Distributed Programming using Role-Parametric Session Types in Go

David Castro¹, Raymond Hu¹, Sung-Shik Jongmans¹,², Nicholas Ng¹, Nobuko Yoshida¹

¹ Imperial College London
² Open University of the Netherlands
Distributed Programming using Role-Parametric Session Types in Go

David Castro\textsuperscript{1}, Raymond Hu\textsuperscript{1}, Sung-Shik Jongmans\textsuperscript{1,2}, Nicholas Ng\textsuperscript{1}, Nobuko Yoshida\textsuperscript{1}

\textsuperscript{1} Imperial College London
\textsuperscript{2} Open University of the Netherlands
Distributed Programming using Role-Parametric Session Types in Go

David Castro¹, Raymond Hu¹, Sung-Shik Jongmans¹,², Nicholas Ng¹, Nobuko Yoshida¹

¹ Imperial College London
² Open University of the Netherlands
Introduction (distributed programming in Go)

Long-term research agenda:

Development of theory and tools to help programmers write safe concurrent programs
Introduction (distributed programming in Go)

Long-term research agenda:

Development of theory and tools to help Go programmers write safe concurrent Go programs

[CC’16, POPL’17, ICSE’18]
Introduction (distributed programming in Go)

- (a) Modern, popular systems language
Introduction (distributed programming in Go)

- (a) Modern, popular systems language

- (b) Primacy of CSP-based concurrency features
  - Lightweight threads, called goroutines
  - Higher-order, typed native channels (across shared memory)
  - First-order, untyped API channels (across a network)
Introduction (distributed programming in Go)

(a) Modern, popular systems language

(b) Primacy of CSP-based concurrency features
   - Lightweight threads, called goroutines
   - Higher-order, typed native channels (across shared memory)
   - First-order, untyped API channels (across a network)

(c) Survey: “Users least agreed that they are able to effectively debug uses of Go’s concurrency features”
Introduction (distributed programming in Go)

- (a) Modern, popular systems language

- (b) Primacy of CSP-based concurrency features
  - Lightweight threads, called goroutines
  - Higher-order, typed native channels (across shared memory)
  - First-order, untyped API channels (across a network)

- (c) Survey: “Users least agreed that they are able to effectively debug uses of Go’s concurrency features”

 multiparty session types? [POPL’08]
Introduction (distributed programming in Go)

Motivating example: htcat

(https://github.com/htcat/htcat)

Parallel downloader of webpages

Post-factum verification very difficult

Our safe-by-construction version: PGet (● ●)
Introduction (distributed programming in Go)

Master

Fetchers

Server

feature 1: parameterisation (in #Fetchers)

feature 2: mixed transports & disparate abstractions

feature 3: channel passing

feature 4: heterogeneous roles
**Introduction** (distributed programming in Go)

- Master: \( M \)
- Fetchers: \( F_1, F_2, \ldots, F_n \)
- Server: \( S \)

**feature 1:** parameterisation (in \#Fetchers)
Introduction (distributed programming in Go)

Master

Fetchers

Server

feature 1: parameterisation (in #Fetchers)

Local

Remote

shared memory channel

TCP channel
Introduction (distributed programming in Go)

Master

Fetchers

Server

- feature 1: parameterisation (in #Fetchers)
- feature 2: mixed transports & disparate abstractions
- feature 3: channel passing
- feature 4: heterogeneous roles

Shared memory channel

TCP channel
Introduction (distributed programming in Go)

Master

Fetchers

Server

feature 2: mixed transports & disparate abstractions

M → F₁ : GetSize(string)
**Introduction** (distributed programming in Go)

**Master**

- Local

**Fetchers**

- $F_1$
- $F_2$
- $\ldots$
- $F_n$

**Server**

- Remote

**Feature 2:**
mixed transports & disparate abstractions

$F_1 \rightarrow S: \text{HttpReq}(\text{byte}[])$
Introduction (distributed programming in Go)

