Asynchronous Multiparty Sessions with Internal Delegation

Mariangiola Dezani-Ciancaglini

joint work with

Franco Barbanera

IFIP WG 2.2 Aachen 24-26/9/2025

Scenario

concurrent processes interacting via message-passing

Context

Alice, Bob, and Charlie want to collaborate on the net

Context

They do it by exchanging some messages

Alice, Bob, and Charlie want to collaborate on the net

Context

```
send "hello" to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

They do it by exchanging some messages

Alice, Bob, and Charlie want to collaborate on the net

Context

```
send "hello" to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

Several potential problems

Context

```
send "hello" to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

Several potential problems

- **Communication errors**

Context

```
send "hello" to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

A string is sent but a Boolean is expected

Several potential problems

- **Communication errors**

Context

```
send "hello" to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ok to Bob }
```

A string is sent but a Boolean is expected

Several potential problems

- **Communication errors**

Context

```
send  true   to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ok to Bob }
```

Several potential problems

- **Communication errors**

Context

```
send true to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
send true to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

A message is sent but there is no corresponding reception

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
send true to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

A message is sent but there is no corresponding reception

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
send true to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
receive ok from Charlie
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
send true to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
receive ok from Charlie
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

There may be deadlocks

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
send true to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Alice;
receive ok from Charlie
```

```
receive x from Alice
if x then {
   send ok to Bob;
   send ok to Alice }
else {
   send ok to Alice;
   send ok to Bob }
```

There may be deadlocks

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
send true to Charlie;
receive ok from Charlie;
send ok to Bob
```

```
receive ok from Charlie;
receive ok from Alice
```

```
receive x from Alice
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ok to Bob }
```

Several potential problems

- Communication errors
- **Protocol errors**

Context

There may be starvation

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
repeat
  send false to Charlie;
  receive y from Charlie;
until y; send ok to Bob
```

```
repeat
  receive x from Alice;
  send x to Alice;
until x;
send ok to Bob
```

```
receive ok from Charlie;
receive ok from Alice
```

There may be starvation

Here Bob starves

Several potential problems

- Communication errors
- **Protocol errors**

Context

```
repeat
  send false to Charlie;
  receive y from Charlie;
until y; send ok to Bob
```

```
repeat
  receive x from Alice;
  send x to Alice;
until x;
send ok to Bob
```

```
receive ok from Charlie;
receive ok from Alice
```

These problems may be due to:

Several potential problems

- Communication errors
- Protocol errors

Context

```
repeat
  send false to Charlie;
  receive y from Charlie;
until y; send ok to Bob
```

```
repeat
  receive x from Alice;
  send x to Alice;
until x;
send ok to Bob
```

```
receive ok from Charlie;
receive ok from Alice
```

These problems may be due to:

- **Programming errors**

Several potential problems

- Communication errors
- Protocol errors

Context

```
repeat
  send false to Charlie;
  receive y from Charlie;
until y; send ok to Bob
```

```
repeat
  receive x from Alice;
  send x to Alice;
until x;
send ok to Bob
```

```
receive ok from Charlie;
receive ok from Alice
```

These problems may be due to:

- **Programming errors**
- **Software evolution**

Several potential problems

- Communication errors
- Protocol errors

Context

```
repeat
  send false to Charlie;
  receive y from Charlie;
until y; send ok to Bob
```

```
repeat
  receive x from Alice;
  send x to Alice;
until x;
send ok to Bob
```

```
receive ok from Charlie;
receive ok from Alice
```

These problems may be due to:

- **Programming errors**
- **Software evolution**
- **Rogue participants**

Several potential problems

- Communication errors
- Protocol errors

Global specifications

Global specifications

Global specifications

Global specifications

- Do not describe (just) the behaviour of each single participant

Global specifications

Global specifications

- Do not describe (just) the behaviour of each single participant
- Describe the abstract global behaviour of the protocol

Global specifications

Global specifications

- Do not describe (just) the behaviour of each single participant
- Describe the abstract global behaviour of the protocol
- Match against/Extract the behaviours of the participants.

Global specifications

Global specifications

- Do not describe (just) the behaviour of each single participant
- Describe the abstract global behaviour of the protocol
- Match against/Extract the behaviours of the participants.

The global specification is compact and synthetic

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
        or
receive ko from Charlie
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
        or
receive ko from Charlie
```

Example of global description

```
Alice sends a Boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive ok from Charlie
        or
receive ko from Charlie
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

Example of global description

```
Alice sends a Boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

An abstract description of this protocol

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
    or
receive ko from Charlie
```

Example of global description

It abstracts values

```
Alice sends a Bool to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive ok from Charlie
or
receive ko from Charlie
```

An abstract description of this protocol

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

Example of global description

```
Alice sends a Boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

It abstracts choices

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
        or
receive ko from Charlie
```

Alice sends a Boolean

```
Alice sends a Boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
        or
receive ko from Charlie
```

Alice sends a Boolean

Charlie receives a Boolean

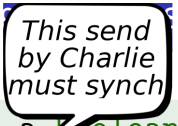
```
Alice sends a Boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
        or
receive ko from Charlie
```




```
Alice sends a boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
        or
receive ko from Charlie
```

This send by Charlie must synch

... with this reception by Bob

```
Alice sends a boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Global specifications

```
send true to Charlie;
receive ok from Charlie;
```

```
receive x from Alice;
if x then {
    send ok to Bob;
    send ok to Alice }
else {
    send ok to Alice;
    send ko to Bob }
```

```
receive ok from Charlie
        or
receive ko from Charlie
```

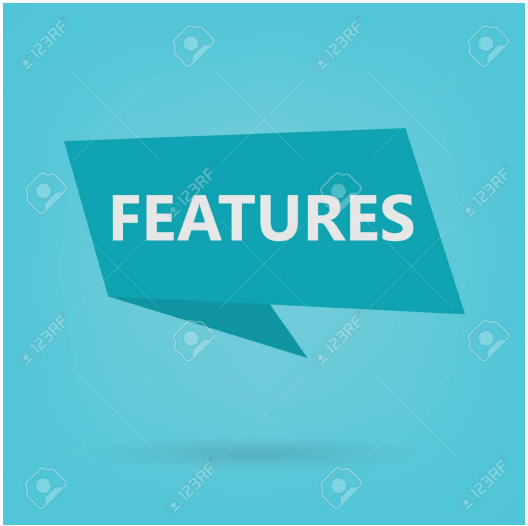
This send by Charlie must synch

... with this reception by Bob

... and not with this one!

