# Mobility Reading Group

## an introduction

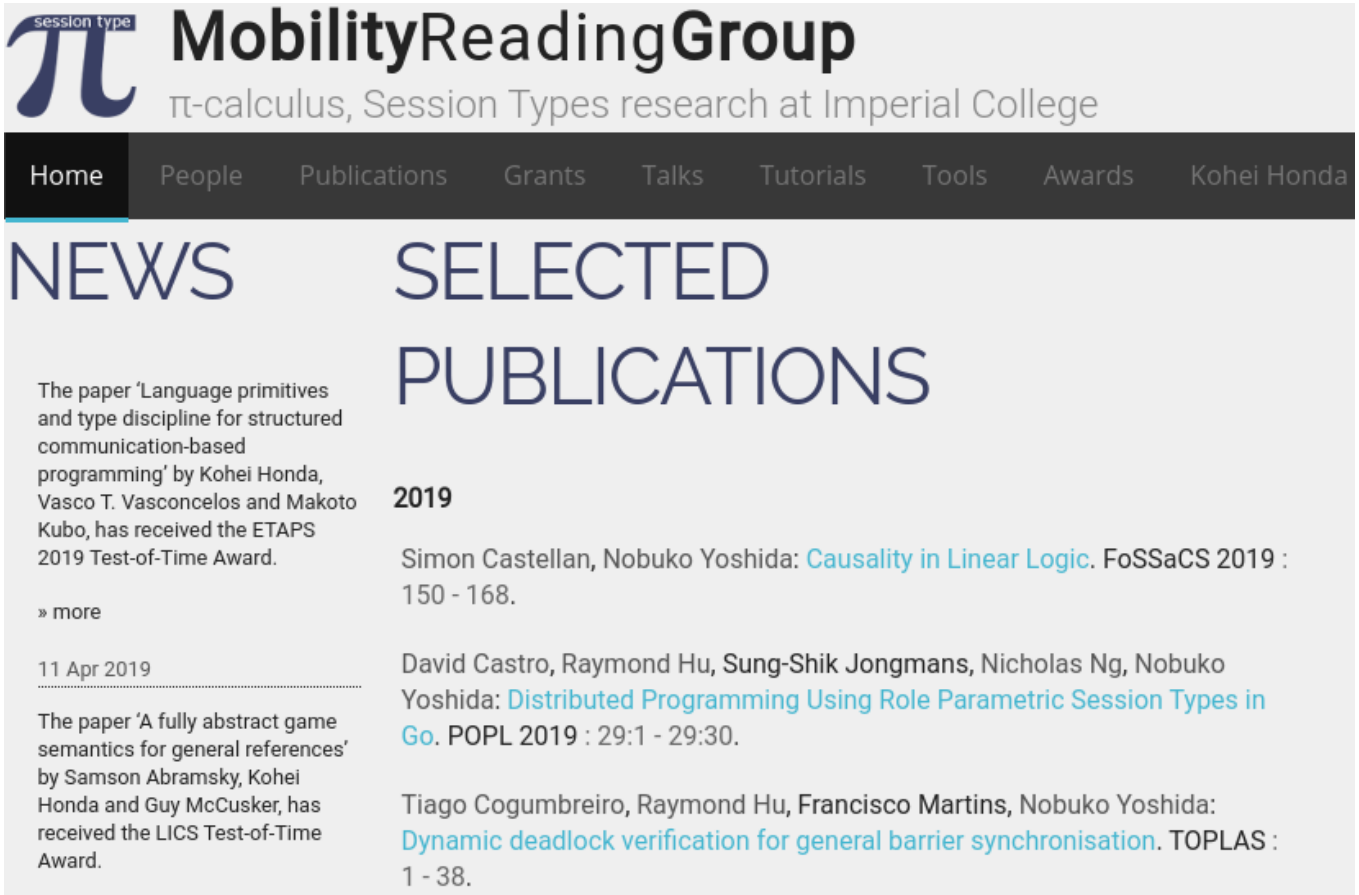
Nobuko Yoshida and David Castro-Perez

2019

# The Mobility Reading Group at Imperial College

Our research group is **specialised in $\pi$-calculus** and **session types** — both **theory** and **applications**

http://mrg.doc.ic.ac.uk/



# Awards


ETAPS 2019
TEST-OF-TIME
AWARD
Language primitives and type discipline for structured communication-based programming' by Kohei Honda, Vasco T. Vasconcelos and Makoto Kubo

# Awards

## POPL 2008 MOST INFLUENTIAL PAPER AWARD

# Awards


LICS'98

## A fully abstract game semantics for general references

Samson Abramsky Kohei Honda
LFCS, University of Edinburgh
{samson,kohei}@dcs.ed.ac.uk

Guy McCusker
St John's College, Oxford
mccusker@comlab.ox.ac.uk

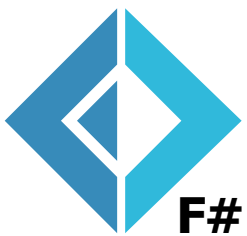
**Abstract**

*A games model of a programming language with higher-order store in the style of ML-references is introduced. The category used for the model is obtained by relaxing certain behavioural conditions on a category of games previously used to provide fully abstract models of pure functional languages. The model is shown to be fully abstract by means of factorization arguments which reduce the question of definability for the language with higher-order store to that for its purely functional fragment.*

# Session types in programming languages

(Elements of) the session types theory have been **implemented in many languages**, e.g.:



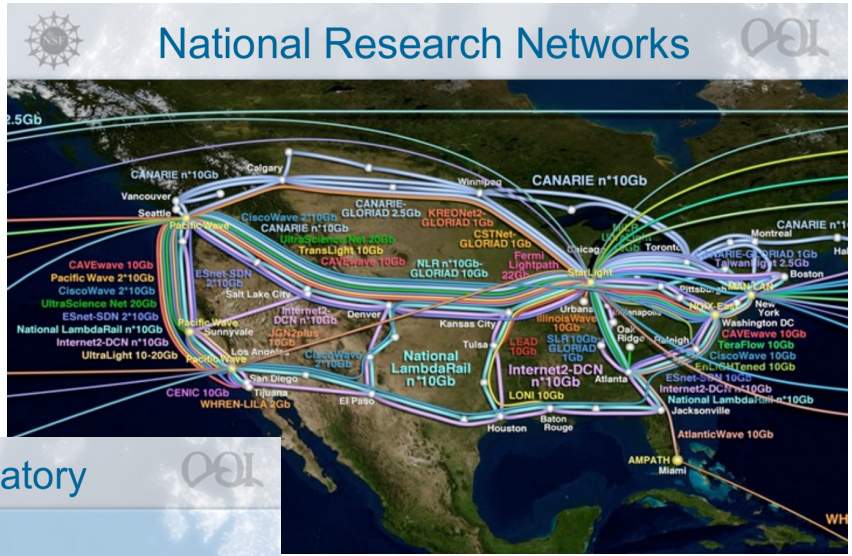



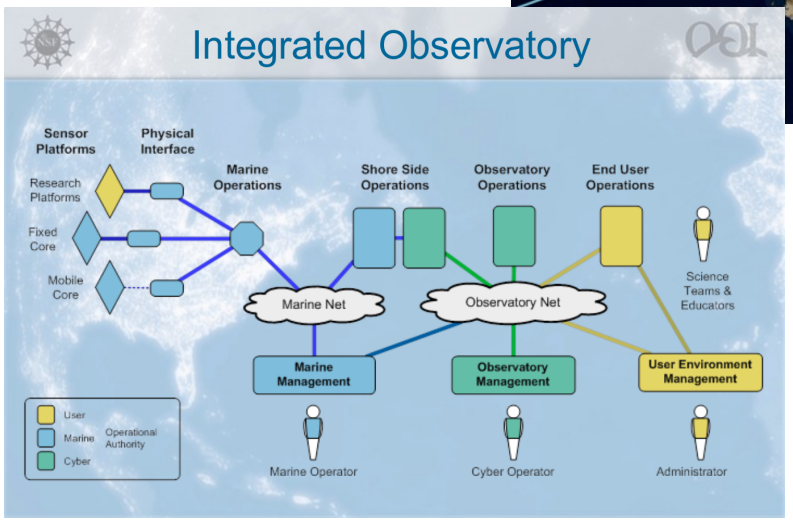




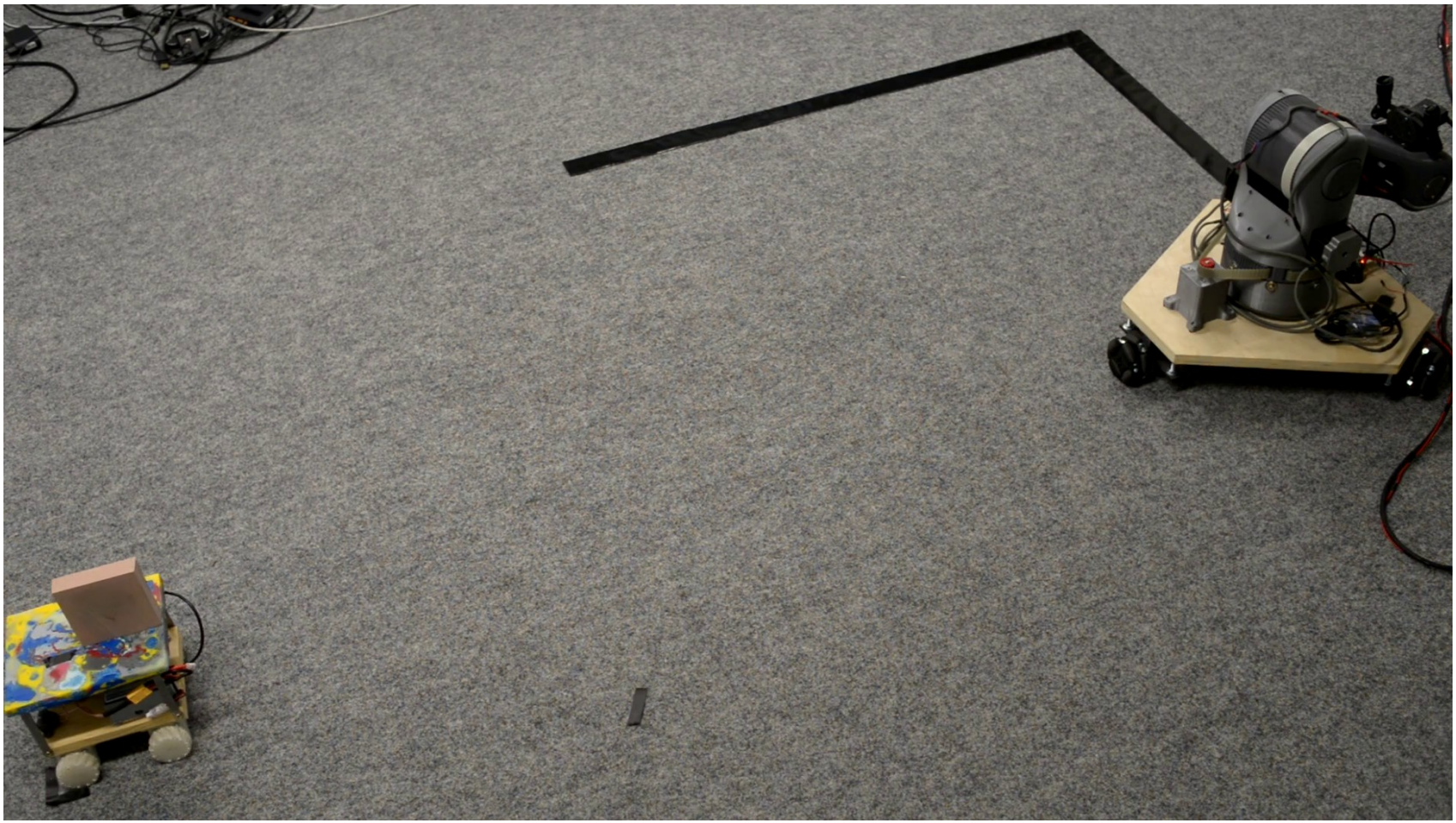
# Collaborations (I): Ocean Observatories Initiative

Collaboration: use session types on top of OOI network to **guarantee global safety**

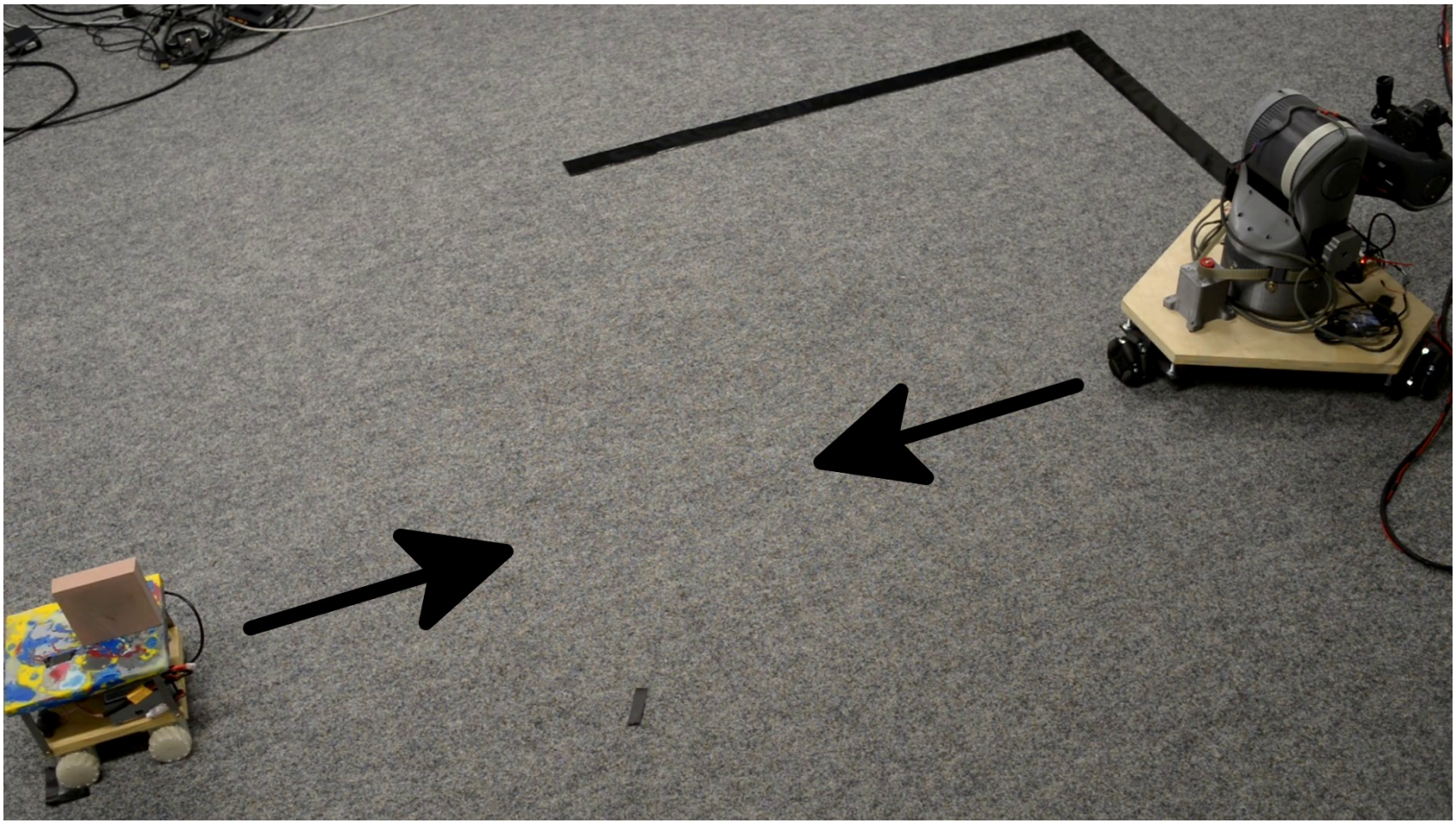



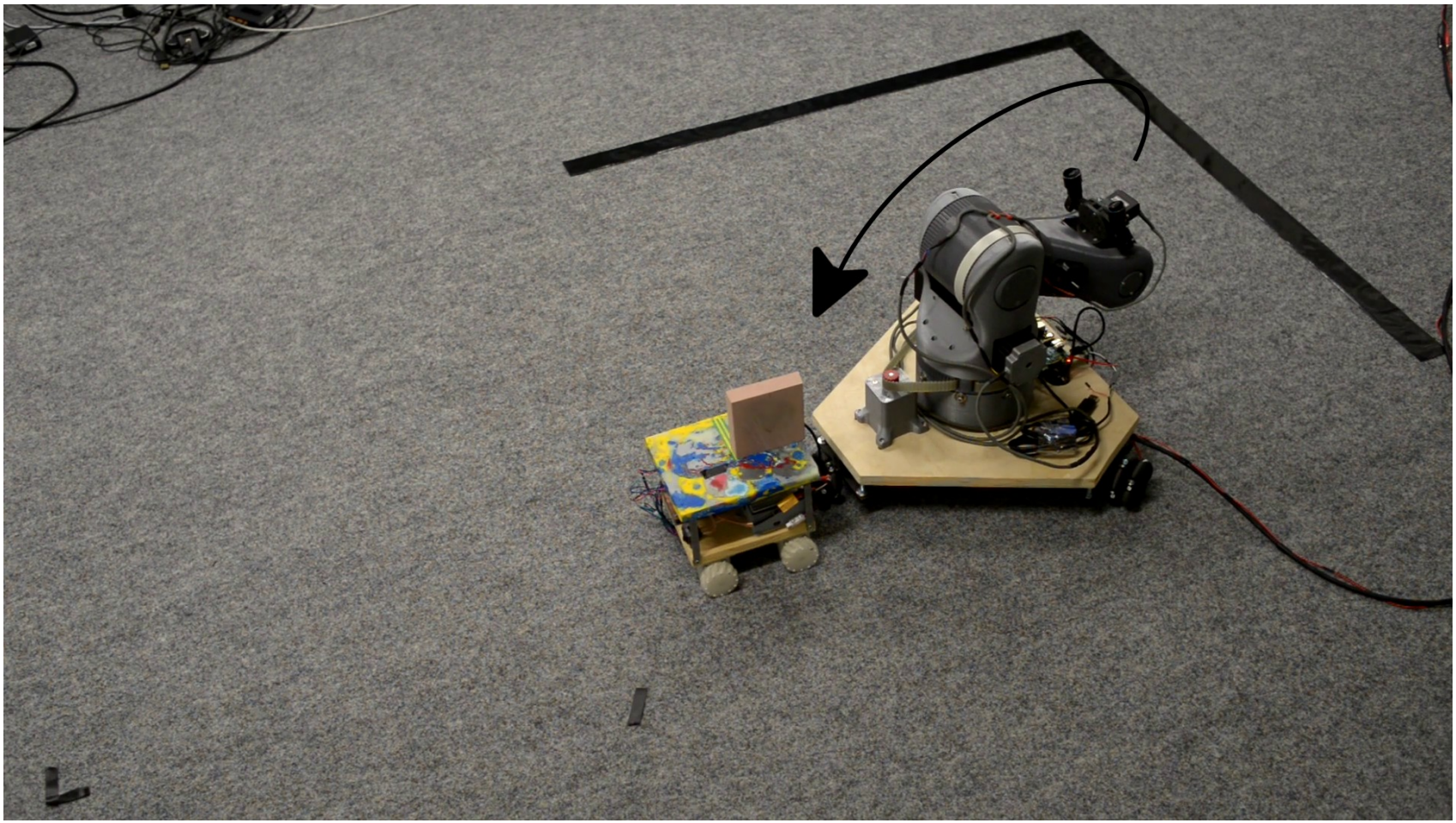
# Collaborations (and II): Session Types for Robotics