Master

Fetchers

Server

feature 2:
mixed transports & disparate abstractions

\[ F_1 \rightarrow S : \text{HttpReq(byte[])} \]
\[ S \rightarrow F_1 : \text{HttpRes(byte[])} \]
Introduction (distributed programming in Go)

Master

Fetchers

Server

feature 2: mixed transports & disparate abstractions

F₁ → M : Size(int)
Introduction (distributed programming in Go)

Master

Fetchers

Server

feature 2:
mixed transports & disparate abstractions

\[ M \rightarrow F[1..n] : \text{GetData}(\text{string}, \text{int}, \text{int}) \]
**Introduction** (distributed programming in Go)

- **Master**
  - Local
  - Fetchers
  - `F_1`, `F_2`, ..., `F_n`
  - Server
    - Remote

- Feature 1: Parameterisation (in #Fetchers)
- Feature 2: Mixed transports & disparate abstractions
- Feature 3: Channel passing
- Feature 4: Heterogeneous roles

F[1..n] → S:HttpReq(byte[])
Introduction (distributed programming in Go)

Master \{ Local \\

Fetchers \{ F_1, F_2, \ldots, F_n \}

Server \{ Remote \}

\[ F[1..n] \rightarrow S: \text{HttpReq(byte[])} \]
\[ S \rightarrow F[1..n]: \text{HttpRes(byte[])} \]
**Introduction** (distributed programming in Go)

**feature 2:** mixed transports & disparate abstractions

Master \( \{ \text{Local} \} \)

Fetchers \( \{ F_1, F_2, \ldots, F_n \} \)

Server \( \{ \text{Remote} \} \)

\( F[1..n] \rightarrow M : \text{Data}(\text{string, chan}) \)
Introduction (distributed programming in Go)

- Master
  - Local
  - Fetchers: $F_1, F_2, \ldots, F_n$
  - Server: $S$

Feature 2: mixed transports & disparate abstractions

$F[1..n] \rightarrow M: \text{Data}(\text{string}, \text{chan})$
Introduction (distributed programming in Go)

Master

Fetchers

Server

F[1..n] → M:Data(string, chan)
Introduction (distributed programming in Go)

Master \{ Local \}

Fetchers \{ \}

Server \{ Remote \}

feature 3: channel passing
Introduction (distributed programming in Go)

- Master
  - Local
  - Fetchers
    - $F_1$
    - $F_2$
    - ... $F_n$
- Server
  - Remote

Features:
1. Parameterisation (in #Fetchers)
2. Mixed transports & disparate abstractions
3. Channel passing
4. Heterogeneous roles

feature 4: heterogeneous roles
**Introduction** (distributed programming in Go)

**Features:**
- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles
Introduction (distributed programming in Go)

Features:
- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles

Challenges (safety):
- Protocol compliance
- Deadlock-freedom
Introduction (distributed programming in Go)

Features:
- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles

Challenges (safety):
- Protocol compliance
- Deadlock-freedom
Introduction (distributed programming in Go)

Features:
- Parameterisation (in #Fetchers)
- Mixed transports & disparate abstractions
- Channel passing
- Heterogeneous roles

Challenges (safety):
- Protocol compliance ✓
- Deadlock-freedom ✓

Real programs need more expressive theory and impl.
Introduction (multiparty session types; MPST)

processes \{ \hat{W_1}, \hat{W_2}, \hat{W_3} \}
**Introduction** (multiparty session types; MPST)

Global type:

\[ G = \]

\[ W_1 \rightarrow W_2 : \text{Int} \]

\[ W_2 \rightarrow W_3 : \text{Bool} \]

Processes:

\[ \{ W_1, W_2, W_3 \} \]
Introduction (multiparty session types; MPST)

Global type \( G \)

Local types \( L_1, L_2, L_3 \)

