Open Problems of Session Types
Us ∈ Mobility Research Group

Our recent work Fencing off Go: Liveness and Safety for Channel-based Programming was summarised on The Morning Paper blog.

2 Feb 2017
Weizhen passed her viva today, congratulations Dr. Yang!

24 Jan 2017
Mariangela Dezani-Ciancaglini, a long-term collaborator with our group working on Session Types turns 70 today, more details here.

23 Dec 2016
Rumyana passed her viva today.

Selected Publications

2017


- Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida: Fencing off Go: Liveness and Safety for Channel-based Programming. *POPL 2017*.


http://mrg.doc.ic.ac.uk/
• **TCS’16**: Monitoring Networks through Multiparty Session Types. Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, Nobuko Yoshida
• **LMCS’16**: Multiparty Session Actors. Rumyana Neykova, Nobuko Yoshida
• **FMSD’15**: Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, Nobuko Yoshida
• **TGC’13**: The Scribble Protocol Language. Nobuko Yoshida, Raymond Hu, Rumyana Neykova, Nicholas Ng
Scribble: Describing Multi Party Protocols

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

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

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

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

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

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

global protocol HelloWorld(role Me, role World) {
    hello() from Me to World;
    choice at World {
        goodMorning1() from World to Me;
    } or {
        goodMorning1() from World to Me;
    }
}
Interactions with Industries

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

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

Nobuko Yoshida
Imperial College, London

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

Static deadlock detector
Tool developed based on our research
github.com/ringoen/gogo-hunter
• Static (compile-time) detection of onefork
• Help prevent deadlocks
• Ongoing research
• Analyzed common concurrency patterns & open source projects

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

The Golang UK Conference
Interactions with Industries

F#unctional Londoners Meetup Group

6 days ago · 6:30 PM
Session Types with Fahd Abdeljallal

43 Members

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

Distributed Systems vs. Compositionality

Dr. Roland Kuhn
@rolandkuhn — CTO of Actyx

Current State

• behaviors can be composed both sequentially and concurrently
• effects are not yet tracked
• Scribble generator for Scala not yet there
• theoretical work at Imperial College, London (Prof. Nobuko Yoshida & Alceste Scalas)
Go concurrency verification research at DoC grabs headline

A paper by DoC researchers at POPL on Go concurrency verification was featured in a tech blog and generates a buzz outside of the research community.

A paper by researchers at the department was recently featured in the morning paper, a blog by venture capitalist Adrian Colye, which summarises an important, influential, topical or otherwise interesting paper in the field of computer science every weekday in an easily digestible way by non-researchers. On the 2 Feb 2017 issue of the morning paper, it was highlighted as "the true spirit of POPL (Principles of Programming Languages)".
Selected Publications 2016/2017

- **[ECOOP’17]** Alceste Scala, Raymond Hu, Ornella Darda, NY: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming.
- **[COORDINATION’17]** Keigo Imai, NY and Shoji Yuen: Session-ocaml: a session-based library with polarities and lenses.
- **[FoSSaCS’17]** Julien Lange, NY: On the Undecidability of Asynchronous Session Subtyping.
- **[FASE’17]** Raymond Hu, NY: Explicit Connection Actions in Multiparty Session Types.
- **[CC’17]** Rumyana Neykova, NY: Let It Recover: Multiparty Protocol-Induced Recovery.
- **[POPL’17]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY: Fencing off Go: Liveness and Safety for Channel-based Programming.
- **[CC’16]** Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
- **[FASE’16]** Raymond Hu, NY: Hybrid Session Verification through Endpoint API Generation.
- **[TACAS’16]** Julien Lange, NY: Characteristic Formulae for Session Types.
- **[POPL’16]** Dominic Orchard, NY: Effects as sessions, sessions as effects.
Selected Publications 2016/2017

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

- derive theories to practices

- make theories understandable

- meet theoretical challenges (concurrency, distributions)

- communicate people
Behavioural Type-Based Static Verification Framework for Go

Julian Lange  Nicholas Ng  Bernardo Toninho  Nobuko Yoshida
GO programming language @ Google (2009)

- Message-Passing based multicore PL, successor of C
- Do not communicate by shared memory; instead, share memory by communicating
  
  Go Lang Proverb

- Explicit channel-based concurrency
  - Buffered I/O communication channels
  - Lightweight thread spawning — goroutines
  - Selective send/receive

FUN

Dropbox, Netflix, Docker, CoreOS
- Go has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi
  e.g. [ACM Survey, 2016] 185 citations, 6 pages
- Dynamic channel creations, unbounded thread creations, recursions, ...
- Scalable (synchronous/asynchronous), Modular, Refinable
- **GO** has a runtime deadlock detector

- How can we detect partial deadlock and channel errors for realistic programs?