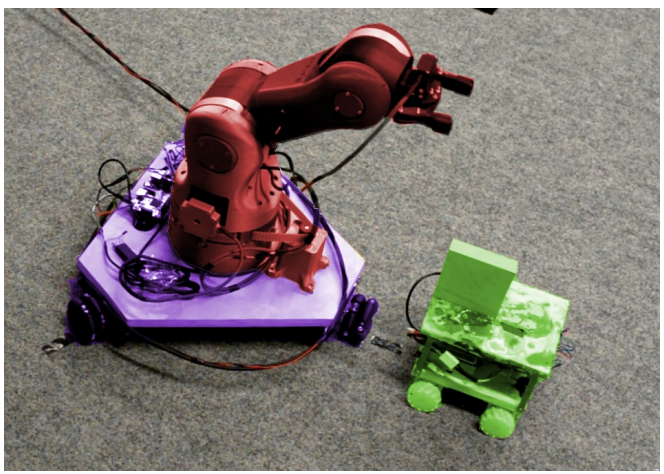
# Collaborations (and II): Session Types for Robotics



# Collaborations (and II): Session Types for Robotics



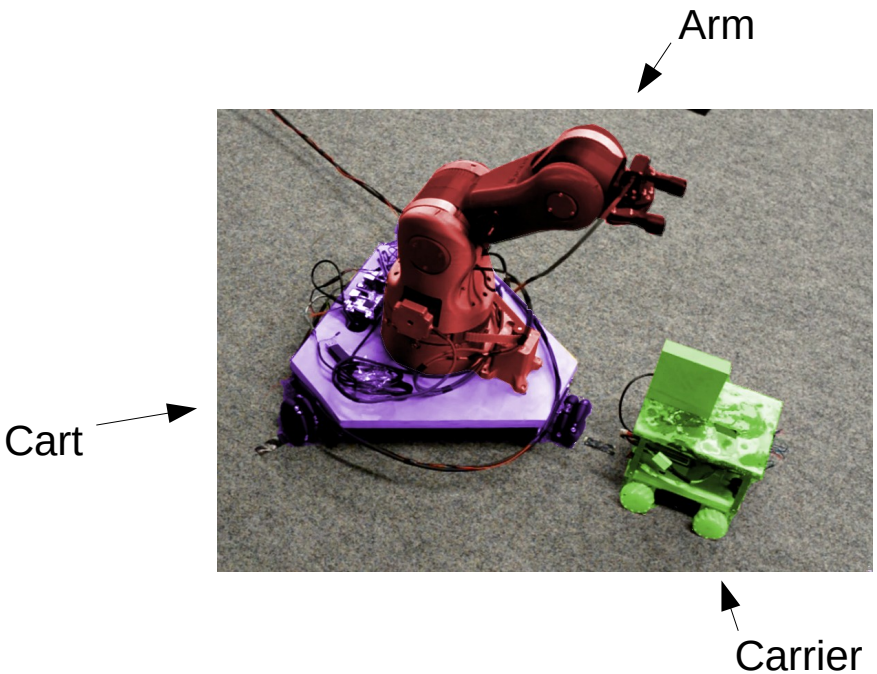
# Collaborations (and II): Session Types for Robotics

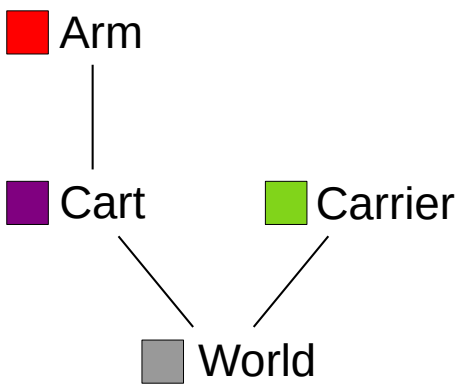



Cart & Arm:

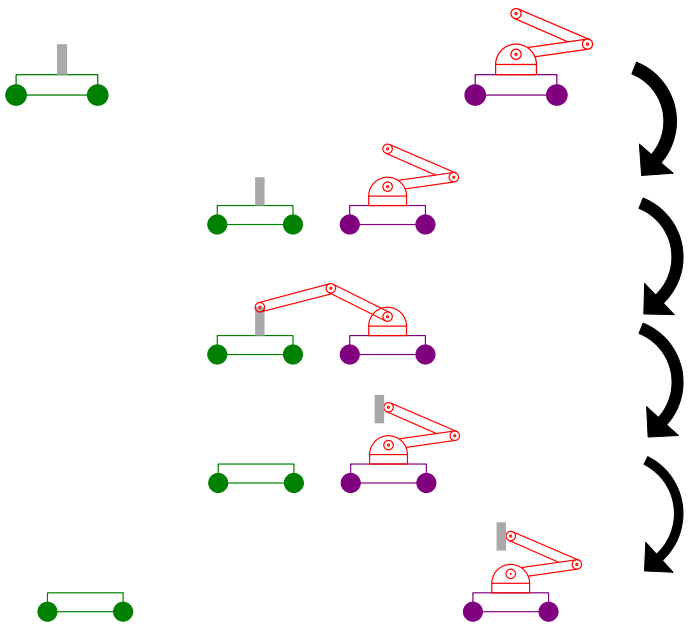
Two robots attached together that act as “one” robot (communication)

# Collaborations (and II): Session Types for Robotics



# Collaborations (and II): Session Types for Robotics



$\mathsf{Cart} \to \mathsf{Arm} : \mathit{fold}(\mathtt{unit}).\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{fold}\rangle.$

$\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.$

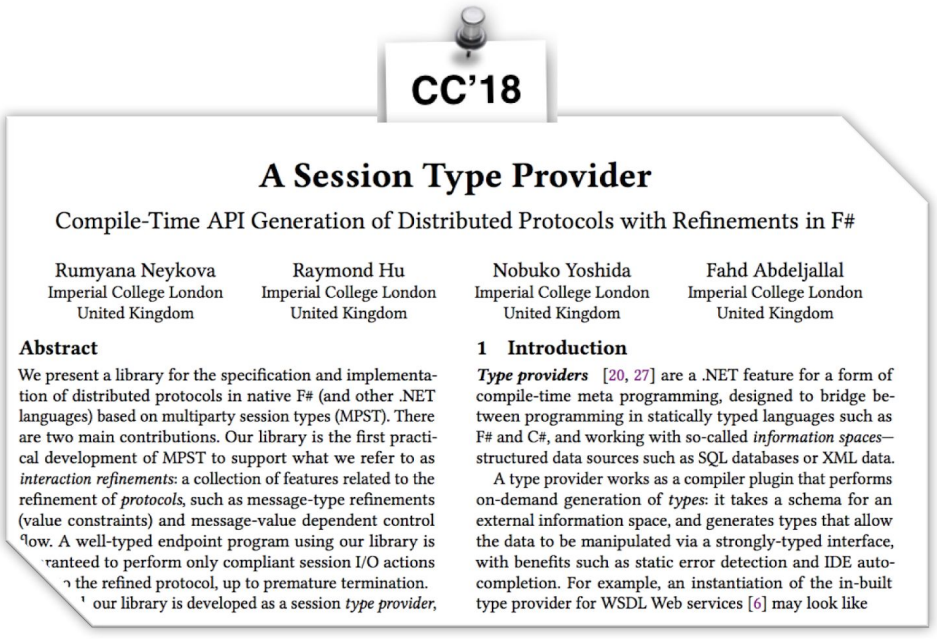
$\mathsf{Carrier} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Arm} : \mathit{grab}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{Cart} : \mathsf{idle}, \mathsf{Carrier} : \mathsf{idle}, \mathsf{Arm} : \mathsf{grip}\rangle.$