Processes \( W_1, W_2, W_3 \)

\[ G = \]
\[ W_1 \rightarrow W_2 : \text{Int} . \]
\[ W_2 \rightarrow W_3 : \text{Bool} \]

\[ L_1 = W_2! \text{Int} \]
\[ L_2 = W_1? \text{Int} . \]
\[ W_3! \text{Bool} \]
\[ L_3 = W_2? \text{Bool} \]
Introduction (multiparty session types; MPST)

\[ G = \]
\[ W_1 \rightarrow W_2 : \text{Int}. \]
\[ W_2 \rightarrow W_3 : \text{Bool} \]

\[ L_1 = W_2!\text{Int} \]
\[ L_2 = W_1?\text{Int}. \]
\[ W_3!\text{Bool} \]
\[ L_3 = W_2?\text{Bool} \]
**Introduction**  (multiparty session types; MPST)

well-typed $\Rightarrow$ protocol compliance $\land$ deadlock-freedom

$G =$

$W_1 \to W_2 : \text{Int}$.

$W_2 \to W_3 : \text{Bool}$

$L_1 = W_2!\text{Int}$

$L_2 = W_1?\text{Int}$.

$W_3!\text{Bool}$

$L_3 = W_2?\text{Bool}$

well-typed $\Rightarrow$ protocol compliance $\land$ deadlock-freedom
**Introduction** (multiparty session types; MPST)

Well-typed $\Rightarrow$ protocol compliance $\land$ deadlock-freedom

$G = W_1 \rightarrow W_2 : \text{Int}.$  
$W_2 \rightarrow W_3 : \text{Bool}$

Local types:

$L_1 = W_2!\text{Int}$
$L_2 = W_1?\text{Int}$
$L_3 = W_2?\text{Bool}$

Processes:

$W_1$
$W_2$
$W_3$

Well-typed $\Rightarrow$ protocol compliance $\land$ deadlock-freedom
Contributions

Theory:
- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness
Contributions

Theory:
- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness

Implementation:
- Extension to Scribble [FASE’16, FASE’17]
- Artifact (reusable and available)
Contributions

Theory:
- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness

Implementation:
- Extension to Scribble [FASE’16, FASE’17]
- Artifact (reusable and available)

Evaluation:
- Competitive performance
- Wide applicability
Theory

Easy part:
Parameterisation

\[ G = \text{foreach } W[i:1..n-1, j:2..n] \text{ do } W[i] \rightarrow W[j] : \text{Msg} \]
Theory

Easy part: Parameterisation

\[ G = \text{foreach } W[i:1..n-1, j:2..n] \text{ do } W[i] \rightarrow W[j] : \text{Msg} \]

Hard part: Role heterogeneity

How to infer from \( G \) there exist three role variants? (first Worker; middle Workers; last Worker)
Theory

\[ G = \text{foreach } W[i:1..n-1, j:2..n] \text{ do } W[i] \rightarrow W[j] : \text{Msg} \]

Key insight: Behaviour of Worker \( x \) is determined by the intervals in which \( x \) occurs (i.e., if \( x \) and \( y \) are contained in the same intervals, Workers \( x \) and \( y \) behave the same)
Theory

\[ G = \text{foreach } W[i:1\ldots n-1, j:2\ldots n] \text{ do } W[i] \rightarrow W[j]:\text{Msg} \]

Key insight: Behaviour of Worker \( x \) is determined by the *intervals* in which \( x \) occurs (i.e., if \( x \) and \( y \) are contained in the same intervals, Workers \( x \) and \( y \) behave the same)

\[
\begin{align*}
    x \in 1\ldots n-1 \land x \in 2\ldots n & \implies x \in 2\ldots n-1 \quad \text{(middle Worker)} \\
    x \in 1\ldots n-1 \land x \notin 2\ldots n & \implies x = 1 \quad \text{(first Worker)} \\
    x \notin 1\ldots n-1 \land x \in 2\ldots n & \implies x = n \quad \text{(last Worker)} \\
    x \notin 1\ldots n-1 \land x \notin 2\ldots n & \implies \bot
\end{align*}
\]
1. Infer role variants as triples $r[D, \bar{D}]$, where:
   - $r$ is a role name
   - $D$ is a set of intervals
   - $\bar{D}$ is a set of "co-intervals"
Theory