```
Alice sends a boolean to Charlie;
either Charlie sends ok to Bob; Charlie sends ok to Alice;
    or Charlie sends ok to Alice; Charlie sends ko to Bob;
```

Features



Description/Verification of concurrent systems

Global description

faithful/property-preserving

Implementation

Asynchronous Communication

Asynchronous Communication

a queue is needed

Asynchronous Communication

a queue is needed

Alice puts her message for Baloo in the queue

Asynchronous Communication

a queue is needed

Alice puts her message for Baloo in the queue

Baloo takes Alice message from the queue

Delegation

Alice, Cheshire Cat and Bank Example

Alice, Cheshire Cat and Bank Example

title →

Alice, Cheshire Cat and Bank Example

title →

← price

Alice, Cheshire Cat and Bank Example

Alice, Cheshire Cat and Bank Example

title

price

ok

card

Alice, Cheshire Cat and Bank Example

title →

← price

ok →

card →

← date

Alice, Cheshire Cat and Bank Example

title

price

ok

card

date

Trust assumption:

Cat does not have authority to handle cards

Alice, Cheshire Cat and Bank Example

title

price

ok

Alice, Cheshire Cat and Bank Example

title

price

ok

price

Alice, Cheshire Cat and Bank Example

title

price

ok

price

forward delegation

Alice, Cheshire Cat and Bank Example

title

price

ok

price

forward delegation

card

the card number sent to Cat goes directly to Bank

Alice, Cheshire Cat and Bank Example

title

price

ok

price

forward delegation

card

Alice, Cheshire Cat and Bank Example

title

price

ok

price

forward delegation

card

backward delegation

Alice, Cheshire Cat and Bank Example

title

price

ok

price

forward delegation

card

backward delegation

date

Asynchronous Delegation

The Cat puts its forward delegation for the Bank in the queue and becomes the frozen Bank

Asynchronous Delegation

The Cat puts its forward delegation for the Bank in the queue and becomes the frozen Bank

The Bank takes Cat forward delegation from the queue and becomes the Cat

Asynchronous Delegation

The Cat puts its forward delegation for the Bank in the queue and becomes the frozen Bank

The Bank takes Cat forward delegation from the queue and becomes the Cat

• • •

Asynchronous Delegation

The Cat puts its forward delegation for the Bank in the queue and becomes the frozen Bank

The Bank takes Cat forward delegation from the queue and becomes the Cat

• • •

The Bank masked as Cat puts its backward delegation for the Cat masked as Bank in the queue and becomes back the Bank

Asynchronous Delegation

The Cat puts its forward delegation for the Bank in the queue and becomes the frozen Bank

The Bank takes Cat forward delegation from the queue and becomes the Cat

• • •

The Bank masked as Cat puts its backward delegation for the Cat masked as Bank in the queue and becomes back the Bank

The Cat masked as frozen Bank takes the backward delegation from the queue and becomes back the Cat

Safety


PASSWORD
CRACKER
SPYWARE
CYBER
CODE
ENCRYPTION
TROJAN
SECURITY
IDENTITY
HACKER
THEFT
PHISHING
PRIVACY
VIRUS
INTRUSION

Safety for Asynchronous Calculi

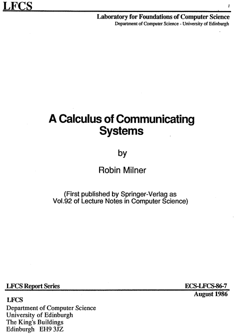
lock freedom: no participant requiring to take a message waits forever

Safety for Asynchronous Calculi

lock freedom: no participant requiring to take a message waits forever

orphan-message freedom: no message stays forever in the queue

Background


LFCS
Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

A Calculus of Communicating Systems

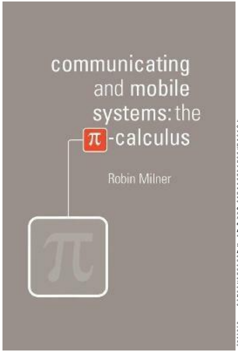
by

Robin Milner

(First published by Springer-Verlag as Vol.92 of Lecture Notes in Computer Science)

LFCS Report Series ECS-LFCS-86-7
August 1986

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ




Multiparty Asynchronous Session Types

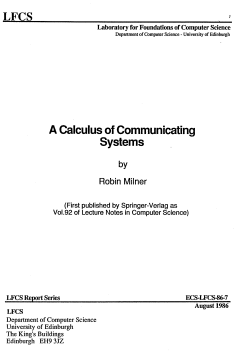
Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

Marco Carbone
Queen Mary, University of London
carbonem@dcs.qmul.ac.uk

Communicating Processes

CCS π-calculus


LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

A Calculus of Communicating Systems

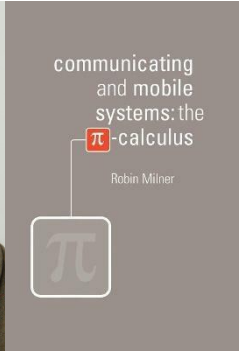
by

Robin Milner

(First published by Springer-Verlag as Vol.92 of Lecture Notes in Computer Science)

LFCS Report Series ECS-LFCS-86-7
August 1986

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ



Communicating Processes

Multiparty Sessions

Multiparty Asynchronous Session Types

Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

Marco Carbone
Queen Mary, University of London
carbonem@dcs.qmul.ac.uk

Abstract

Communication is becoming one of the central elements in software development. As a potential typed foundation for structured communication-centred programming, session types have been studied over the last decade for a wide range of process calculi and programming languages, focussing on binary (two-party) sessions. This work extends the foregoing theories of binary session types to multiparty, asynchronous sessions, which often arise in practical communication-centred applications. Presented as a typed calculus for mobile processes, the theory introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. Global types retain a friendly type syntax of binary session types while capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficient type checking through its projection onto individual peers. The fundamental properties of the session type discipline such as communication safety, progress and session fidelity are established for general *n*-party asynchronous interactions.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Semantics of Programming Languages*]: Process models

General Terms Theory, Types, Design

Keywords communications, multiparty, structured programming, session types, mobile processes, causality, choreography

1. Introduction

