- Concurrency model built on CSP (process calculi)
- Message-passing **communication** over channels

"Do not communicate by sharing memory; instead, share memory by communicating."
-- Effective Go (developer guide)
Java API Generation [FASE’16]
Session Type Mobility Group

www.mrg.doc.ic.ac.uk