Synchronous Multiparty Session Types

Andi Bejleri Nobuko Yoshida
Imperial College London

Aim of this work

The aim of this work is to study and implement a formal model where communication protocols are abstracted at type-level, and a static type system that checks if programs implement a coherent and loyal protocol with respect to the abstracted one.

The study of the formal model will be based on a variant of the π-Calculus and its implementation will be based on an extension of Java.

Communication between processes

- A feature that is widely used in: Web Services, Distributed Algorithms, Business Processes and Multi-core Programming.

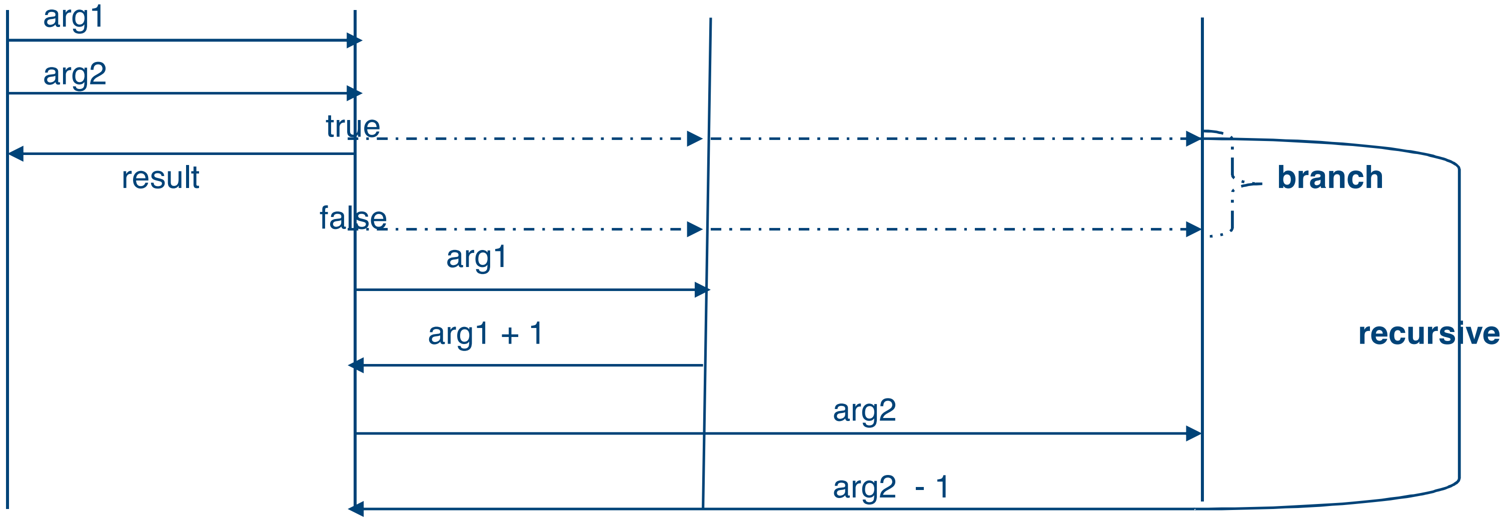
- Programs of these systems implement a communication protocol: order of sending and receiving messages.

Binary session types (PARLE 94)

- A **session** is the interactive structure formed by the sending-receiving primitives and the conditional-iteration/recursion constructs.
- An abstraction at type-level of the session between two processes.
- Type system checks if the inferred binary session type are reciprocal with each other.