Backgrounds Communication is becoming one of the central elements in software development, ranging from web services to business protocols to parallel scientific computing to multicore programming. As a potential typed foundation for structured communication-centred programming, session types have been studied in many contexts over the last decade, including calculi of mobile processes (Takeuchi et al. 1994; Gay and Hole 2005; Honda et al. 1998; Bonelli and Compagnoni 2008), higher-order processes (Mostrous and Yoshida 2007), Ambients (Garralda et al. 2006), multi-threaded ML (Vasconcelos et al. 2006), Haskell (Neubauer and Thiemann 2004b), F# (Corin et al. 2007), operating systems (Fähndrich et al. 2006), Java (Dezani-Ciancaglini et al. 2006; Coppo et al. 2007; Hu et al. 2007), and Web Services (Carbone et al. 2006, 2007; WS-CDL; Sparkes 2006; Honda et al. 2007a). A basic observation underlying session types is that a communication-centred application often exhibits a highly structured sequence of interactions involving, for example, branching and recursion, which as a whole form a natural unit of conversation, or *session*. The structure of a conversation is abstracted as a type through an intuitive syntax, which is then used as a basis of validating programs through an associated type discipline.

As an example, the following session type describes a simple business protocol between Buyer and Seller from Buyer's viewpoint: Buyer sends the title of a book (a string), Seller sends a quote (an integer). If Buyer is satisfied by the quote, then sends his address (a string) and Seller sends back the delivery date (a date); otherwise it quits the conversation.

$$!\mathsf{string};?\mathsf{int};\oplus\{\mathsf{ok}:!\mathsf{string};?\mathsf{date};\mathsf{end},\quad \mathsf{quit}:\mathsf{end}\} \qquad (1)$$

Above !t denotes an output of a value of type t, dually for ?t; ⊕ denotes a choice of the options; and end represents the termination of the conversation.

Such explicit representation of conversation structures helps us deal with one of the most common bugs in programming with communication, the synchronisation bugs. A programmer expects that communicating programs should together realise a consistent conversation, but they easily fail to handle a specific incoming message or to send a message at the correct timing, with no way to detect such errors before runtime. An explicit specification as in (1) guides principled programming of communication behaviour and enables automatic protocol validation (WS-CDL; UNIFI; Hu et al. 2007). In addition, a clean separation between abstraction and implementation given by type-based abstraction and associated primitives leads to intelligible programs and flexible implementations (Hu et al. 2007). Underlying these merits are the following central properties guaranteed by session types.

1. Interactions within a session never incur a communication error (communication safety).
2. Channels for a session are used linearly (linearity) and are deadlock-free in a single session (progress).
3. The communication sequence in a session follows the scenario declared in the session type (session fidelity, predictability).

Multiparty Asynchronous Sessions The foregoing studies on session types have focussed on binary (two-party) sessions. While many conversation patterns can be captured through a composition of binary sessions, there are cases where binary session types are not powerful enough for describing and validating interactions which involve more than two parties.

As an example, let us consider a simple refinement of the above Buyer-Seller protocol: consider two buyers, Buyer1 and Buyer2, wish to buy an expensive book from Seller by combining their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how

POPL'08, January 7–12, 2008, San Francisco, California, USA.



Communicating Processes

processes communicate by means of channels

Communicating Processes

processes communicate by means of channels

delegation is realised by the synchronous sending of a channel over another channel from the principal to the deputy

Choreographic Typing

Robin Milner (2002) :

“Types are the leaven of computer programming; they make it digestible.”

Choreographic Typing

Local and Global Types

Multiparty Asynchronous Session Types

Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Nobuko Yoshida
Imperial College London
yoshida@doc.ic.ac.uk

Marco Carbone
Queen Mary, University of London
carbonem@dcs.qmul.ac.uk

Abstract

Communication is becoming one of the central elements in software development. As a potential typed foundation for structured communication-centred programming, session types have been studied over the last decade for a wide range of process calculi and programming languages, focussing on binary (two-party) sessions. This work extends the foregoing theories of binary session types to multiparty, asynchronous sessions, which often arise in practical communication-centred applications. Presented as a typed calculus for mobile processes, the theory introduces a new notion of types in which interactions involving multiple peers are directly abstracted as a global scenario. Global types retain a friendly type syntax of binary session types while capturing complex causal chains of multiparty asynchronous interactions. A global type plays the role of a shared agreement among communication peers, and is used as a basis of efficient type checking through its projection onto individual peers. The fundamental properties of the session type discipline such as communication safety, progress and session fidelity are established for general n-party asynchronous interactions.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Semantics of Programming Languages*]: Process models

General Terms Theory, Types, Design

Keywords communications, multiparty, structured programming, session types, mobile processes, causality, choreography

1. Introduction