$\mathsf{Arm} \to \mathsf{Cart} : \mathit{ok}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{ok}(\mathtt{unit}).$

$\mathsf{dt}\langle \mathsf{Cart} : \mathsf{move}, \mathsf{Carrier} : \mathsf{move}, \mathsf{Arm} : \mathsf{idle}\rangle.$

$\mathsf{Cart} \to \mathsf{Arm} : \mathit{done}(\mathtt{unit}).\mathsf{Cart} \to \mathsf{Carrier} : \mathit{done}(\mathtt{unit}).\mathtt{end}$



CC'18

# A Session Type Provider

## Compile-Time API Generation of Distributed Protocols with Refinements in F#


Rumyana Neykova
Imperial College London
United Kingdom

Raymond Hu
Imperial College London
United Kingdom

Nobuko Yoshida
Imperial College London
United Kingdom

Fahd Abdeljallal
Imperial College London
United Kingdom



**Abstract**

We present a library for the specification and implementation of distributed protocols in native F# (and other .NET languages) based on multiparty session types (MPST). There are two main contributions. Our library is the first practical development of MPST to support what we refer to as *interaction refinements*: a collection of features related to the refinement of *protocols*, such as message-type refinements (value constraints) and message-value dependent control flow. A well-typed endpoint program using our library is guaranteed to perform only compliant session I/O actions on the refined protocol, up to premature termination. Our library is developed as a session *type provider*,


**1 Introduction**

***Type providers*** [20, 27] are a .NET feature for a form of compile-time meta programming, designed to bridge between programming in statically typed languages such as F# and C#, and working with so-called *information spaces*—structured data sources such as SQL databases or XML data.

A type provider works as a compiler plugin that performs on-demand generation of *types*: it takes a schema for an external information space, and generates types that allow the data to be manipulated via a strongly-typed interface, with benefits such as static error detection and IDE auto-completion. For example, an instantiation of the in-built type provider for WSDL Web services [6] may look like

**Graydon Hoare** @graydon_pub

(This stuff is _fantastic_)

11:31 PM - 11 Mar 2018

32 Retweets 83 Likes

**shots fired** @zeeshanlakhani · Mar 12

Replying to @graydon_pub @dsyme

Awesome!

**Brendan Zabarauskas** @brendanzab ·

Replying to @graydon_pub

This stuff fills me with hope!

**Ryan Riley** @panesofglass · Mar 12

Replying to @graydon_pub

This is amazing! I guess I need to switch



Imperial College London

News

Home | College and Campus | Science | Engineering | Health | Business | Search here... | Go

**Go concurrency verification research at DoC grabs headline**

POPL'17

currency
erates a

tured in the
hich
nteresting
easily
e of the
L (Principles

ICSE'18

# the morning paper

an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

Home About InfoQ QR Editions Subscribe

## A static verification framework for message passing in Go using behavioural types

JANUARY 25, 2018

*tags:* Concurrency, Programming Languages

**A static verification framework for message passing in Go using behavioural types** Lange et al., *ICSE 18*

*With thanks to Alexis Richardson who first forwarded this paper to me.*

We're jumping ahead to ICSE 18 now, and a paper that has been accepted for publication there later this year. It fits with the theme we've been exploring this week though, so I thought I'd cover it now. We've seen verification techniques applied in the context of **Rust** and **JavaScript**, looked at the integration of **linear types in Haskell**, and today it is the turn of Go!

SUBSCRIBE

never miss an issue! The Morning Paper delivered straight to your inbox.

SEARCH

type and press enter

ARCHIVES

Select Month

MOST READ IN THE LAST FEW DAYS

# Selected Publications [2018-2019]

- [PLDI19] Alceste Scalas, NY, Elias Benussi: Verifying message-passing programs with dependent behavioural types.
- [CAV19] Julien Lange, NY: Verifying Asynchronous Interactions via Communicating Session Automata.
- [CONCUR19] Mario Bravetti, Marco Carbone, Julien Lange, NY, Gianluigi Zavattaro: A Sound Algorithm for Asynchronous Session Subtyping
- [FSE19] Nicola Atzei, Massimo Bartoletti, Stefano Lande, NY, Roberto Zunino:Developing secure Bitcoin contracts with BitML
- [ECOOP19] Rupak Majumdar, Marcus Pirron, NY, and Damien Zufferey, Motion Session Types for Robotic Interactions
- [FoSSaCs19] Simon Castellan, NY: Causality in Linear Logic
- [ESOP19] Laura Bocchi, Maurizio Murgia, Vasco T. Vasconcelos, NY: Asynchronous Timed Session Types
- [POPL19] Simon Castellan, NY: Two Sides of the Same Coin: Session Types and Game Semantics
- [POPL19] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, NY: Distributed Programming Using Role Parametric Session Types in Go
- [POPL19] Alceste Scalas, Nobuko Yoshida: Less Is More: Multiparty Session Types Revisited
- [POPL19] Bernardo Toninho, NY: Interconnectability of Session Based Logical Processes
- [ICSE18] Julien Lange, Nicholas Ng, Bernardo Toninho, NY: A Static Verification Framework for Message Passing in Go using Behavioural Types
- [LICS18] Romain Demangeon, NY: Causal Computational Complexity of Distributed Processes
- [FoSSaCs18] Bernardo Toninho, Nobuko Yoshida: Depending On Session Typed Process
- [ESOP18] Bernardo Toninho, NY: On Polymorphic Sessions And Functions: A Tale of Two (Fully Abstract) Encodings
- [CC18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#

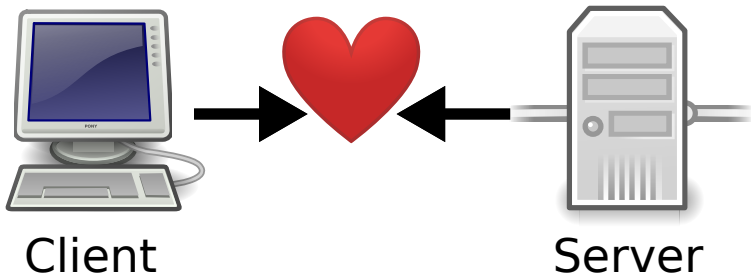
# Selected Publications [2018-2019]

- [PLDI19] Alceste Scalas, NY, Elias Benussi: Verifying message-passing programs with dependent behavioural types.
- [CAV19] Julien Lange, NY: Verifying Asynchronous Interactions via Communicating Session Automata.
- [CONCUR19] Mario Bravetti, Marco Carbone, Julien Lange, NY, Gianluigi Zavattaro: A Sound Algorithm for Asynchronous Session Subtyping
- [FSE19] Nicola Atzei, Massimo Bartoletti, Stefano Lande, NY, Roberto Zunino:Developing secure Bitcoin contracts with BitML
- [ECOOP19] Rupak Majumdar, Marcus Pirron, NY, and Damien Zufferey, Motion Session Types for Robotic Interactions
- [FoSSaCs19] Simon Castellan, NY: Causality in Linear Logic
- [ESOP19] Laura Bocchi, Maurizio Murgia, Vasco T. Vasconcelos, NY: Asynchronous Timed Session Types
- [POPL19] Simon Castellan, NY: Two Sides of the Same Coin: Session Types and Game Semantics
- [POPL19] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, NY: Distributed Programming Using Role Parametric Session Types in Go
- [POPL19] Alceste Scalas, Nobuko Yoshida: Less Is More: Multiparty Session Types Revisited
- [POPL19] Bernardo Toninho, NY: Interconnectability of Session Based Logical Processes
- [ICSE18] Julien Lange, Nicholas Ng, Bernardo Toninho, NY: A Static Verification Framework for Message Passing in Go using Behavioural Types
- [LICS18] Romain Demangeon, NY: Causal Computational Complexity of Distributed Processes
- [FoSSaCs18] Bernardo Toninho, Nobuko Yoshida: Depending On Session Typed Process
- [ESOP18] Bernardo Toninho, NY: On Polymorphic Sessions And Functions: A Tale of Two (Fully Abstract) Encodings
- [CC18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#

# From client-server...

Session types can formalise **complex client-server protocols**...

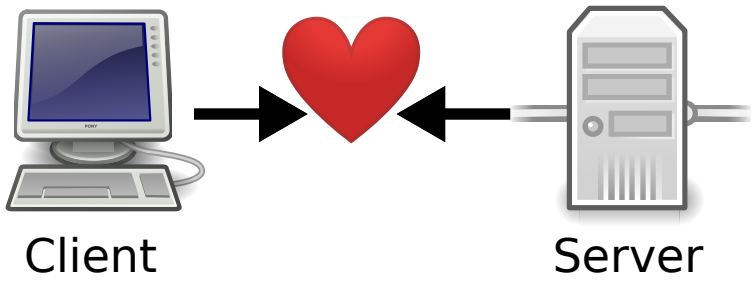
- SMTP
- HTTP
- POP3
- ...