- Use behavioural types in process calculi
  
  e.g. [ACM Survey, 2016] 185 citations, 6 pages

- Dynamic channel creations, unbounded thread creations, recursions, ..

- **Scalable** (synchronous/asynchronous) Modular, Refinable
- Go has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi e.g. [ACM Survey, 2016] 185 citations, 6 pages
- Dynamic, unbounded thread creations, recursions, ...
- Scalable (synchronous/asynchronous) Modular, Refinable
- **GO** has a runtime deadlock detector

- How can we detect partial deadlock and channel errors for realistic programs?

- Use behavioural types in process calculi

  e.g. [ACM Survey, 2016] 185 citations, 6 pages

- Channel creations, unbounded thread creations, recursions, ...

- **Scalable** (synchronous/asynchronous) · Modular, Refinable
- **Go** has a runtime deadlock detector.

- How can we detect partial deadlock and channel errors for **realistic programs**?

- Use **behavioral types** in process calculi.
  e.g. [ACM Survey, 2016] 185 citations, 6 pages.

- Dynamic channel creations, unbounded thread creations,...

- **Scalable** (synchronous/asynchronous) · Modular, **reusable** · **Understandable**
Our Framework

**STEP 1** Extract Behavioural Types
- (Most) Message passing features of **GO**
- Tricky primitives: selection, channel creation

**STEP 2** Check Safety/Liveness of Behavioural Types
- Model-Checking (Finite Control)

**STEP 3**
- Relate Safety/Liveness of Behavioural Types and **GO** Programs
  - 3 Classes [POPL’17]
  - Termination Check
Our Framework

STEP 1 Extract Behavioural Types
- (Most) Message passing features of **GO**
- Tricky primitives: selection, channel creation

STEP 2 Check Safety/Liveness of Behavioural Types
- Model-Checking (Finite Control)

STEP 3
- Relate Safety/Liveness of Behavioural Types and **GO** Programs
  - 3 Classes [POPL’17]
  - Termination Check
Verification framework for Go

Overview

Check safety and liveness

Create input model and formula

(2) Model checking

(3) Termination checking

Transform and verify

Behavioural types

(1) Type inference

SSA IR

Go source code

Address type and process gap

Pass to termination prover
Concurrent in Go

```go
func main() {
    ch, done := make(chan int), make(chan int)
    go send(ch) // Spawn as goroutine.
    go func() {
        for i := 0; i < 2; i++ {
            print("Working...")
        }
    }()
    go recv(ch, done)
    go recv(ch, done) // Who is ch receiving from?
    print("Done:", <-done, <-done) // 2 receivers, 2 replies
}
func send(ch chan int) { ch <- 1 } // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.
```

- Send/receive blocks goroutines if channel full/empty resp.
- Close a channel `close(ch)`
- Guarded choice `select { case <-ch; case <-ch2; }`
Concurrency in Go
Deadlock detection

```go
func main() {
    ch, done := make(chan int), make(chan int)
    go send(ch) // Spawn as goroutine.
    go func() {
        for i := 0; i < 2; i++ {
            print("Working...")
        }
    }()
    go recv(ch, done)
    go recv(ch, done) // Who is ch receiving from?
    print("Done:", <-done, <-done) // 2 receivers, 2 replies
}

func send(ch chan int) { ch <- 1 } // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.
```

Run program:

```
$ go run main.go
fatal error: all goroutines are asleep - deadlock!
```
func main() {
    ch, done := make(chan int), make(chan int)
    go send(ch) // Spawn as goroutine.
    go func() {
        for i := 0; ; i++ { // infinite
            print("Working...")
        }
    }()
    go recv(ch, done)
    go recv(ch, done) // Who is ch receiving from?
    print("Done:", <-done, <-done) // 2 receivers, 2 replies
}
func send(ch chan int) { ch <- 1 } // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.
Concurrency in Go
Deadlock detection

```go
func main() {
    ch, done := make(chan int), make(chan int)
    go send(ch) // Spawn as goroutine.
    go func() {
        for i := 0; ; i++ { // infinite loop
            print("Working...")
        }
    }()
    go recv(ch, done)
    go recv(ch, done) // Who is ch receiving from?
    print("Done:", <-done, <-done) // 2 receivers, 2 replies
}
func send(ch chan int) { ch <- 1 } // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.

Deadlock NOT detected (some goroutines are running)
```
Concurrency in Go