Backgrounds Communication is becoming one of the central elements in software development, ranging from web services to business protocols to parallel scientific computing to multicore programming. As a potential typed foundation for structured communication-centred programming, session types have been studied in many contexts over the last decade, including calculi of mobile processes (Takeuchi et al. 1994; Gay and Hole 2005; Honda et al. 1998; Bonelli and Compagnoni 2008), higher-order processes (Mostrous and Yoshida 2007), Ambients (Garralda et al. 2006), multi-threaded ML (Vasconcelos et al. 2006), Haskell (Neubauer and Thiemann 2004b), F# (Corin et al. 2007), operating systems (Fähndrich et al. 2006), Java (Dezani-Ciancaglini et al. 2006; Coppo et al. 2007; Hu et al. 2007), and Web Services (Carbone et al. 2006, 2007; WS-CDL; Sparkes 2006; Honda et al. 2007a). A basic observation underlying session types is that a communication-centred application often exhibits a highly structured sequence of interactions involving, for example, branching and recursion, which as a whole form a natural unit of conversation, or *session*. The structure of a conversation is abstracted as a type through an intuitive syntax, which is then used as a basis of validating programs through an associated type discipline.

As an example, the following session type describes a simple business protocol between Buyer and Seller from Buyer's viewpoint: Buyer sends the title of a book (a string), Seller sends a quote (an integer). If Buyer is satisfied by the quote, then sends his address (a string) and Seller sends back the delivery date (a date); otherwise it quits the conversation.

$$!\mathsf{string};\,?\mathsf{int};\,\oplus\{\mathsf{ok}: !\mathsf{string};\,?\mathsf{date};\,\mathsf{end},\quad \mathsf{quit}: \mathsf{end}\} \qquad (1)$$

Above !t denotes an output of a value of type t, dually for ?t; ⊕ denotes a choice of the options; and end represents the termination of the conversation.

Such explicit representation of conversation structures helps us deal with one of the most common bugs in programming with communication, the synchronisation bugs. A programmer expects that communicating programs should together realise a consistent conversation, but they easily fail to handle a specific incoming message or to send a message at the correct timing, with no way to detect such errors before runtime. An explicit specification as in (1) guides principled programming of communication behaviour and enables automatic protocol validation (WS-CDL; UNIFI; Hu et al. 2007). In addition, a clean separation between abstraction and implementation given by type-based abstraction and associated primitives leads to intelligible programs and flexible implementations (Hu et al. 2007). Underlying these merits are the following central properties guaranteed by session types.

1. Interactions within a session never incur a communication error (communication safety).
2. Channels for a session are used linearly (linearity) and are deadlock-free in a single session (progress).
3. The communication sequence in a session follows the scenario declared in the session type (session fidelity, predictability).

Multiparty Asynchronous Sessions The foregoing studies on session types have focussed on binary (two-party) sessions. While many conversation patterns can be captured through a composition of binary sessions, there are cases where binary session types are not powerful enough for describing and validating interactions which involve more than two parties.

As an example, let us consider a simple refinement of the above Buyer-Seller protocol: consider two buyers, Buyer1 and Buyer2, wish to buy an expensive book from Seller by combining their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how

POPL'08, January 7–12, 2008, San Francisco, California, USA.

Choreographic Typing

local types describe the actions of the single participants

Choreographic Typing

local types describe the actions of the single participants

local types are the types of processes

Choreographic Typing

local types describe the actions of the single participants

local types are the types of processes

global types describe the whole conversation scenario

Choreographic Typing

local types describe the actions of the single participants

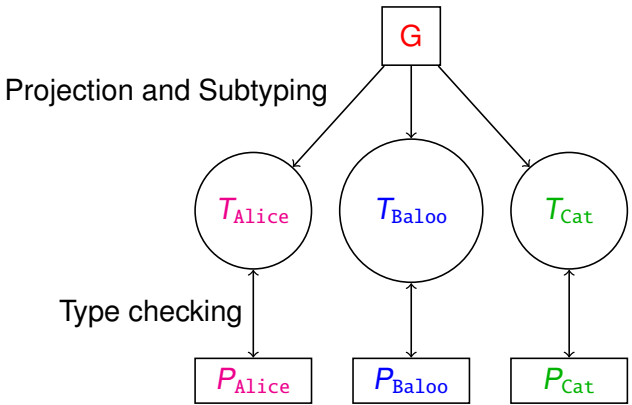
local types are the types of processes

global types describe the whole conversation scenario

global types can be projected on single participants to get their local types

Choreographic Typing

a 3-layer approach



Simple Multiparty Sessions

Precise subtyping for synchronous multiparty sessions *

Mariangiola Dezani-Ciancaglini
Università di Torino, Italy †

Silvia Ghilezan
Univerzitet u Novom Sadu, Serbia

Svetlana Jakšić
Univerzitet u Novom Sadu, Serbia

Jovanka Pantović
Univerzitet u Novom Sadu, Serbia

Nobuko Yoshida
Imperial College London ‡

The notion of subtyping has gained an important role both in theoretical and applicative domains: in lambda and concurrent calculi as well as in programming languages. The soundness and the completeness, together referred to as the preciseness of subtyping, can be considered from two different points of view: operational and denotational. The former preciseness has been recently developed with respect to type safety, i.e. the safe replacement of a term of a smaller type when a term of a bigger type is expected. The latter preciseness is based on the denotation of a type which is a mathematical object that describes the meaning of the type in accordance with the denotations of other expressions from the language. The result of this paper is the operational and denotational preciseness of the subtyping for a synchronous multiparty session calculus. The novelty of this paper is the introduction of characteristic global types to prove the operational completeness.

1 Introduction

In modelling distributed systems, where many processes interact by means of message passing, one soon realises that most interactions are meant to occur within the scope of private channels according to disciplined protocols. Following [13], we call such private interactions *multiparty sessions* and the protocols that describe them *multiparty session types*.