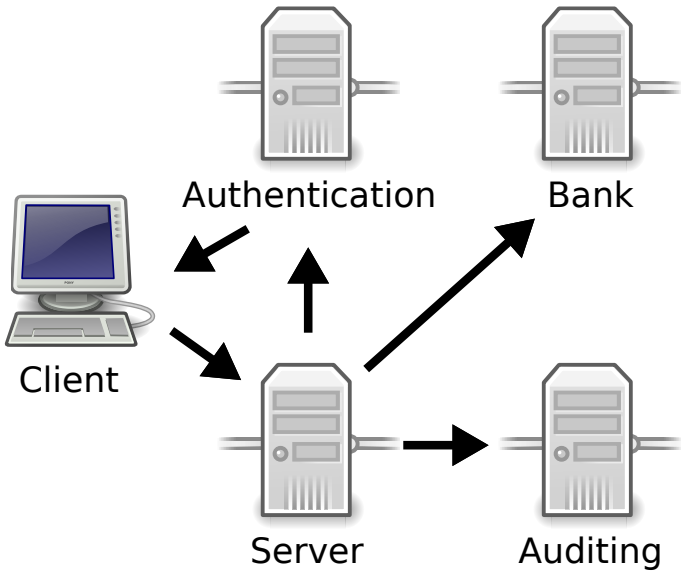
# From client-server... to multiparty protocols

Session types can formalise **complex client-server protocols**...

- SMTP
- HTTP
- POP3
- ...




...and also extend to **multiparty protocols**



# Data types

One of the **most successful concepts** in computer science

Data types describe the **intended usage** of data

```
int x = 1;
int y = 2;
int z = x + y; // Correct :)
```

# Data types

One of the **most successful concepts** in computer science

Data types describe the **intended usage** of data

```
int x = 1;
int y = 2;
int z = x + y; // Correct :)
```

Allow to **document code** and **spot errors at compile-time**

```
public int add(int a, int b) { ... }

int x = add(1, 2)       // Correct :)
int y = add(1, "Hi!") // Does not compile :)
```

# Data types (cont'd)

```
public int add(int a, int b) { ... }
```

From this definition, we can tell that:

1. add takes two integers a, b
2. it does *something*, possibly **using** a, b
   - and if a,b are used, they are **only used as integers**
3. finally, it returns an integer

# What about communicating systems?

Say that we need to write a **network client** implementing a **certain protocol**. Are data types helpful?

# What about communicating systems?

Say that we need to write a **network client** implementing a **certain protocol**. Are data types helpful?

```
public void client(Socket s) { ... }
```

From this definition, we can tell that:

1. `client` takes a **network socket** `s`
2. it can **use** `s` **to send/receive data** to/from the server
3. it returns nothing

# What about communicating systems?

Say that we need to write a **network client** implementing a **certain protocol**. Are data types helpful?

```
public void client(Socket s) { ... }
```

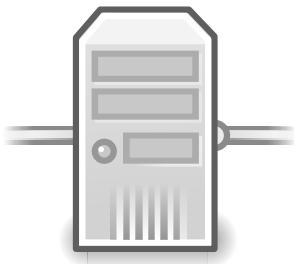
From this definition, we can tell that:

1. `client` takes a **network socket** `s`
2. it can **use** `s` **to send/receive data** to/from the server
3. it returns nothing

But **what messages are sent/received?**
And **in what order?**
Does `client` really **implement the intended protocol?**

# What could possibly go wrong?

Client

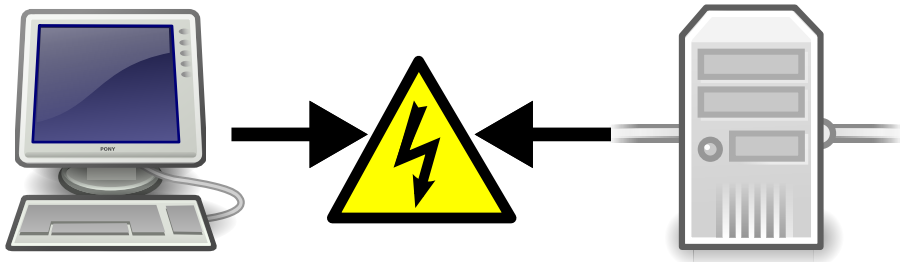


Server

E.g., consider a very simple protocol:

1. a **client** **sends two integers** to a **server**
2. then, the **server** **answers with a boolean**

# What could possibly go wrong?




E.g., consider a very simple protocol:

1. a **client** **sends two integers** to a **server**
2. then, the **server** **answers with a boolean**

If one of the two does not correctly implement its part of the protocol, we can encounter **various run-time errors**:

- **mismatching inputs/outputs** ✗
- **deadlocks** ✗

# From data types to session types

A **session type** formalises a **protocol** as a **structured sequence of communications**

- unlike data types, session types focus on **communication**

# From data types to session types

A **session type** formalises a **protocol** as a **structured sequence of communications**

- unlike data types, session types focus on **communication**

E.g., consider again our simple protocol:

1. a **client** **sends two integers** to a **server**
2. then, the **server** **answers with a boolean**

# From data types to session types

A **session type** formalises a **protocol** as a **structured sequence of communications**

- unlike data types, session types focus on **communication**

E.g., consider again our simple protocol:

1. a **client** **sends two integers** to a **server**
2. then, the **server** **answers with a boolean**

From the **client** **viewpoint**, the **session type** is:

$$\text{Session} = \mathbf{send}(\text{int}).\mathbf{send}(\text{int}).\mathbf{receive}(\text{bool})$$

# Session types enforce channel usage

From the **client** **viewpoint**, the **session type** is:

$$\text{Session} = \mathbf{send}(\text{int}).\mathbf{send}(\text{int}).\mathbf{receive}(\text{bool})$$

With session types, a client looks like:

```
public void client(Session s) { ... }
```

. . . where `s` is a **communication channel** of type "Session"

# Session types enforce channel usage

From the **client viewpoint**, the **session type** is:

$$\text{Session} = \mathbf{send}(\text{int}).\mathbf{send}(\text{int}).\mathbf{receive}(\text{bool})$$

With session types, a client looks like:

```
public void client(Session s) { ... }
```

. . . where `s` is a **communication channel** of type "Session"

The type determines the **intended usage of** `s`: i.e., if the code compiles, then `client` uses `s` to **send two** integers, and then **receive a** boolean

# Session types and duality

From the **client** **viewpoint**, the **session type** is:

$$\text{Session} = \mathbf{send}(\text{int}).\mathbf{send}(\text{int}).\mathbf{receive}(\text{bool})$$

What about the **server**?

# Session types and duality

From the **client** **viewpoint**, the **session type** is:

$$\mathsf{Session} \;=\; \mathbf{send}(\mathsf{int}).\mathbf{send}(\mathsf{int}).\mathbf{receive}(\mathsf{bool})$$

What about the **server**? Its view of the protocol is **dual:**

$$\overline{\mathsf{Session}} \;=\; \mathbf{receive}(\mathsf{int}).\mathbf{receive}(\mathsf{int}).\mathbf{send}(\mathsf{bool})$$

# Session types and duality

From the **client** **viewpoint**, the **session type** is:

$$\mathsf{Session} \;=\; \mathbf{send}(\mathsf{int}).\mathbf{send}(\mathsf{int}).\mathbf{receive}(\mathsf{bool})$$

What about the **server**? Its view of the protocol is **dual:**

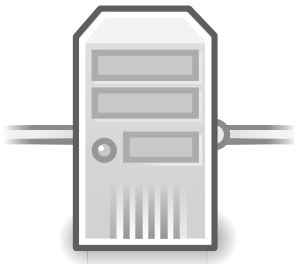
$$\overline{\mathsf{Session}} \;=\; \mathbf{receive}(\mathsf{int}).\mathbf{receive}(\mathsf{int}).\mathbf{send}(\mathsf{bool})$$

. . . and correspondingly, its code uses a dual type:

```
public void server(Session s) { ... }
```

# Session-typed interactions never go wrong

Client

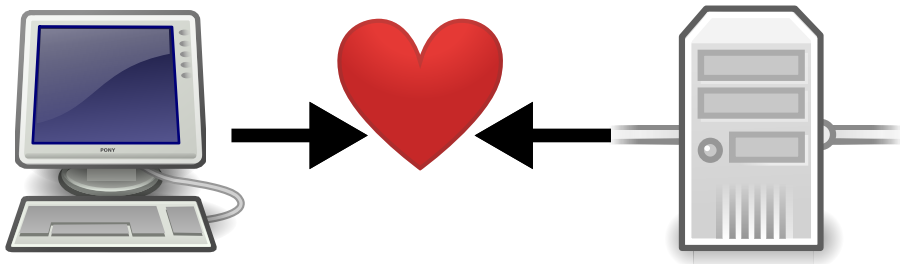


Server

Now, **at compile time**, we can verify that **client** and **server** implement **dual sessions types**...

$$\begin{aligned} \mathsf{Session} &= \mathbf{send}(\mathsf{int}).\mathbf{send}(\mathsf{int}).\mathbf{receive}(\mathsf{bool}) \\ \overline{\mathsf{Session}} &= \mathbf{receive}(\mathsf{int}).\mathbf{receive}(\mathsf{int}).\mathbf{send}(\mathsf{bool}) \end{aligned}$$

# Session-typed interactions never go wrong




Now, **at compile time**, we can verify that **client** and **server** implement **dual sessions types**...