Deadlock detection

- Go has a runtime deadlock detector, panics (crash) if deadlock
- Deadlock if all goroutines are blocked
- Some packages (e.g. `net` for networking) **disables** it

```go
import _ "net" // Load "net" package

func main() {
    ch := make(chan int)
    send(ch)
    print(<-ch)
}

func send(ch chan int) { ch <- 1 }
```

---

Nobuko Yoshida
Open Problems of Session Types

mrg.doc.ic.ac.uk
Concurrency in Go
Deadlock detection

- Go has a runtime deadlock detector, panics (crash) if deadlock
- Deadlock if all goroutines are blocked
- Some packages (e.g. `net` for networking) **disables** it

```go
import _ "net" // Load "net" package

func main() {
    ch := make(chan int)
    send(ch)
    print(<-ch)
}

func send(ch chan int) { ch <- 1 }
```

Deadlock **NOT** detected
Go Programs as Processes

Go Program

\[ P, Q \ := \ \pi; P \]
\[ \pi \ := \ u! \langle e \rangle \ | \ u? (y) \ | \ \tau \]
Go Programs as Processes

Go Program

\[ P, Q \ ::= \ \pi; P \quad \quad \pi \ ::= u!\langle e \rangle \mid u?(y) \mid \tau \mid u; P \]
Go Programs as Processes

Go Program

\[ P, Q := \pi; P \]
\[ | \quad u; P \]
\[ | \quad \{\pi_i; P_i\}_{i \in I} \]

\[ \pi := u!\langle e \rangle | u?(y) | \tau \]
Go Programs as Processes

Go Program

\[
P, Q \quad ::= \quad \pi; P \\
| \quad u; P \\
| \quad \{\pi_i; P_i\}_{i \in I} \\
| \quad e \; P \; Q \\
\]

\[
\pi \quad ::= \quad u!\langle e \rangle \mid u?(y) \mid \tau
\]
Go Programs as Processes

Go Program

\[
P, Q \ := \ \pi; P \\
\quad u; P \\
\quad \{\pi_i; P_i\}_{i \in I} \\
\quad e P Q \\
\quad (y:\sigma); P \\
\]

\[
\pi \ := \ u!\langle e \rangle \mid u?(y) \mid \tau
\]
Go Programs as Processes

Go Program

\[
P, Q ::= \pi; P \quad \quad \pi ::= u!\langle e \rangle \mid u?\langle y \rangle \mid \tau
\]

\[
\quad u; P
\]

\[
\quad \{\pi_i; P_i\}_{i \in I}
\]

\[
\quad e \ P \ Q
\]

\[
\quad (y: \sigma); P
\]

\[
\quad P \mid Q \mid 0 \mid (\nu c)P
\]
Go Programs as Processes

Go Program

\[
\begin{align*}
P, Q & := \pi; P \\
& \quad | \quad u; P \\
& \quad | \quad \{\pi_i; P_i\}_{i \in I} \\
& \quad | \quad e P Q \\
& \quad | \quad (y: \sigma); P \\
& \quad | \quad P \mid Q \mid 0 \mid (\nu c)P \\
& \quad | \quad X(\tilde{e}, \tilde{u}) \\
\end{align*}
\]

\[
\begin{align*}
\pi & := u!\langle e \rangle \mid u?(y) \mid \tau \\
D & := X(\tilde{x}) = P \\
P & := \{D_i\}_{i \in I} P
\end{align*}
\]
# Go Programs as Processes

## Go Program

\[
P, Q \quad ::= \quad \pi; P \quad \pi \quad ::= \quad u!\langle e \rangle \mid u?y \mid \tau
\]

- \( u; P \)
- \( \{\pi_i; P_i\}_{i \in I} \)
- \( e \) \( P \) \( Q \)
- \( (y:\sigma); P \)
- \( P \mid Q \mid 0 \mid (\nu c)P \)
- \( X\langle \tilde{e}, \tilde{u} \rangle \)

\[
D \quad ::= \quad X(\tilde{x}) = P
\]

\[
P \quad ::= \quad \{D_i\}_{i \in I} \quad P
\]
Abstracting Go with Behavioural Types

Types

\[ \alpha \ := \ \bar{u} \mid u \mid \tau \]

\[ T, S \ := \ \alpha; T \mid T \oplus S \mid \&\{\alpha_i; T_i\}_{i \in I} \mid (T \mid S) \mid 0 \]

\[ \mid (\text{new } a)T \mid \text{close } u; T \mid t(\bar{u}) \]

\[ T \ := \ \{t(\bar{y}_i) = T_i\}_{i \in I} \text{ in } S \]