An arithmetic problem: Addition of two naturals

- In a context where addition, successor and predecessor are defined as processes; arguments and results are passed via communications.
- Client Addition Successor Predecessor



Representing the solution in binary session types

- Client-Addition: sending of the two arguments and result

  !<int>;!<int>;?<int>;**end**   ?<int>;?<int>;!<int>;**end**

- Addition-Successor: successor of the first argument

  μ**t**.©{**true**: **end**, **false**: !<int>;?<int>;**t**}

  μ**t**.&{**true**: **end**, **false**: ?<int>;!{int};**t**}

- Addition-Predecessor: predecessor of the second argument

  μ**t**.©{**true**: **end**, **false**: !<int>;?<int>;**t**}

  μ**t**.&{**true**: **end**, **false**: ?<int>;!{int};**t**}

Programming problem: Order of communications

- The three binary session types are separate between each other.
- Order of communications are not captured by binary session types.
  - The behavior of Client-Addition depends on the behavior of Successor-Addition and Predecessor-Addition.
  - Evident when the second argument is not 0
  - Addition has to perform addition in Successor-Addition and Predecessor-Addition before returning the result to Client.

Solution: Global types

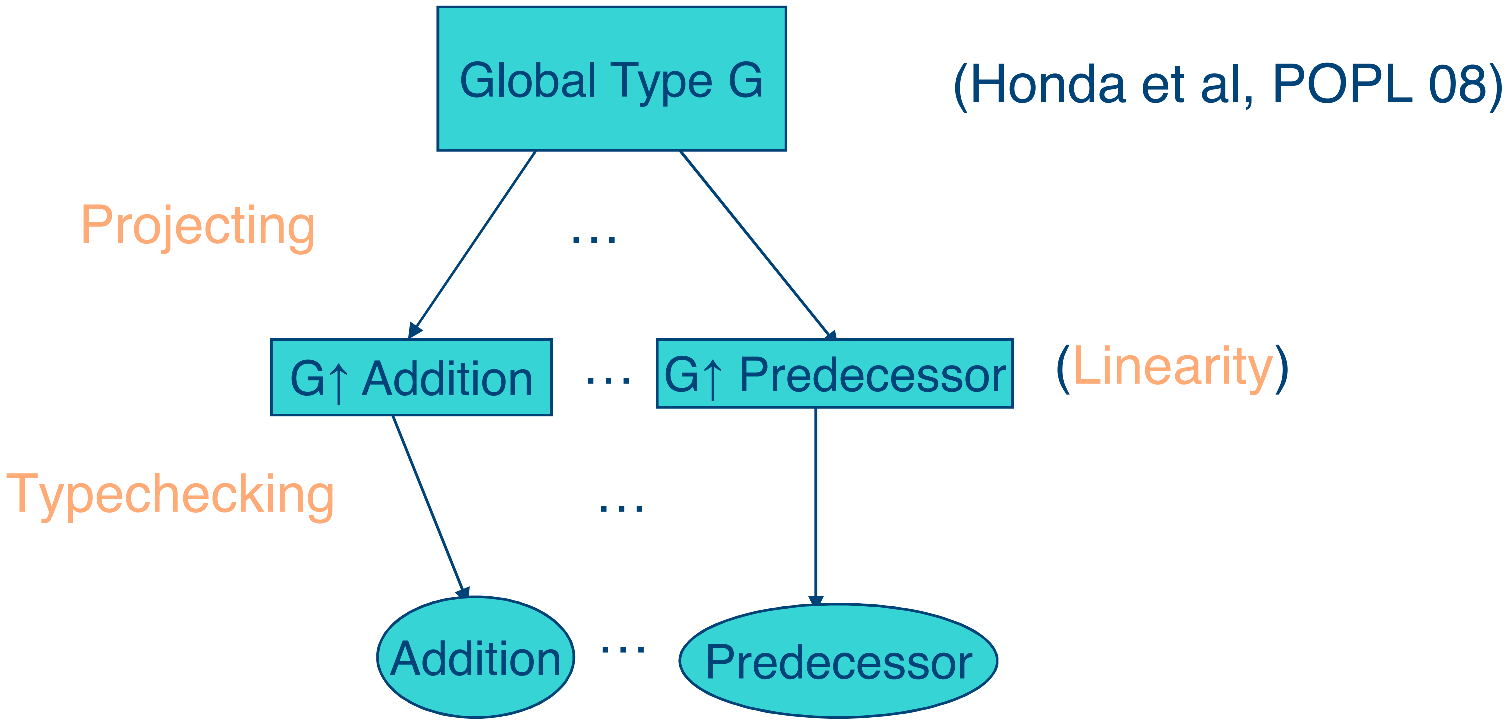
- A syntax of arrows, names and types
- A global perspective of the protocol