The ability to describe complex interaction protocols by means of a formal, simple and yet expressive type language can have a profound impact on the way distributed systems are designed and developed. This is witnessed by the fact that some important standardisation bodies for web-based business and finance protocols [1,2,12,26] have recently investigated design and implementation frameworks for specifying message exchange rules and validating business logic based on the notion of multiparty sessions, where multiparty session types are "shared agreements" between teams of programmers developing possibly large and complex distributed protocols or software systems.

Subtyping has been extensively studied as one of the most interesting issues in type theory. The correctness of subtyping relations has been usually provided as the operational soundness: If T is a subtype of T′ (notation $T \leq T'$), then a term of type T may be provided whenever a term of type T′ is needed, see [19] (Chapter 15) and [7] (Chapter 23). The converse direction, the operational completeness, has been largely ignored in spite of its usefulness to define the greatest subtyping relation ensuring type safety. If $[\![T]\!]$ is the set interpretating type T, then a subtyping is denotationally sound when $T \leq T'$ implies $[\![T]\!] \subseteq [\![T']\!]$ and denotationally complete when $[\![T]\!] \subseteq [\![T']\!]$ implies $T \leq T'$. *Preciseness* means both soundness and completeness.

* Partly supported by COST ICT201 BETTY and DART bilateral project between Italy and Serbia.
† Partly supported by MIUR PRIN Project CINA Prot. 2010LHT4KM and Torino University/Compagnia San Paolo Project SALT.
‡ Partly supported by EPSRC EP/K011715/1, EP/K034413/1, and EP/L000709X/1, and EU Project FP7-612985 UpScale.

S. J. Gay and J. Alglave (Eds.): Programming Language Approaches to Concurrency- and Communication-Centric Software (PLACES 2015).
EPTCS 203, 2016, pp. 29–35, doi:10.4204/EPTCS.203.3

Logical Methods in Computer Science
Volume 19, Issue 1, 2023, pp. 2:1–2:41
https://lmcs.episciences.org

Submitted Nov. 24, 2021
Published Jan. 13, 2023

DECONFINED GLOBAL TYPES FOR ASYNCHRONOUS SESSIONS

FRANCESCO DAGNINO [a], PAOLA GIANNINI [b], AND MARIANGIOLA DEZANI-CIANCAGLINI [c]

[a] DIBRIS, Università di Genova, Italy
e-mail address: francesco.dagnino@dibris.unige.it

[b] DISSTE, Università del Piemonte Orientale, Alessandria, Italy
e-mail address: paola.giannini@uniupo.it

[c] Dipartimento di Informatica, Università di Torino, Italy
e-mail address: dezani@di.unito.it

Abstract. *Multiparty sessions* with asynchronous communications and global types play an important role for the modelling of interaction protocols in distributed systems. In designing such calculi the aim is to enforce, by typing, good properties for all participants, maximising, at the same time, the accepted behaviours. Our type system improves the state-of-the-art by typing all asynchronous sessions and preserving the key properties of Subject Reduction, Session Fidelity and Progress when some well-formedness conditions are satisfied. The type system comes together with a sound and complete type inference algorithm. The well-formedness conditions are undecidable, but an algorithm checking an expressive restriction of them recovers the effectiveness of typing.

1. Introduction

Multiparty sessions [HYC08, HYC16] are at the core of communication-based programming, since they formalise message exchange protocols. A key choice in the modelling is synchronous versus asynchronous communications, giving rise to synchronous and asynchronous multiparty sessions. In the multiparty session approach *global types* play the fundamental role of describing the whole scenario, while the behaviour of participants is implemented by processes. A natural question is when a set of processes agrees with a global type, meaning that participants behave according to the protocol described by the global type. The straightforward answer is the design of type assignment systems relating processes and global types. Typically, global types are *projected onto* participants to get the local behaviours prescribed by the protocol and then the processes implementing the participants are checked against such local behaviours. In conceiving such systems one wants to permit all possible typings which guarantee desirable properties: the mandatory Subject Reduction, but also Session Fidelity and Progress. *Session Fidelity* [HYC08, HYC16] means that the content and the order of exchanged messages respect the prescriptions of the global type.

This work was partially funded by the MUR project "T-LADIES" (PRIN 20207LNSNX).
This original research has the financial support of the Università del Piemonte Orientale.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.46298/LMCS-19(1:2)2023

Processes

no channels, only participant names

Processes

Processes

C!title

Processes

C!title

A?title

Processes

C!title

A?title
A!price

Processes

C!title

A?title

A!price

C?price

Processes

C!title

A?title

A!price

C?price

C!ok

Processes

C!title

A?title

A!price

C?price

C!ok

A?ok

Processes



Processes

B!price

Processes

B!price

C?price

Processes

B!price

C?price

B!∈

Processes

B!price

C?price

B!∈

Processes

B!price

C?price

B!∈

C?∈

Processes


B!price

C?price

B!∈

C?∈

Processes

B!price

C?price

B!∈

C?∈

C!card

Processes


B!price

C?price

B!∈

C?∈

C!card

A?card

Processes


B!price

C?price

B!∈

C?∈

C!card

A?card

B!∋

Processes


B!price

C?price

B!∈

C?∈

C!card

A?card

B!∋

Processes


B!price

C?price

B!∈

C?∈

C!card

A?card

B!∋

C?∋

Processes


B!price

C?price

B!∈

C?∈

C!card

A?card

B!∋

C?∋

Processes

A!date

Processes

A!date

C?date

Processes


C!title

A?title

A!price

C?price

C!ok

A?ok

B!price

C?price

B!∈

C?∈

C!card

A?card

B!∋

C?∋

A!date

C?date

Sessions

parallel composition of pairs participant/process and a queue of messages

Sessions

parallel composition of pairs participant/process and a queue of messages
messages are triples sender/tag/receiver

Sessions

parallel composition of pairs participant/process and a queue of messages
