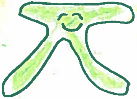
OPEN PROBLEMS of SESSION TYPES



Us ∈ Mobility Research Group

MobilityReadingGroup

π-calculus, Session Types research at Imperial College

Home People Publications Grants Talks Tools Awards Kohei Honda

NEWS

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

Mariangiola 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

Raymond Hu , Nobuko Yoshida : Explicit Connection Actions in Multiparty Session Types. *To appear in* FASE 2017 .

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

Rumyana Neykova , Nobuko Yoshida : Let It Recover: Multiparty Protocol-Induced Recovery. CC 2017 .

Julien Lange , Nobuko Yoshida : On the Undecidability of Asynchronous Session Subtyping. *To appear in* FoSSaCS 2017 .

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

Academic Staff

- Nobuko Yoshida

Research Associate

- Raymond Hu
- Julien Lange
- Nicholas Ng
- Xinyu Niu
- Alceste Scalas
- Bernardo Toninho

PhD Student

- Assel Altayeva
- Juliana Franco
- Rumyana Neykova
- Weizhen Yang

OOI Collaboration

- **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

www.scribble.org

Home Getting Started Downloads Documentation Community

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.

Online tool : http://scribble.doc.ic.ac.uk/

```
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;
  }
}

```

Load a sample | Check | Protocol: examples.HelloWorld | Role: Me | Project | Generate Graph

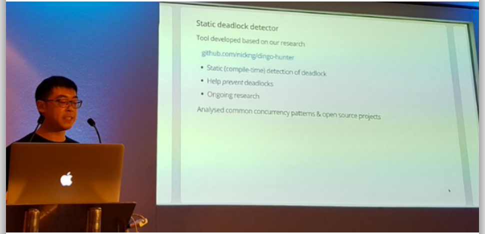
Interactions with Industries

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 #pwlconf, I want to learn more.

Nobuko Yoshida
Imperial College, London

DoC researcher to speak at Golang UK conference

by Vicky Kapogianni
20 July 2016



DoC researcher to speak at industry-focused Golang UK conference on results of concurrency research

Click here to add content

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

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*

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, Ornela 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.
- **[FPL'16]** Xinyu Niu , Nicholas Ng , Tomofumi Yuki , Shaojun Wang , NY, Wayne Luk : EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits.
- **[ECOOP'16]** Alceste Scala, NY: Lightweight Session Programming in Scala
- **[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.
- **[ESOP'16]** Dimitrios Kouzapas, Jorge A. Pérez, NY: On the Relative Expressiveness of Higher-Order Session Processes.
- **[POPL'16]** Dominic Orchard, NY: Effects as sessions, sessions as effects .

Selected Publications 2016/2017

- [ECOOP'17] Alceste Scala, Raymond Hu, Ornela 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.
- [FPL'16] Xinyu Niu , Nicholas Ng , Tomofumi Yuki , Shaojun Wang , NY, Wayne Luk: EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits.
- [ECOOP'16] Alceste Scala, NY: Lightweight Session Programming in Scala
- [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.
- [ESOP'16] Dimitrios Kouzapas, Jorge A. Pérez, NY: On the Relative Expressiveness of Higher-Order Session Processes.
- [POPL'16] Dominic Orchard, NY: Effects as Sessions, Sessions as Effects.

HOW to

- deriver theories to practices
- make theories understandable
- meet theoretical challenges (concurrency · distributions)
- communicate people

Behavioural Type-Based Static Verification Framework for

Julien 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 – gorounines
  - Selective send / receive

CSP 80'

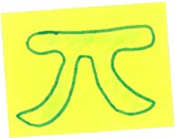
FUN

Dropbox, Netfix, 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

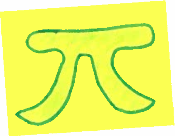
- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi



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

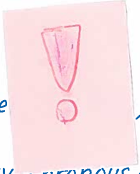
- Dynamic channel creations, unbounded thread creations, recursions,..
- Scalable (synchronous/asynchronous) Modular, Refinable

- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?



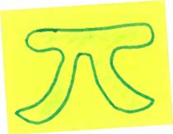
- Use behavioural types in process calculi

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



- Dyn ne , 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

186 ??