G = Client → Addition :<int>.
Client → Addition : <int>.
μ**t**. Addition → Successor, Predecessor : {
true: Addition → Client :<int>.**end**,
false: Addition → Successor : <int>.
Successor → Addition : <int>.
Addition → Predecessor : <int>.
Predecessor → Addition : <int>.**t**}

Channels are omitted for simplicity: Client-Add : r, Add-Succ: s, Add-Pred:t

Not only as a blueprint but also …

As a typing discipline.



Contributions

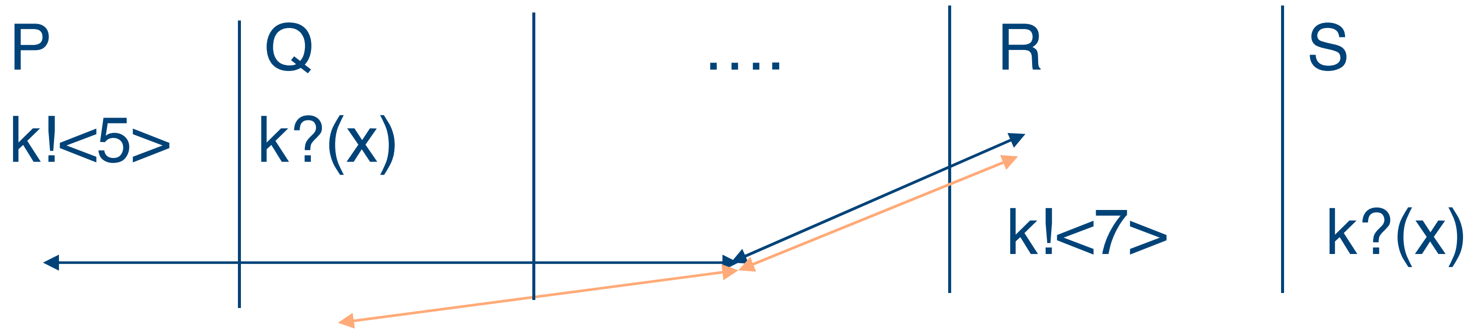
- Synchronous Communications model control for timing events and strong sequentiality order of messages.
- A simpler calculus
- Higher-order communication as k!(k')|k?(x)
- Multicasting (Addition ⟶ Successor, Predecessor)
- Global Type + Linearity
- Type Preservation

Linearity checking for global types

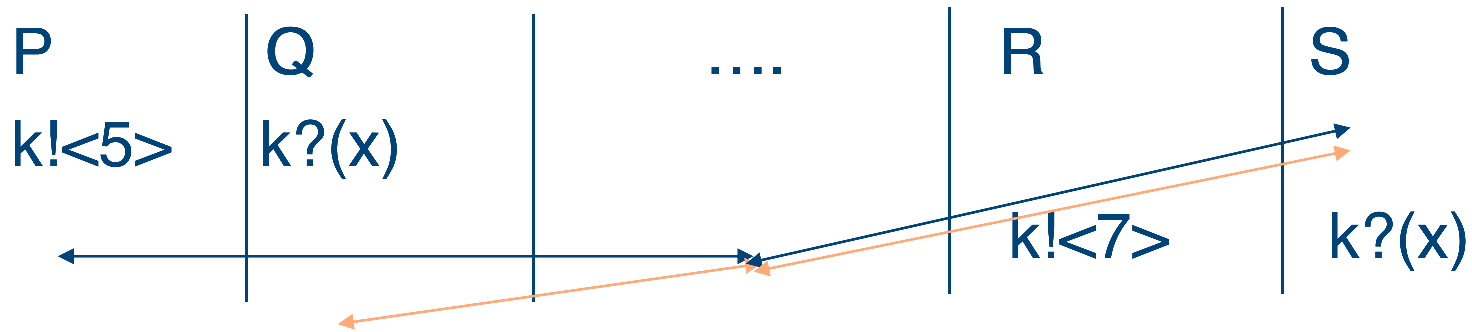
- In a context where channels are shared among different communications
- E.g. part of local memory
- The typing discipline (global types + projection) does not guarantee that the communications at run-time are the same as those specified in the global behavior
- Ambiguity: k!<5>;k!<4> | k?(x) | k?(y)
- Errors: Broken Invariants