messages are triples sender/tag/receiver

$$A[\![\, C!\mathrm{title}.C?\mathrm{price}.C!\mathrm{ok}.C!\mathrm{card}.C?\mathrm{date} \,]\!] \;\|$$
$$C[\![\, A?\mathrm{title}.A!\mathrm{price}.A?\mathrm{ok}.B!\mathrm{price}.B!\Subset.C?\Supset.A!\mathrm{date} \,]\!] \;\|$$
$$B[\![\, C?\mathrm{price}.C?\Subset.A?\mathrm{card}.B!\Supset \,]\!] \;\| \;\emptyset$$

Sessions

parallel composition of pairs participant/process and a queue of messages
messages are triples sender/tag/receiver

$$A[\![\, C!\mathrm{title}.C?\mathrm{price}.C!\mathrm{ok}.C!\mathrm{card}.C?\mathrm{date} \,]\!] \;\|$$
$$C[\![\, A?\mathrm{title}.A!\mathrm{price}.A?\mathrm{ok}.B!\mathrm{price}.B!{\Subset}.C?{\ni}.A!\mathrm{date} \,]\!] \;\|$$
$$B[\![\, C?\mathrm{price}.C?{\Subset}.A?\mathrm{card}.B!{\ni} \,]\!] \;\| \;\emptyset$$
$$\downarrow_{AC!\mathrm{title}}$$
$$A[\![\, C?\mathrm{price}.C!\mathrm{ok}.C!\mathrm{card}.C?\mathrm{date} \,]\!] \;\|$$
$$C[\![\, A?\mathrm{title}.A!\mathrm{price}.A?\mathrm{ok}.B!\mathrm{price}.B!{\Subset}.C?{\ni}.A!\mathrm{date} \,]\!] \;\|$$
$$B[\![\, C?\mathrm{price}.C?{\Subset}.A?\mathrm{card}.B!{\ni} \,]\!] \;\| \;\langle A, \mathrm{title}, C \rangle$$

Sessions

parallel composition of pairs participant/process and a queue of messages
messages are triples sender/tag/receiver

$$
\begin{array}{c}
A[\![\, C!\mathsf{title}.C?\mathsf{price}.C!\mathsf{ok}.C!\mathsf{card}.C?\mathsf{date} \,]\!] \parallel \\
C[\![\, A?\mathsf{title}.A!\mathsf{price}.A?\mathsf{ok}.B!\mathsf{price}.B!\Subset.C?\ni.A!\mathsf{date} \,]\!] \parallel \\
B[\![\, C?\mathsf{price}.C?\Subset.A?\mathsf{card}.B!\ni \,]\!] \parallel \emptyset \\
\downarrow_{AC!\mathsf{title}} \\
A[\![\, C?\mathsf{price}.C!\mathsf{ok}.C!\mathsf{card}.C?\mathsf{date} \,]\!] \parallel \\
C[\![\, A?\mathsf{title}.A!\mathsf{price}.A?\mathsf{ok}.B!\mathsf{price}.B!\Subset.C?\ni.A!\mathsf{date} \,]\!] \parallel \\
B[\![\, C?\mathsf{price}.C?\Subset.A?\mathsf{card}.B!\ni \,]\!] \parallel \langle A, \mathsf{title}, C \rangle \\
\downarrow_{CA?\mathsf{title}} \\
A[\![\, C?\mathsf{price}.C!\mathsf{ok}.C!\mathsf{card}.C?\mathsf{date} \,]\!] \parallel \\
C[\![\, A!\mathsf{price}.A?\mathsf{ok}.B!\mathsf{price}.B!\Subset.C?\ni.A!\mathsf{date} \,]\!] \parallel \\
B[\![\, C?\mathsf{price}.C?\Subset.A?\mathsf{card}.B!\ni \,]\!] \parallel \emptyset
\end{array}
$$

Sessions

$$
\begin{array}{c}
\mathtt{A}[\![\, \mathtt{C?price}.\mathtt{C!ok}.\mathtt{C!card}.\mathtt{C?date} \,]\!] \parallel \\
\mathtt{C}[\![\, \mathtt{A!price}.\mathtt{A?ok}.\mathtt{B!price}.\mathtt{B!}\Subset.\mathtt{C?}\Supset.\mathtt{A!date} \,]\!] \parallel \\
\mathtt{B}[\![\, \mathtt{C?price}.\mathtt{C?}\Subset.\mathtt{A?card}.\mathtt{B!}\Supset \,]\!] \parallel \emptyset \\
\downarrow^{*} \\
\mathtt{A}[\![\, \mathtt{C!card}.\mathtt{C?date} \,]\!] \parallel \mathtt{C}[\![\, \mathtt{B!}\Subset.\mathtt{C?}\Supset.\mathtt{A!date} \,]\!] \parallel \mathtt{B}[\![\, \mathtt{C?}\Subset.\mathtt{A?card}.\mathtt{B!}\Supset \,]\!] \parallel \emptyset
\end{array}
$$

Sessions

$$\mathsf{A}[\![\, \mathsf{C}?\mathsf{price}.\mathsf{C}!\mathsf{ok}.\mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date} \,]\!] \parallel \mathsf{C}[\![\, \mathsf{A}!\mathsf{price}.\mathsf{A}?\mathsf{ok}.\mathsf{B}!\mathsf{price}.\mathsf{B}!\Subset.\mathsf{C}?\Supset.\mathsf{A}!\mathsf{date} \,]\!] \parallel \mathsf{B}[\![\, \mathsf{C}?\mathsf{price}.\mathsf{C}?\Subset.\mathsf{A}?\mathsf{card}.\mathsf{B}!\Supset \,]\!] \parallel \emptyset$$

$$\downarrow^{*}$$

$$\mathsf{A}[\![\, \mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date} \,]\!] \parallel \mathsf{C}[\![\, \mathsf{B}!\Subset.\mathsf{C}?\Supset.\mathsf{A}!\mathsf{date} \,]\!] \parallel \mathsf{B}[\![\, \mathsf{C}?\Subset.\mathsf{A}?\mathsf{card}.\mathsf{B}!\Supset \,]\!] \parallel \emptyset$$

$$\downarrow_{\mathsf{CB}!\Subset}$$

$$\mathsf{A}[\![\, \mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date} \,]\!] \parallel \overset{\ast}{\mathsf{B}}[\![\, \mathsf{C}?\Supset.\mathsf{A}!\mathsf{date} \,]\!] \parallel \mathsf{B}[\![\, \mathsf{C}?\Subset.\mathsf{A}?\mathsf{card}.\mathsf{B}!\Supset \,]\!] \parallel \langle \mathsf{C}, \Subset, \mathsf{B} \rangle$$

sending the forward delegation the principal becomes the frozen deputy

Sessions

$$\mathsf{A}[\![\, \mathsf{C}?\mathsf{price}.\mathsf{C}!\mathsf{ok}.\mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date} \,]\!] \parallel \mathsf{C}[\![\, \mathsf{A}!\mathsf{price}.\mathsf{A}?\mathsf{ok}.\mathsf{B}!\mathsf{price}.\mathsf{B}!\Subset.\mathsf{C}?\ni.\mathsf{A}!\mathsf{date} \,]\!] \parallel \mathsf{B}[\![\, \mathsf{C}?\mathsf{price}.\mathsf{C}?\Subset.\mathsf{A}?\mathsf{card}.\mathsf{B}!\ni \,]\!] \parallel \emptyset$$

$$\downarrow^{*}$$

$$\mathsf{A}[\![\, \mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date} \,]\!] \parallel \mathsf{C}[\![\, \mathsf{B}!\Subset.\mathsf{C}?\ni.\mathsf{A}!\mathsf{date} \,]\!] \parallel \mathsf{B}[\![\, \mathsf{C}?\Subset.\mathsf{A}?\mathsf{card}.\mathsf{B}!\ni \,]\!] \parallel \emptyset$$

$$\downarrow_{\mathsf{CB}!\Subset}$$

$$\mathsf{A}[\![\, \mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date} \,]\!] \parallel \overset{\ast}{\mathsf{B}}[\![\, \mathsf{C}?\ni.\mathsf{A}!\mathsf{date} \,]\!] \parallel \mathsf{B}[\![\, \mathsf{C}?\Subset.\mathsf{A}?\mathsf{card}.\mathsf{B}!\ni \,]\!] \parallel \langle \mathsf{C}, \Subset, \mathsf{B} \rangle$$

$$\downarrow_{\mathsf{BC}?\Subset}$$

$$\mathsf{A}[\![\, \mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date} \,]\!] \parallel \overset{\ast}{\mathsf{B}}[\![\, \mathsf{C}?\ni.\mathsf{A}!\mathsf{date} \,]\!] \parallel \mathsf{C}[\![\, \mathsf{A}?\mathsf{card}.\mathsf{B}!\ni \,]\!] \parallel \emptyset$$

receiving the forward delegation the deputy becomes the principal

Sessions

$$
\begin{array}{c}
\mathrm{A}[\![\, \mathrm{C}?\mathrm{price}.\mathrm{C}!\mathrm{ok}.\mathrm{C}!\mathrm{card}.\mathrm{C}?\mathrm{date} \,]\!] \parallel \\
\mathrm{C}[\![\, \mathrm{A}!\mathrm{price}.\mathrm{A}?\mathrm{ok}.\mathrm{B}!\mathrm{price}.\mathrm{B}!\in.\mathrm{C}?\ni.\mathrm{A}!\mathrm{date} \,]\!] \parallel \\