- ... hannel 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?

π

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

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

- Scalable (synchronous/asynchronous) Modular, Refinable

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)

GAP

STEP 3

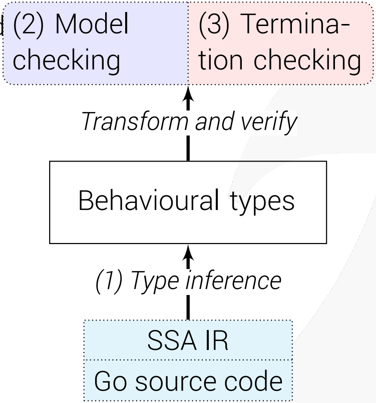
- Relate Safety / Liveness of Behavioural Types and GO Programs
  - 3 Classes [POPL'17]
  - Termination Check

Verification framework for Go

Overview

Check safety and liveness

Create input model and formula



Address type and process gap

Pass to termination prover

Concurrency in 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

```
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!
```

Concurrency in Go

Deadlock detection

```
func main() {
      ch, done := make(chan int), make(chan int)
      go send(ch) // Spawn as goroutine.
      go func() {
            for i := 0; ;  i++ { // infinit
                  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.
```

Change to infinite

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++ { // infinit
                  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.
```

Change to infinite

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

```
import _ "net"  // Load "net" pa
func main() {
      ch := make(chan int)
      send(ch)
      print(<-ch)
}
func send(ch chan int) { ch <- 1 }
```

Add benign import

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