- Types of a CCS-like process calculus
- Abstracts Go concurrency primitives
  - Send/Recv, new (channel), parallel composition (spawn)
  - Go-specific: Close channel, Select (guarded choice)
Channel Safety

- Channel is closed at most once.
- Can only input from a closed channel (default value).
- Others raise an error and crash.
Channel Safety

- Channel is closed at most once.
- Can only input from a closed channel (default value).
- Others raise an error and crash.

\[ P \text{ is channel safe if } P \xrightarrow{*} (\forall a) Q \text{ and } Q \downarrow_{\text{close}(a)} \]

\[ \neg (Q \downarrow_{\text{end}(a)}) \land \neg (Q \downarrow \overline{a}) \]

never closing never send
Migo  Liveness / Safety

- Liveness

All reachable actions are eventually performed

\[ P \text{ is live if } P \xrightarrow{\ast \forall} Q \]

\[ Q \downarrow a \Rightarrow Q \downarrow \tau \text{ at } a \]

\[ Q \downarrow \overline{a} \Rightarrow Q \downarrow \tau \text{ at } a \]
Select

$P_1 = \text{select } \{ a!, b? \}$

$P_2 = \text{select } \{ a!, b? \}$

$R_1 = a?$

if $P$ is live

$P_1$ is live

$P_2$ is not live

$P_2 | R_2$ is Time Out
\[ P_1 = \text{select} \{ a! \mid b? \mid z \mid P \} \]
\[ P_2 = \text{select} \{ a! \mid b? \} \]

Barb \( \downarrow a \)

\[ \pi_i \downarrow A_i \quad \pi_i \rightarrow P_i \downarrow A_i \]

\[ \text{select} \{ \pi_i \mid P_i \rightarrow \downarrow A_i \} \]

\[ P \downarrow a \quad Q \downarrow \bar{a} \]

\[ P \rightarrow Q \downarrow \bar{A_i} \]

Liveness \( Q \downarrow \bar{a} \Rightarrow Q \downarrow z \) at \( a_i \)
Verification framework for Go
Model checking with mCRL2

Generate LTS model and formulae from types

- Finite control (no parallel composition in recursion)
- Properties (formulae for model checker):
  - ✓ Global deadlock
  - ✓ Channel safety (no send/close on closed channel)
  - ✓ Liveness (partial deadlock)
  - ✓ Eventual reception
    - Require additional guarantees
the \( \mu \)-calculus  encoding properties with barbs

Global Deadlock

Channel Safety

Liveness

\[ \wedge a \in C \ (\downarrow a \lor \downarrow \overline{a}) \Rightarrow \langle \alpha \rangle T \]

\[ \wedge a \in C \downarrow \text{close } a \Rightarrow \neg (\downarrow \overline{a} \lor \downarrow \text{close } a) \]

\[ \wedge a \in C \ (\downarrow a \lor \downarrow \overline{a}) \Rightarrow \Phi (\langle a \rangle T) \wedge \]

\[ \wedge \overline{a} \in C^m \downarrow \overline{a} \Rightarrow \Phi (V a \epsilon \overline{a} \langle a \rangle T) \]

[Lange & NY TACAS '17]
Verification framework for Go
Termination checking with KITTeL

- Extracted types do not consider *data* in process
- Type liveness != program liveness
  - Especially when involving iteration
  - Check for loop termination

- Properties:
  ✓ Global deadlock
  ✓ Channel safety (no send/closed close on closed channel)
  ✓ Liveness (partial deadlock)
  ✓ Eventual reception

```go
func main() {
    ch := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            // Does not terminate
        }
        ch <- 1
    }()
    <-ch
}
```

- Type: Live
- Program: NOT live
Tool demo
Conclusion

Verification framework based on **Behavioural Types**

- Behavioural types for Go concurrency
- Infer types from Go source code
- Model check types for safety/liveness
- + termination for iterative Go code

Diagram:

- **Go source code** → **SSA IR** → **Behavioural types**
  - Type inference
  - Transform and verify
  - Model checking
  - Termination checking
Future work

- Extend framework to support more properties
- Unlimited possibilities!
  - Different verification techniques
    - e.g. [POPL'17], Choreography synthesis [CC'15]
  - Different concurrency issues
    - Other synchronisation mechanisms
    - Race conditions