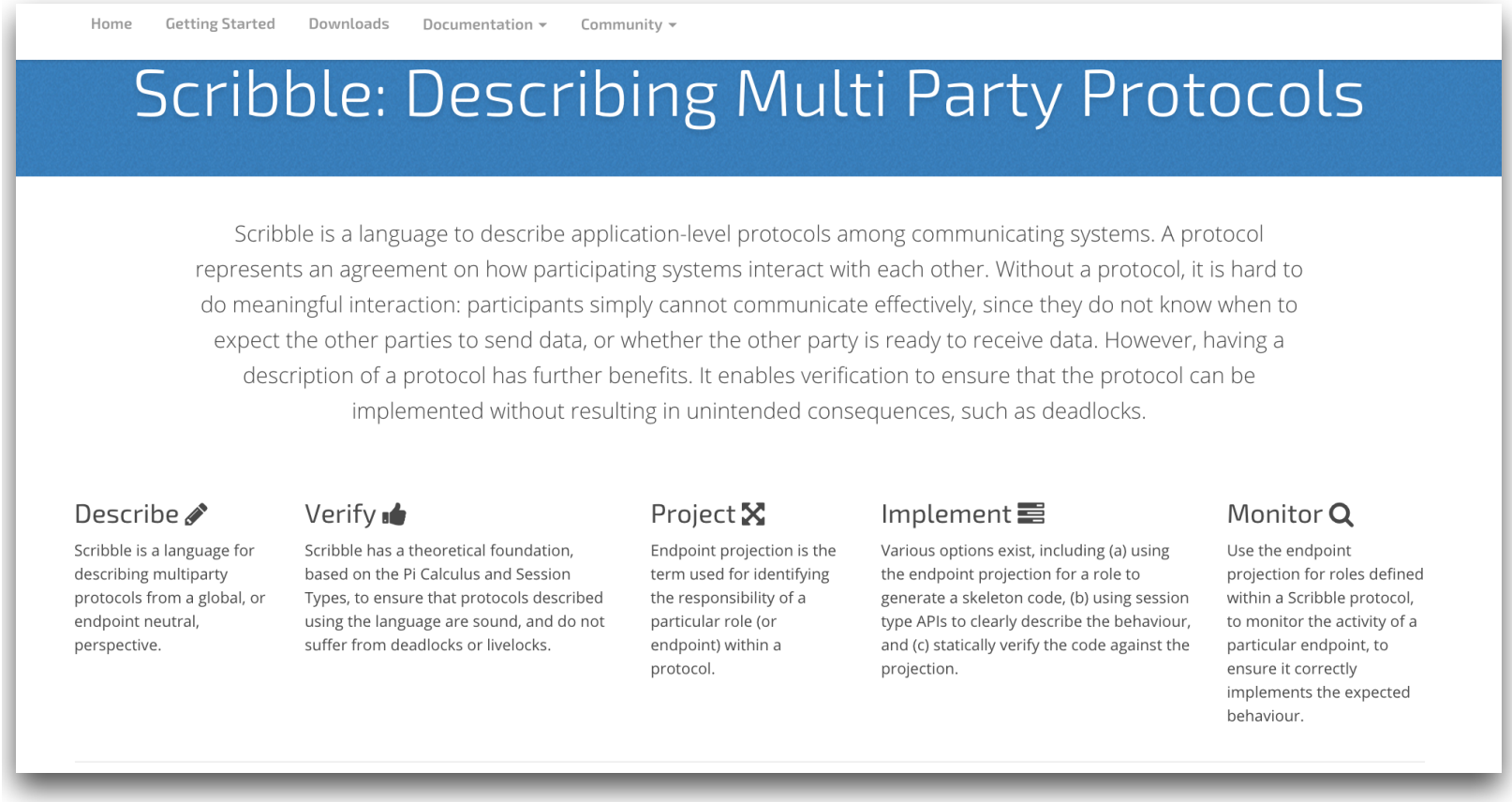
$$\begin{aligned} \text{Session} &= \mathbf{send}(\text{int}).\mathbf{send}(\text{int}).\mathbf{receive}(\text{bool}) \\ \overline{\text{Session}} &= \mathbf{receive}(\text{int}).\mathbf{receive}(\text{int}).\mathbf{send}(\text{bool}) \end{aligned}$$

...and this guarantees **correct interaction at run-time**:

- **communication safety** — no mismatching input/output ✓
- **progress** — no deadlocks ✓

# Scribble

www.scribble.org



# DEMO:

**Scribble** and **RFC 5321**
("Simple" Mail Transfer Protocol)

# Conclusion

Session types allow to:

1. **formalise protocols** for concurrent/distributed systems
2. **verify concurrent programs** at **compile-time**
3. and also **implement automatic run-time monitoring**

This leads to two main uses:

- spot **protocol violations** and **deadlocks** at **compile-time**
- and detect and **report protocol violations** at **run-time**

# Conclusion

Session types allow to:

1. **formalise protocols** for concurrent/distributed systems
2. **verify concurrent programs** at **compile-time**
3. and also **implement automatic run-time monitoring**

This leads to two main uses:

- spot **protocol violations** and **deadlocks** at **compile-time**
- and detect and **report protocol violations** at **run-time**

For papers and more info, visit:

http://mrg.doc.ic.ac.uk/

. . . and get in touch for enquiries and collaboration!

**Questions?**

# Predictable Concurrent & Distributed Systems

David Castro-Perez⟨dcastrop@ic.ac.uk⟩

## Problem

**Estimating execution times** of concurrent & distributed systems.

**Relevance:**

- Parallel speedups.
- Throughput.
- Server response times.

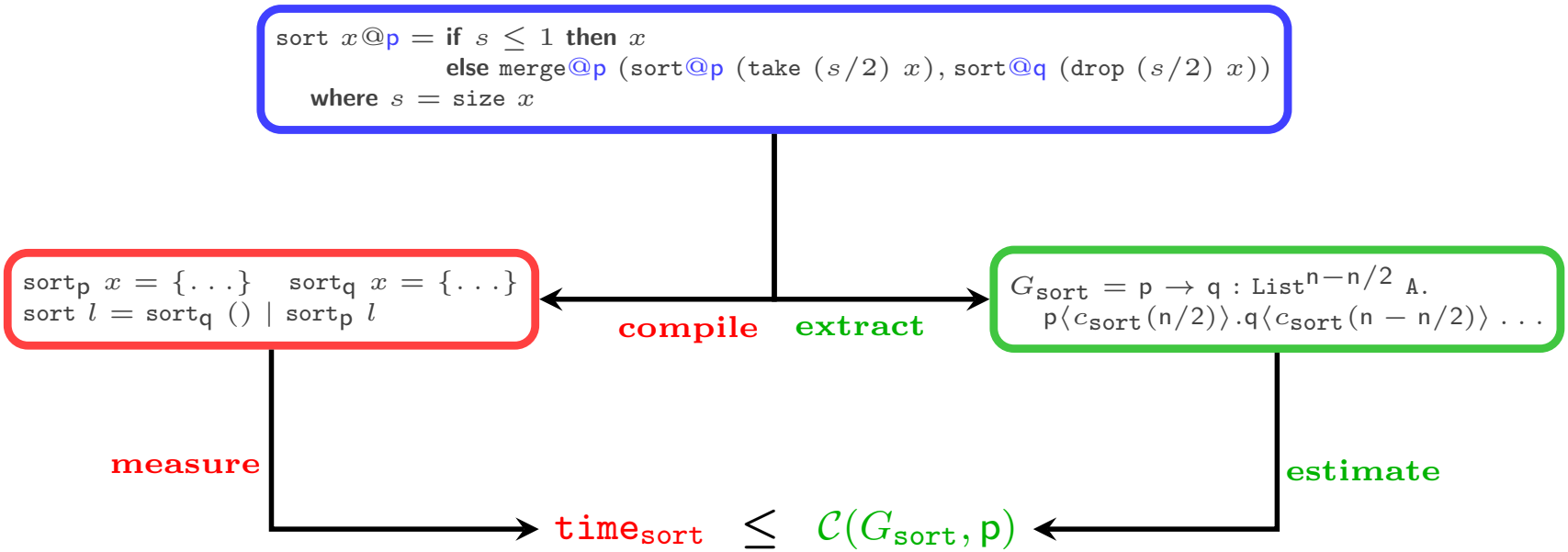
## Our Approach

- Instrument **protocols with cost**.

$$G = \mu X.\ \mathsf{p} \to \mathsf{w}_1 : \mathtt{Int}^{s_1}.\mathsf{w}_1\langle c_1\rangle.\ \mathsf{p} \to \mathsf{w}_2 : \mathtt{Int}^{s_2}.\mathsf{w}_2\langle c_2\rangle.\mathsf{w}_1 \to \mathsf{q} : \mathtt{Int}^{s_3}.\ \mathsf{w}_2 \to \mathsf{q} : \mathtt{Int}^{s_4}.\mathsf{q}\langle c_3\rangle.X$$

- Compute execution-time properties on $G$.

## Application to parallel programming (`https://github.com/dcastrop/SAlg`)



# From sequencing to dependence: CAUSALITY

Simon CASTELLAN

## WHAT?

```
x = 1;        x = 1;
y = 2;        y = x+1;
```

Independence — Dependence

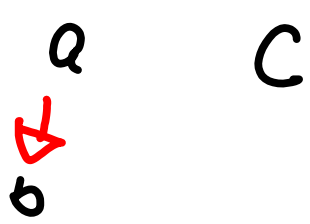
## WHY?