```
import _ "net"  // Load "net" pa
func main() {
      ch := make(chan int)
      send(ch)
      print(<-ch)
}
func send(ch chan int) { ch <- 1 }
```

Add benign import

Deadlock **NOT** detected

Go Programs as Processes

Go Program

$$P, Q \quad := \quad \pi; P \qquad\qquad \pi := u!\langle e\rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$
\begin{array}{rcl}
P, Q & := & \pi; P \\
 & \mid & u; P
\end{array}
\qquad
\pi := u!\langle e\rangle \mid u?(y) \mid \tau
$$

Go Programs as Processes

Go Program

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

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$
\begin{array}{rcl}
P, Q & := & \pi; P \\
& \mid & u; P \\
& \mid & \{\pi_i; P_i\}_{i \in I} \\
& \mid & e\ P\ Q
\end{array}
\qquad
\pi := u!\langle e\rangle \mid u?(y) \mid \tau
$$

Go Programs as Processes

Go Program

$$
\begin{array}{rcl}
P, Q & := & \pi; P \\
& \mid & u; P \\
& \mid & \{\pi_i; P_i\}_{i \in I} \\
& \mid & e\ P\ Q \\
& \mid & \color{red}{(y{:}\sigma); P}
\end{array}
$$

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$
\begin{array}{rcl}
P, Q & := & \pi; P \\
 & \mid & u; P \\
 & \mid & \{\pi_i; P_i\}_{i \in I} \\
 & \mid & e\ P\ Q \\
 & \mid & (y{:}\sigma); P \\
 & \mid & P \mid Q \mid \mathbf{0} \mid (\nu c)P
\end{array}
\qquad
\pi := u!\langle e\rangle \mid u?(y) \mid \tau
$$

Go Programs as Processes

Go Program

$$
\begin{array}{lcl}
P, Q & := & \pi; P \\
& \mid & u; P \\
& \mid & \{\pi_i; P_i\}_{i \in I} \\
& \mid & e\ P\ Q \\
& \mid & (y{:}\sigma); P \\
& \mid & P \mid Q \mid \mathbf{0} \mid (\nu c)P \\
& \mid & X\langle \tilde{e}, \tilde{u} \rangle \\
D & := & X(\tilde{x}) = P \\
\mathbf{P} & := & \{D_i\}_{i \in I}\ P
\end{array}
$$

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$
\begin{array}{lcl}
P, Q & := & \pi; P \\
 & \mid & u; P \\
 & \mid & \{\pi_i; P_i\}_{i \in I} \\
 & \mid & e\ P\ Q \\
 & \mid & (y{:}\sigma); P \\
 & \mid & P \mid Q \mid \mathbf{0} \mid (\nu c)P \\
 & \mid & X\langle \tilde{e}, \tilde{u} \rangle \\
D & := & X(\tilde{x}) = P \\
\mathbf{P} & := & \{D_i\}_{i \in I}\ P
\end{array}
$$

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Abstracting Go with Behavioural Types

Types

$$
\begin{array}{rcl}
\alpha & := & \overline{u} \mid u \mid \tau \\
T, S & := & \alpha; T \mid T \oplus S \mid \&\{\alpha_i; T_i\}_{i \in I} \mid (T \mid S) \mid \mathbf{0} \\
 & \mid & (\mathbf{new}\, a)T \mid \mathtt{close}\, u; T \mid \mathbf{t}\langle \tilde{u} \rangle \\
\mathbf{T} & := & \{\mathbf{t}(\tilde{y}_i) = T_i\}_{i \in I} \ \mathtt{in}\ S
\end{array}
$$

- 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)

MiGo Liveness / Safety

$P \Downarrow a$

Barb [Milner & Sangiorgi 92]

Channel Safety

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

MiGo Liveness / Safety

$P \Downarrow a$

Barb [Milner & Sangiorgi 92]

Channel Safety

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

P is **channel safe** if $P \xrightarrow{}^{*} (\nu \tilde{c}) Q$ and $Q \downarrow close(a)$

a closed

$\neg (Q \downarrow end(a)) \wedge \neg (Q \Downarrow \bar{a} \quad)$

never closing

never send

Migo Liveness / Safety

- **Liveness**

  All reachable actions are eventually performed

$$P \text{ is live if } P \rightarrow^{*}_{(\nu\tilde{c})} Q$$

$$Q \downarrow a \Rightarrow Q \Downarrow \tau \text{ at } a$$

$$Q \downarrow \bar{a} \Rightarrow Q \Downarrow \tau \text{ at } a$$

Reduction (tau) at a

Select

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

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

$R_1 = a?$

Time Out

if P is live

P_1 is live

P_2 is <u>not</u> live

$P_2 | R_2$ is

Select

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

Time Out

if P is live P_1 is live

$P_2 = \text{select}\ \{ a!,\ b? \}$ $\qquad R_1 = a?$

P_2 is <u>not</u> live

$P_2 \mid R_2$ is

Barb $\downarrow \tilde{a}$

$$\frac{\pi_i \downarrow a_i}{\text{select}\ \{\pi_i .\ P_i\} \downarrow \tilde{a}} \qquad \frac{P \downarrow \tilde{a} \qquad Q \downarrow \bar{a_i}}{P \mid Q \downarrow [a_i]}$$

Liveness $\quad Q \downarrow \tilde{a} \Rightarrow Q \Downarrow z$ 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 μ-calculus — encoding properties with barbs

Global Deadlock

$$\bigwedge_{a \in C} (\downarrow a \vee \downarrow \bar{a}) \Rightarrow \langle \alpha \rangle \mathrm{T}$$

Channel Safety

$$\bigwedge_{a \in C} \downarrow \mathit{close}\ a \Rightarrow \neg(\downarrow \bar{a} \vee \downarrow \mathit{close}\ a)$$

Liveness

$$\bigwedge_{a \in C} (\downarrow a \vee \downarrow \bar{a}) \Rightarrow \Phi(\langle [a] \rangle \mathrm{T}) \wedge$$

$$\bigwedge_{\tilde{a} \in C^n} \downarrow \tilde{a} \Rightarrow \Phi\left(\bigvee_{a \in \tilde{a}} \langle [a] \rangle \mathrm{T}\right)$$

[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/`close` on closed channel)
  - ✓ Liveness (partial deadlock)
  - ✓ Eventual reception

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

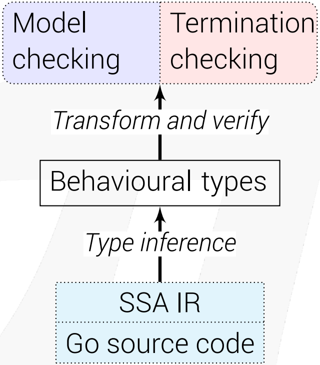
- Type: Live
- Program: NOT live

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



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