Linearity on synchronous communications 1/3

P	Q	….	R	S
k!<5>	k?(x)			
			k!<7>	k?(x)



Linearity on synchronous communications 2/3



$P \longrightarrow Q : k$ $R \longrightarrow Q : l$ (II)

$P \longrightarrow Q : k$ $Q \longrightarrow R : l$ (IO)

$Q \longrightarrow P : k$ $R \longrightarrow Q : l$ (OI)

$Q \longrightarrow R : k$ $Q \longrightarrow P : l$ (OO)

$P \longrightarrow Q : k$ $P \longrightarrow Q : k$ (OO,II)

$P \longrightarrow Q : k$ $Q \longrightarrow P : k$ (IO,OI)

Linearity on synchronous communications 3/3

- G is *linear* if, whenever n1 = p1 ⟶ p1':m n2 = p2 ⟶ p2':m are in G for some m and do not occur in different branches of a branching, then a chain (φ1, ..., φn) such that n1 <_ φ1 ...<_ φn n2 exists such that the two conditions are satisfied

  1) pj ⟶ p2: m' or p2 ⟶ pj: m' is a node of the chain

  2) pk ⟶ p2' :m'' or p2' ⟶ pk:m'' is a node of the chain.

In case of multicasting (Addition ⟶ Successor, Predecessor :m), all the chains obtained by distributing each prefix of multicasting on the rest of G have to be checked if they satisfy the above conditions. If G carries other global types, we inductively demand the same.}

Global types syntax

G ::= p ⟶ p1, ..., pj: m1, ..., mj<S1, …, Sp>.G' forall i, k \in {1, ..., j}.mi ≠mk *values*
p ⟶ p":m<T@q>.G' *values*
p ⟶ p1, ..., pj: m1, ..., mj{lh:Gh}{h in J} forall i, k in {1, ..., j}.mi ≠ mk *branching*
G,G' *parallel*
μt.G *recursion*
t *variable*
end

- U ::= S1,…, Sn l T@p Value
- S ::= bool l nat l ...l <G> Sort
- m ::= 1 l 2 l …

Invisibility of delegation at global types

- Global Types should define only the interactions of that session.

G = Client → Addition: r<int>.
  Client → Addition: r<int>.
  μ**t**. Addition → Successor, Predecessor : s, t{
    true: Addition → Client :r<int>.**end**,
    false: Addition → Successor :s <int>.
      Successor → Addition : s<int>.
      Addition → Predecessor : t<int>.
      Predecessor → Addition : t<int>.**t**}

G'= Client → P: d<r?<int>@Client>

Conclusion

- Multiparty Session Types: Global Types + Local Types (Types obtained by projection)
- Global Types: Intuitive syntax + global perspective = Clear view of the protocol
- Typing discipline: global type + linearity + projection + type-checking = Duality (reciprocity) + Fidelity (loyalty)

Related work

- Asynchronous Multiparty Session Types: Honda et al. (POPL 2008)
- Scribble at http://pi4scribble.wiki.sourceforge.net/