- Compiler optimisations
- Hardware optimisations
- Distributed systems

## HOW?

Program → Partial order

```
a;
b;
c;
```



# From sequencing to dependence: CAUSALITY

Simon CASTELLAN

## WHAT?

```
x = 1;        x = 1;
y = 2;        y = x + 1;
```

Independence — Dependence

## WHY?

- Compiler optimisations
- Hardware optimisations
- Distributed systems

## HOW?

Program → Partial order

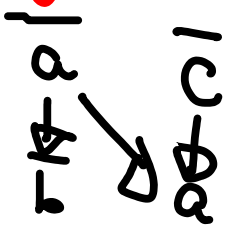
```
a;
b;
c;
```


## APPLICATIONS

### ① Message-passing programs

```
send(a); || send(c);
send(b);    recv(a)
```

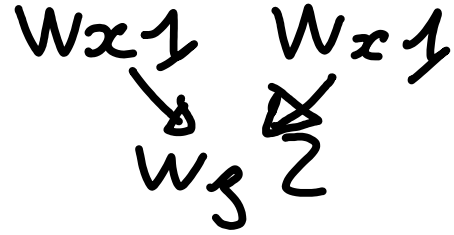
⇝




### ② Out of order execution

```
x := 1
y := 1
z := x + y;
```

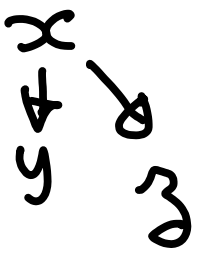
⇝




### ③ Bayesian inference

```
x = sample();
y = sample(x);
z = sample(x)
```

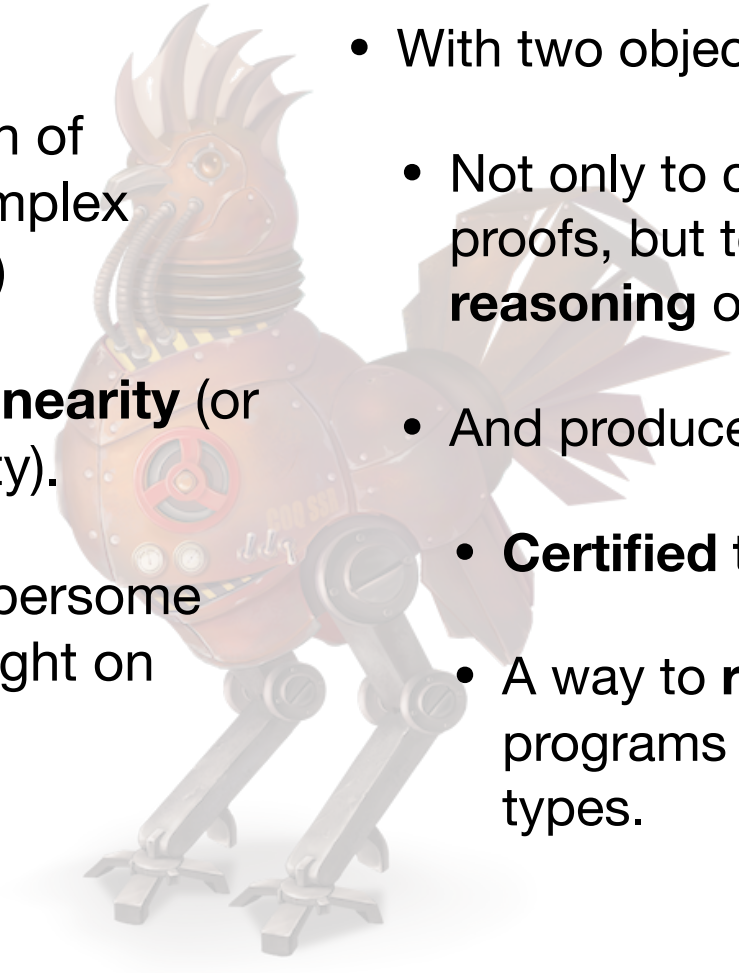
⇝



**Francisco Ferreira**

# Mechanisation and certified code.

- The **meta-theory** of Session Types Calculi is particularly challenging. Due to:
  - A first-class notion of **names** (and a complex binding structure.)
  - The presence of **linearity** (or resource sensibility).
- The proofs are cumbersome and difficult to get right on paper.



- We work on formalising the definitions, and mechanising the meta-theory.
- With two objectives:
  - Not only to certify existing proofs, but to allow for **easier reasoning** of newer proofs.
  - And produce:
    - **Certified tools**
    - A way to **reason** about programs with session types.

# Dependent session types for the HO$\pi$-calculus (Ferreira, **Ruoppolo**, Yoshida)

Dependent session types for the HO$\pi$-calculus ( Ferreira, **Ruoppolo**, Yoshida )

Channels communicating pieces of code

$$a(x:\sigma).P \parallel a\langle V\rangle.Q \quad \rightarrow \quad P\{V/x\} \parallel Q$$

(...plus some functional flavour: $(\lambda x:\sigma.P)Q \rightarrow P\{Q/x\}$)

Dependent session types for the HO$\pi$-calculus ( Ferreira, **Ruoppolo**, Yoshida )

Channels communicating pieces of code

$$a(x:\sigma).P \parallel a\langle \mathsf{V} \rangle.Q \quad \rightarrow \quad P\{\mathsf{V}/x\} \parallel Q$$

(...plus some functional flavour: $(\lambda x:\sigma.P)Q \rightarrow P\{Q/x\}$)

Dependent session types for the HO$\pi$-calculus ( Ferreira, **Ruoppolo**, Yoshida )

Channels communicating pieces of code

$$a(x:\sigma).P \parallel a\langle V \rangle.Q \quad \rightarrow \quad P\{V/x\} \parallel Q$$

(...plus some functional flavour: $(\lambda x:\sigma.P)Q \rightarrow P\{Q/x\}$)

Types as protocols for dual binary communication

$$a\langle \texttt{true} \rangle.b\langle 3 \rangle.0 : \left[ a : \texttt{!bool.end}, b : \texttt{!int.end} \right]$$

# Dependent session types for the HO$\pi$-calculus (Ferreira, **Ruoppolo**, Yoshida)

## Channels communicating pieces of code

$$a(x:\sigma).P \parallel a\langle V \rangle.Q \quad \rightarrow \quad P\{V/x\} \parallel Q$$

(...plus some functional flavour: $(\lambda x:\sigma.P)Q \rightarrow P\{Q/x\}$)

## Types as protocols for dual binary communication

$$a\langle \texttt{true} \rangle.b\langle 3 \rangle.0 : \left[ a : \texttt{!bool.end}, b : \texttt{!int.end} \right]$$

## Types depending and quantifying over terms

Proof certificates: $P : \Pi x : \left[ a : \texttt{!bool.end}, b : \texttt{!int.end} \right]. \mathcal{A}(x)$

Plus a built-in statically type-checkable equality of types!

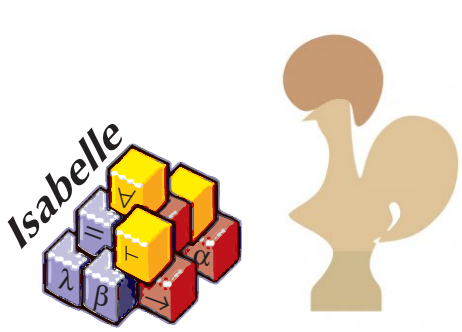
# Syntax with Bindings, Proof Assistants and Other Stories

Modular framework for datatypes with bindings:

- complex variable binders
- infinitary syntax too (including coinductive datatypes)
- capture-avoiding substitution
- (co)induction and (co)recursion
- modularity
- formalized framework in Isabelle/HOL

Proof assistants and mathematical logic
for the verification of
the (meta)theory of programming languages.

Syntax, semantics, type systems, ...



# Eva Graversen: Controlled Reversibility

- *Rollback* reverses a specific action and everything caused by it, not letting the process continue forwards until the roll is complete.
- *Commit* prevents a certain action and all its causes from reversing. This also allows those past actions to be garbage collected.

**Example**

Client $C(n)$ sends out $n$ requests for resources and commits if all of them get received and none rolled back.

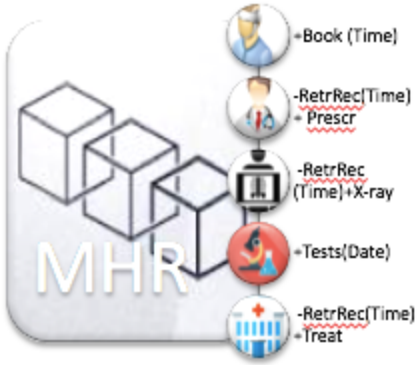
$R(k)$ has $k$ resources to give, and when it gets a request can either roll back the request or wait for the client to commit, at which point it will have $k - 1$ resources to give.