1. Infer role variants as triples $r[D, \bar{D}]$, where:
   - $r$ is a role name
   - $D$ is a set of intervals
   - $\bar{D}$ is a set of "co-intervals"

2. Project $G$ onto inferred role variants, e.g.:

   $G \upharpoonright W[\{1..n-1, 2..n\}, \emptyset] = W[\text{self-1}]?\text{Msg} . W[\text{self+1}]!\text{Msg}$
   $G \upharpoonright W[\{1..n-1\}, \{2..n\}] = W[\text{self+1}]!\text{Msg}$
   $G \upharpoonright W[\{2..n\}, \{1..n-1\}] = W[\text{self-1}]?\text{Msg}$
Theory

Theorem: Inferring role variants is decidable

Theorem: Checking well-formedness is decidable

Theorem: Projecting well-formed global types is semantics-preserving, i.e., correct
Implementation

Extension of protocol description language **Scribble**

(http://www.scribble.org)

protocol spec → state machines → APIs

- Global type:
  - role variants, well-formedness, and projection (using Z3)
- Local types:
  - (role variant-specific)

Go code generation
Implementation

Extension of protocol description language Scribble
(http://www.scribble.org)

- role variants,
- well-formedness,
- and projection (using Z3)

Go code generation

APIs guide programmer towards safety

- global type
- local types
  - (role variant-specific)
Implementation

Artifacts Evaluated
Reusable

Artifacts Available

(demo video)
Implementation

Guarantees:

- Protocol compliance
- Deadlock-freedom (up to “protocol-unrelated” program behaviour, premature termination, and delegation)

Achieved through:

- Native Go typing
- Lightweight run-time checks for linearity
Evaluation (benchmarks)

- Microbenchmarks
  - Speed-up \(\frac{t_1}{t_2}\) of Scribble \(t_2\) vs. native Go \(t_1\)
  - Per communication: \(\sim 20\text{ns}\)
Evaluation (benchmarks)

- Microbenchmarks
  - Speed-up \((t_1/t_2)\) of *Scribble* \((t_2)\) vs. native Go \((t_1)\)
  - Per communication: \(~20\)ns
- Computer Language Benchmark Games (CLBG)
## Evaluation (expressiveness)

<table>
<thead>
<tr>
<th>Pt</th>
<th>Sc</th>
<th>Ga</th>
<th>FE</th>
<th>Core I/O patterns</th>
<th>Parallel Topologies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1. One-to-Many (§ 6.1)</td>
<td>4. Pipeline (§ 4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2. Many-to-One (§ 6.1)</td>
<td>5. Ring (§ 3; 4)</td>
</tr>
</tbody>
</table>

Above, ○ are possible alt. implementations

|    |    |    |    | 7. Mesh (§ 4) | 8. Fork-Join |

9. Pget² (□ is the difference between the two versions in § 3.2; § 3.3)
10. Vickrey auction (Supplement, § IV.1.2)
12. n-body simulation (based on Ring) [Bejleri et al. 2009]
13. Iterative linear equation solver (based on Mesh) [Ng and Yoshida 2015]
14. k-nucleotide [Gouy 2017] (§ 6.1)
17. Fibonacci [Lange et al. 2017]
18. Quote-Request [Austin et al. 2004; Ng and Yoshida 2015]
19. P2P multiplayer game [Scalas et al. 2017]

Pt: point-to-point; Sc: Scatter; Ga: Gather; FE: Foreach; Pipe: Pipeline; MS: MS choices; PP: PP choices; Rec: Recursion; Del: Delegation

21 patterns, topologies, and applications (each uses various features of our framework)
Conclusion

Also in the paper:
- Branching, selection, recursion, merge
- Implementation
  - Transport independence
  - Linearity checks (Go does not have linear types)

Technical report with all details:

https://www.doc.ic.ac.uk/research/technicalreports/2018/#4
Conclusion

Theory:
- MPST + parameterisation + role heterogeneity
- Proofs of decidability and correctness

Implementation:
- Extension to Scribble
- Artifact (reusable and available)

Evaluation:
- Competitive performance
- Wide applicability