\mathrm{B}[\![\, \mathrm{C}?\mathrm{price}.\mathrm{C}?\in.\mathrm{A}?\mathrm{card}.\mathrm{B}!\ni \,]\!] \parallel \emptyset \\
\downarrow^{*} \\
\mathrm{A}[\![\, \mathrm{C}!\mathrm{card}.\mathrm{C}?\mathrm{date} \,]\!] \parallel \mathrm{C}[\![\, \mathrm{B}!\in.\mathrm{C}?\ni.\mathrm{A}!\mathrm{date} \,]\!] \parallel \mathrm{B}[\![\, \mathrm{C}?\in.\mathrm{A}?\mathrm{card}.\mathrm{B}!\ni \,]\!] \parallel \emptyset \\
\downarrow_{\mathrm{CB}!\in} \\
\mathrm{A}[\![\, \mathrm{C}!\mathrm{card}.\mathrm{C}?\mathrm{date} \,]\!] \parallel \overset{*}{\mathrm{B}}[\![\, \mathrm{C}?\ni.\mathrm{A}!\mathrm{date} \,]\!] \parallel \mathrm{B}[\![\, \mathrm{C}?\in.\mathrm{A}?\mathrm{card}.\mathrm{B}!\ni \,]\!] \parallel \langle \mathrm{C}, \in, \mathrm{B} \rangle \\
\downarrow_{\mathrm{BC}?\in} \\
\mathrm{A}[\![\, \mathrm{C}!\mathrm{card}.\mathrm{C}?\mathrm{date} \,]\!] \parallel \overset{*}{\mathrm{B}}[\![\, \mathrm{C}?\ni.\mathrm{A}!\mathrm{date} \,]\!] \parallel \mathrm{C}[\![\, \mathrm{A}?\mathrm{card}.\mathrm{B}!\ni \,]\!] \parallel \emptyset \\
\downarrow^{*} \\
\mathrm{A}[\![\, \mathrm{C}?\mathrm{date} \,]\!] \parallel \overset{*}{\mathrm{B}}[\![\, \mathrm{C}?\ni.\mathrm{A}!\mathrm{date} \,]\!] \parallel \mathrm{C}[\![\, \mathrm{B}!\ni \,]\!] \parallel \emptyset
\end{array}
$$

Sessions

$$A[\![\, C!\mathrm{card}.C?\mathrm{date} \,]\!] \parallel C[\![\, B!{\Subset}.C?{\ni}.A!\mathrm{date} \,]\!] \parallel B[\![\, C?{\Subset}.A?\mathrm{card}.B!{\ni} \,]\!] \parallel \emptyset$$

$$\downarrow_{CB!\Subset}$$

$$A[\![\, C!\mathrm{card}.C?\mathrm{date} \,]\!] \parallel \overset{*}{B}[\![\, C?{\ni}.A!\mathrm{date} \,]\!] \parallel B[\![\, C?{\Subset}.A?\mathrm{card}.B!{\ni} \,]\!] \parallel \langle C, \Subset, B \rangle$$

$$\downarrow_{BC?\Subset}$$

$$A[\![\, C!\mathrm{card}.C?\mathrm{date} \,]\!] \parallel \overset{*}{B}[\![\, C?{\ni}.A!\mathrm{date} \,]\!] \parallel C[\![\, A?\mathrm{card}.B!{\ni} \,]\!] \parallel \emptyset$$

$$\downarrow^{*}$$

$$A[\![\, C?\mathrm{date} \,]\!] \parallel \overset{*}{B}[\![\, C?{\ni}.A!\mathrm{date} \,]\!] \parallel C[\![\, B!{\ni} \,]\!] \parallel \emptyset$$

$$\downarrow_{CB!\ni}$$

$$A[\![\, C?\mathrm{date} \,]\!] \parallel \overset{*}{B}[\![\, C?{\ni}.A!\mathrm{date} \,]\!] \parallel B[\![\, \mathbf{0} \,]\!] \parallel \langle C, \ni, B \rangle$$

sending the backward delegation the deputy goes back to its identity

Sessions

$$A[\![\, C!\mathsf{card}.C?\mathsf{date} \,]\!] \parallel C[\![\, B!\Subset.C?\ni.A!\mathsf{date} \,]\!] \parallel B[\![\, C?\Subset.A?\mathsf{card}.B!\ni \,]\!] \parallel \emptyset$$

$$\downarrow_{CB!\Subset}$$

$$A[\![\, C!\mathsf{card}.C?\mathsf{date} \,]\!] \parallel \overset{*}{B}[\![\, C?\ni.A!\mathsf{date} \,]\!] \parallel B[\![\, C?\Subset.A?\mathsf{card}.B!\ni \,]\!] \parallel \langle C, \Subset, B \rangle$$

$$\downarrow_{BC?\Subset}$$

$$A[\![\, C!\mathsf{card}.C?\mathsf{date} \,]\!] \parallel \overset{*}{B}[\![\, C?\ni.A!\mathsf{date} \,]\!] \parallel C[\![\, A?\mathsf{card}.B!\ni \,]\!] \parallel \emptyset$$

$$\downarrow^{*}$$

$$A[\![\, C?\mathsf{date} \,]\!] \parallel \overset{*}{B}[\![\, C?\ni.A!\mathsf{date} \,]\!] \parallel C[\![\, B!\ni \,]\!] \parallel \emptyset$$

$$\downarrow_{CB!\ni}$$

$$A[\![\, C?\mathsf{date} \,]\!] \parallel \overset{*}{B}[\![\, C?\ni.A!\mathsf{date} \,]\!] \parallel B[\![\, \mathbf{0} \,]\!] \parallel \langle C, \ni, B \rangle$$

$$\downarrow_{BC?\ni}$$

$$A[\![\, C?\mathsf{date} \,]\!] \parallel C[\![\, A!\mathsf{date} \,]\!] \parallel B[\![\, \mathbf{0} \,]\!] \parallel \emptyset$$

receiving the backward delegation the principal goes back to its identity

Sessions

$$\mathsf{A}[\![\,\mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date}\,]\!] \parallel \mathsf{C}[\![\,\mathsf{B}!\in.\mathsf{C}?\ni.\mathsf{A}!\mathsf{date}\,]\!] \parallel \mathsf{B}[\![\,\mathsf{C}?\in.\mathsf{A}?\mathsf{card}.\mathsf{B}!\ni\,]\!] \parallel \emptyset$$

$$\downarrow_{\mathsf{CB}!\in}$$

$$\mathsf{A}[\![\,\mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date}\,]\!] \parallel \overset{*}{\mathsf{B}}[\![\,\mathsf{C}?\ni.\mathsf{A}!\mathsf{date}\,]\!] \parallel \mathsf{B}[\![\,\mathsf{C}?\in.\mathsf{A}?\mathsf{card}.\mathsf{B}!\ni\,]\!] \parallel \langle \mathsf{C}, \in, \mathsf{B} \rangle$$

$$\downarrow_{\mathsf{BC}?\in}$$

$$\mathsf{A}[\![\,\mathsf{C}!\mathsf{card}.\mathsf{C}?\mathsf{date}\,]\!] \parallel \overset{*}{\mathsf{B}}[\![\,\mathsf{C}?\ni.\mathsf{A}!\mathsf{date}\,]\!] \parallel \mathsf{C}[\![\,\mathsf{A}?\mathsf{card}.\mathsf{B}!\ni\,]\!] \parallel \emptyset$$

$$\downarrow^{*}$$

$$\mathsf{A}[\![\,\mathsf{C}?\mathsf{date}\,]\!] \parallel \overset{*}{\mathsf{B}}[\![\,\mathsf{C}?\ni.\mathsf{A}!\mathsf{date}\,]\!] \parallel \mathsf{C}[\![\,\mathsf{B}!\ni\,]\!] \parallel \emptyset$$

$$\downarrow_{\mathsf{CB}!\ni}$$

$$\mathsf{A}[\![\,\mathsf{C}?\mathsf{date}\,]\!] \parallel \overset{*}{\mathsf{B}}[\![\,\mathsf{C}?\ni.\mathsf{A}!\mathsf{date}\,]\!] \parallel \mathsf{B}[\![\,\mathbf{0}\,]\!] \parallel \langle \mathsf{C}, \ni, \mathsf{B} \rangle$$

$$\downarrow_{\mathsf{BC}?\ni}$$

$$\mathsf{A}[\![\,\mathsf{C}?\mathsf{date}\,]\!] \parallel \mathsf{C}[\![\,\mathsf{A}!\mathsf{date}\,]\!] \parallel \mathsf{B}[\![\,\mathbf{0}\,]\!] \parallel \emptyset$$

$$\downarrow^{*}$$

$$\mathsf{A}[\![\,\mathbf{0}\,]\!] \parallel \mathsf{C}[\![\,\mathbf{0}\,]\!] \parallel \mathsf{B}[\![\,\mathbf{0}\,]\!] \parallel \emptyset$$

Simple Choreography Typing

Simple Choreography Typing

no local types no projections only global types

Simple Choreography Typing

no local types no projections only global types

global types are built with processes

Simple Choreography Typing

no local types no projections only global types

global types are built with processes

wellformedness conditions on global types and queues

Simple Choreography Typing

no local types no projections only global types

global types are built with processes

wellformedness conditions on global types and queues

$$\vdash \mathsf{A}[\![\,\mathbf{0}\,]\!] \parallel \mathsf{C}[\![\,\mathbf{0}\,]\!] \parallel \mathsf{B}[\![\,\mathbf{0}\,]\!] \parallel \emptyset : \mathtt{End}$$