Imperial College London

Assel Altayeva ✉aaltayev@imperial.ac.uk @altasela

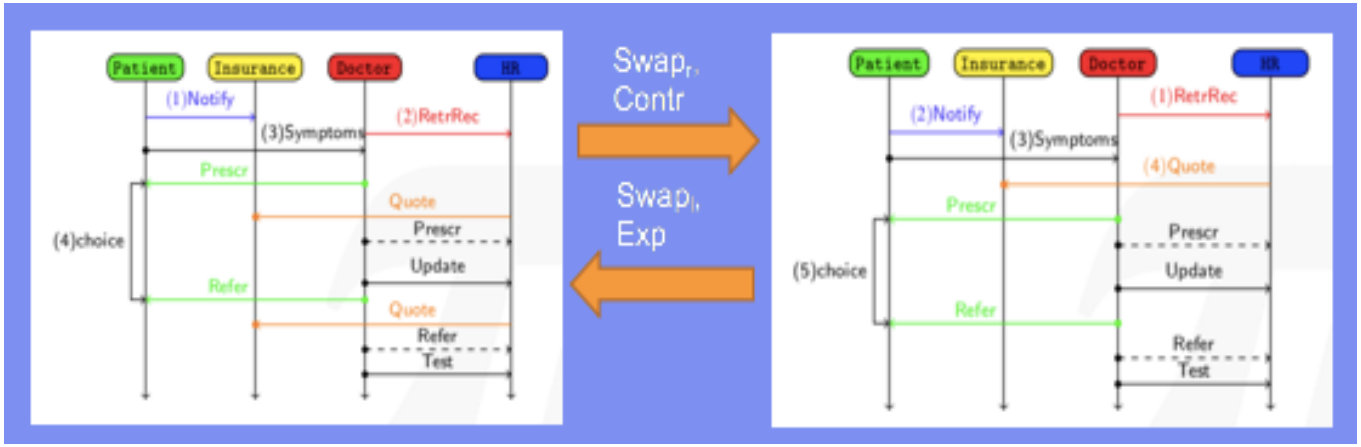
# Service Architecture Equivalence via Multiparty Session Type Isomorphisms

**Approach:**
We axiomatize MPST isomorphism to produce a combinator based calculus for proving isomorphism – an algebra for certification of substitutability.

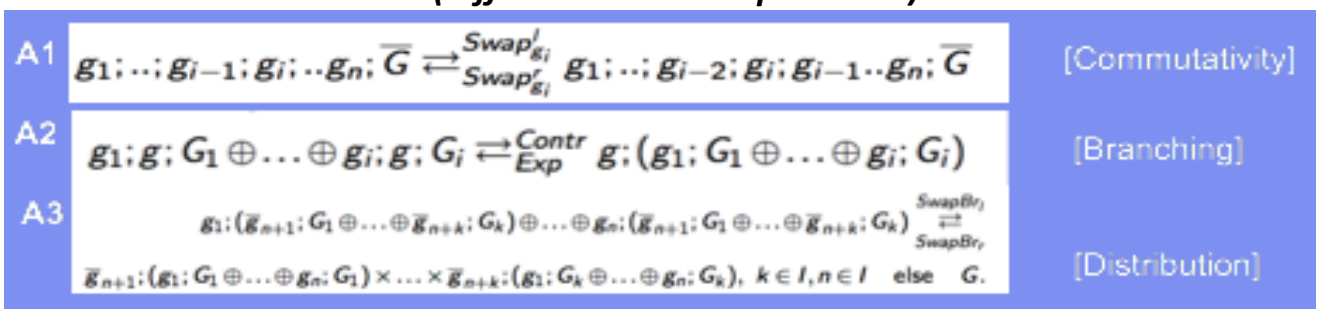



**Research Problem:**
How to achieve certified substitutability of services within a global choreography?

- Axioms of Isomorphism are **complete** with respect to t**race-based semantics.**
- **Functional combinators** over syntactic structure of global types.
- Multiparty Session types with **security levels** for both **participants** and **data.**




***Equivalent modulo MPST isomorphism global choreographies for Health Record (different Insurance providers).***




***Example MPST isomorphism axioms.***

# Data Race Detection in Go Programing

| Application | Behavior | | Cause | |
|---|---|---|---|---|
| | blocking | non-blocking | shared memory | message passing |
| Docker | 21 | 23 | 28 | 16 |
| Kubernetes | 17 | 17 | 20 | 14 |
| etcd | 21 | 16 | 18 | 19 |
| CockroachDB | 12 | 16 | 23 | 5 |
| gRPC | 11 | 12 | 12 | 11 |
| BoltDB | 3 | 2 | 4 | 1 |
| **Total** | 85 | 86 | 105 | 66 |

**Table 5. Taxonomy.** *This table shows how our studied bugs distribute across different categories and applications.*

| Application | Shared Memory | | | | Message Passing | |
|---|---|---|---|---|---|---|
| | traditional | anon. | waitgroup | lib | chan | lib |
| Docker | 9 | 6 | 0 | 1 | 6 | 1 |
| Kubernetes | 8 | 3 | 1 | 0 | 5 | 0 |
| etcd | 9 | 0 | 2 | 2 | 3 | 0 |
| CockroachDB | 10 | 1 | 3 | 2 | 0 | 0 |
| gRPC | 8 | 1 | 0 | 1 | 2 | 0 |
| BoltDB | 2 | 0 | 0 | 0 | 0 | 0 |
| **Total** | 46 | 11 | 6 | 6 | 16 | 1 |

**Table 9. Root causes of non-blocking bugs.** *traditional: traditional non-blocking bugs; anonymous function: non-blocking bugs caused by anonymous function; waitgroup: misusing WaitGroup; lib: Go library; chan: misusing channel.*

```go
func main() {
  var a sync.RWMutex
  var b int
  go goroutine1(&a, &b)
  go goroutine2(&a, &b)
}

func goroutine1(m *sync.RWMutex, x *int) {
  m.Lock()
  *x = 1
  m.Unlock()
  m.RLock()
  print(*x)    // Race with write in goroutine2
  m.RUnlock()
}

func goroutine2(m *sync.RWMutex, x *int) {
  m.RLock()    // bad usage of RLock
  *x = 2       // Race with read in goroutine1
  m.RUnlock()
}
```

```
def t0() : letsync a rwmutex ;
           letmem b;
           spawn t1(a,b);
           spawn t2(a,b);

def t1(m,x) : lock m;
              write x;
              unlock m;
              rlock m;
              read x;
              runlock m;

def t2(m,x): rlock m;
             write x;
             runlock m;
```

Tables from: Tengfei, Xiaoyu, Linhai and Yiying, *Understanding Real-World Concurrency Bugs in Go* (ASPLOS), 2019.

# Combining refinement types and session types

## Refinement Types

Predicate restricts permitted values

$$type\ nat = \{v{:}\ int \mid v \geq 0\}$$

$$1 \in nat? \longrightarrow [\![(v \geq 0)[v/1]]\!] \longrightarrow Valid$$

SMT assisted typechecking

## Multiparty Session Types

Message Passing for multiple parties with types

$$\mathbf{s}\rightarrow\mathbf{c}: \left\{ \begin{array}{l} \texttt{login} \,.\, \mathbf{c}\rightarrow\mathbf{a}{:}\texttt{passwd}(\texttt{Str}) \,.\, \mathbf{a}\rightarrow\mathbf{s}{:}\texttt{auth}(\texttt{Bool}) \,.\, \mathbf{end}\,, \\ \texttt{cancel} \,.\, \mathbf{c}\rightarrow\mathbf{a}{:}\texttt{quit} \,.\, \mathbf{end} \end{array} \right\}$$

Provides guarantees on deadlock freedom and session fidelity

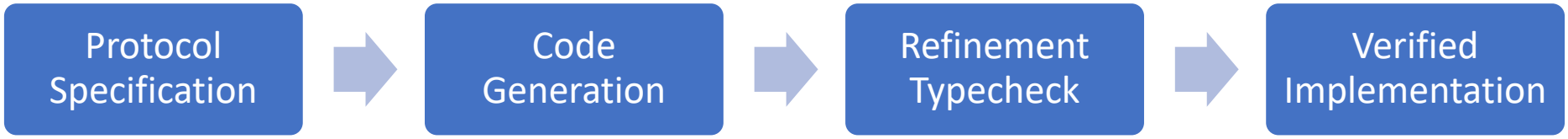
# Our work

Extended Scribble protocol description language **with refinements**

Code Generator converts local types to F* code with stubs

F* type system verifies Implementation correctness

Verified implementation extracted from F*



# Security of Blockchains using Session Types

Nicolas LAGAILLARDIE[1]
Supervisor: Nobuko YOSHIDA[2]
2nd Supervisor: Rumyana NEYKOVA[3]

[1]Imperial College London: PhD
Université de Lyon: MSc Cyber Physical Social-Systems
École des Mines de Saint-Étienne: Diplôme d'Ingénieur

[2]Imperial College London: Professor of Computing

[3]Brunel university London: Lecturer

October 9, 2019

Imperial College London

# Past work

- MSc Cyber Physical Social-Systems
  - Multi-agent programming
  - Internet of Things
  - Cloud computing
  - Semantic web
- Diplôme d'Ingénieur
  - Majors: Data Science and Computer Science
  - Minors: Network, Big Data, High-performance computing
- Professional (Internships)
  - CEA: simulating and improving the Blockchain protocol Tendermint
  - Predisurge: developing a Web tool for 3D visualization

Imperial College London