Simple Choreography Typing

no local types no projections only global types

global types are built with processes

wellformedness conditions on global types and queues

$$\frac{\vdash A[\![\,\mathbf{0}\,]\!] \parallel C[\![\,\mathbf{0}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \emptyset : \mathtt{End}}{\vdash A[\![\,C?\mathrm{date}\,]\!] \parallel C[\![\,\mathbf{0}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \langle A, \mathrm{date}, C\rangle : AC?\mathrm{date}}$$

Simple Choreography Typing

no local types no projections only global types

global types are built with processes

wellformedness conditions on global types and queues

$$\frac{\vdash A[\![\,\mathbf{0}\,]\!] \parallel C[\![\,\mathbf{0}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \emptyset : \mathtt{End}}{\vdash A[\![\,C?\mathrm{date}\,]\!] \parallel C[\![\,\mathbf{0}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \langle A, \mathrm{date}, C\rangle : AC?\mathrm{date}}$$

$$\vdash A[\![\,C?\mathrm{date}\,]\!] \parallel C[\![\,A!\mathrm{date}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \emptyset : CA!\mathrm{date}.AC?\mathrm{date}$$

Simple Choreography Typing

no local types no projections only global types

global types are built with processes

wellformedness conditions on global types and queues

$$\dfrac{\dfrac{\dfrac{\vdash A[\![\mathbf{0}]\!] \parallel C[\![\mathbf{0}]\!] \parallel B[\![\mathbf{0}]\!] \parallel \emptyset : \mathtt{End}}{\vdash A[\![C?\mathrm{date}]\!] \parallel C[\![\mathbf{0}]\!] \parallel B[\![\mathbf{0}]\!] \parallel \langle A, \mathrm{date}, C\rangle : AC?\mathrm{date}}}{\vdash A[\![C?\mathrm{date}]\!] \parallel C[\![A!\mathrm{date}]\!] \parallel B[\![\mathbf{0}]\!] \parallel \emptyset : CA!\mathrm{date}.AC?\mathrm{date}}}{\vdash A[\![C?\mathrm{date}]\!] \parallel \overset{*}{B}[\![C?\ni.A!\mathrm{date}]\!] \parallel B[\![\mathbf{0}]\!] \parallel \langle C, \ni, B\rangle : BC?\ni.CA!\mathrm{date}.AC?\mathrm{date}}$$

Simple Choreography Typing

no local types no projections only global types

global types are built with processes

wellformedness conditions on global types and queues

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash A[\![\,\mathbf{0}\,]\!] \parallel C[\![\,\mathbf{0}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \emptyset : \mathtt{End}}{\vdash A[\![\,C?\mathrm{date}\,]\!] \parallel C[\![\,\mathbf{0}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \langle A, \mathrm{date}, C\rangle : AC?\mathrm{date}}}{\vdash A[\![\,C?\mathrm{date}\,]\!] \parallel C[\![\,A!\mathrm{date}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \emptyset : CA!\mathrm{date}.AC?\mathrm{date}}}{\vdash A[\![\,C?\mathrm{date}\,]\!] \parallel \overset{*}{B}[\![\,C?\ni.A!\mathrm{date}\,]\!] \parallel B[\![\,\mathbf{0}\,]\!] \parallel \langle C, \ni, B\rangle : BC?\ni.CA!\mathrm{date}.AC?\mathrm{date}}}{\vdash A[\![\,C?\mathrm{date}\,]\!] \parallel \overset{*}{B}[\![\,C?\ni.A!\mathrm{date}\,]\!] \parallel C[\![\,B!\ni\,]\!] \parallel \emptyset : CB!\ni.BC?\ni.CA!\mathrm{date}.AC?\mathrm{date}}$$

Properties

An LTS for global types in parallel with queues

Properties

An LTS for global types in parallel with queues

Subject Reduction If a session typed by a global type reduces with a tag, then the parallel of the global type with the queue of the session reduces with the same tag and the reduced global type types the reduced session.

Properties

An LTS for global types in parallel with queues

Subject Reduction If a session typed by a global type reduces with a tag, then the parallel of the global type with the queue of the session reduces with the same tag and the reduced global type types the reduced session.

Session Fidelity If a session is typed by a global type and the parallel of the global type with the queue of the session reduces with a tag, then the session reduces with the same tag and the reduced global type types the reduced session.

Properties

An LTS for global types in parallel with queues

Subject Reduction If a session typed by a global type reduces with a tag, then the parallel of the global type with the queue of the session reduces with the same tag and the reduced global type types the reduced session.

Session Fidelity If a session is typed by a global type and the parallel of the global type with the queue of the session reduces with a tag, then the session reduces with the same tag and the reduced global type types the reduced session.

Lock freedom A typed session is lock free.

Properties

An LTS for global types in parallel with queues

Subject Reduction If a session typed by a global type reduces with a tag, then the parallel of the global type with the queue of the session reduces with the same tag and the reduced global type types the reduced session.

Session Fidelity If a session is typed by a global type and the parallel of the global type with the queue of the session reduces with a tag, then the session reduces with the same tag and the reduced global type types the reduced session.

Lock freedom A typed session is lock free.

Orphan-message freedom A typed session is orphan-message free.

Properties

An LTS for global types in parallel with queues

Subject Reduction If a session typed by a global type reduces with a tag, then the parallel of the global type with the queue of the session reduces with the same tag and the reduced global type types the reduced session.

Session Fidelity If a session is typed by a global type and the parallel of the global type with the queue of the session reduces with a tag, then the session reduces with the same tag and the reduced global type types the reduced session.

Lock freedom A typed session is lock free.

Orphan-message freedom A typed session is orphan-message free.

In a typed session with the empty queue each delegation start is followed by a delegation end.

Thanks


obrigado
Dank U
Merci
mahalo
Köszi
спасибо
Grazie
Thank you
mauruuru
Takk
Gracias
Dziękuję
Děkuju
danke
Kiitos