Behavioural Type-Based Static Verification Framework for GO

Julian Lange Nicholas Net
Bernardo Toninho Nobuko Yoshida
Language primitives and type discipline for structured communication-based programming by Kohei Honda, Vasco T. Vasconcelos and Makoto Kubo
POPL 2008 MOST INFLUENTIAL PAPER AWARD

POPL 2008 Most Influential Paper Award
Kohei Honda, Nobuko Yoshida and Marco Carbone

Multiparty asynchronous session types
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.
http://mrg.doc.ic.ac.uk

Mobility Research Group

NEWS

SELECTED PUBLICATIONS

2018


Post-docs:
Simon CASTELLAN
David CASTRO
Francisco FERREIRA
Raymond HU
Rumyana NEYKOVA
Nicholas NG
Alceste SCALAS

PhD Students:
Assel ALTAYEVA
Juliana FRANCO
Eva GRAVERSEN
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.
• **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
End-to-End Switching Programme by DCC

1. All design work takes place in ABACUS, DCC’s enterprise architecture tool. This can export standard XMI files (an open standard for UML5).

2. XMI is converted into OpenTracing format for consumption by managed service.

3. OpenTracing files are combined to build a model in Scribble.

4. Model holds types rather than instances to understand behaviour.

5. Scribble compiler identifies inconsistency, change & design flaws.

6. Issues highlighted graphically in Eclipse.

7. Generate exception report and send back to DCC.
Interactions with Industries

**F#unctional Londoners Meetup**

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...

**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)
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 Abdeljalal
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 over the refined protocol, up to premature termination.

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
Selected Publications [2018-2019]

• [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
• [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]

- [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] 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
- [CC18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#
Behavioural Type-Based Static Verification Framework for Go
A static verification framework for message passing in Go using behavioural types

JANUARY 25, 2018

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!
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, ad-hoc, 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 behavioural types in process calculi

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

- Dynamic channel creations, unbounded thread creation...

- Scalable (synchronous/asynchronous) Modular, verifiable

  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
Static verification framework for Go 🐼

Overview

1. Type inference
2. Model checking
   - mCRL2 model checker
   - Check safety and liveness
3. Termination checking
   - KITTeL termination prover
   - Address type ↔ program gap

Behavioural Types

Go source code

SSA IR

Transform and verify
Concurrent in Go 🦖

Goroutines

```
func main() {
    ch := make(chan string)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```

- `go` keyword + function call
- Spawns function as goroutine
- Runs in parallel to parent
Concurrent in Go

Channels

```go
func main() {
    ch := make(chan string)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```

- **Create new channel**
  - Synchronous by default

- **Receive from channel**

- **Close a channel**
  - No more values sent to it
  - Can only close once

- **Send to channel**
Concurrency in Go 🌊

Channels

```go
func main() {
    ch := make(chan string)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```

Also `select-case`:
- Wait on multiple channel operations
- `switch-case` for communication
Concurrency in Go 🧟

Deadlock detection

```go
func main() {
    ch := make(chan string)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```

- Send message thru channel
- Print message on screen

Output:

```
$ go run hello.go
Hello Kent!
$ 
```
**Concurrency in Go 🥷**

**Deadlock detection**

- Only one (main) goroutine
- Send without receive - blocks

Output:

```
$ go run deadlock.go
fatal error: all goroutines are asleep - deadlock!
```

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

*A Static Verification Framework for Message Passing in Go using Behavioural Types*

mrg.doc.ic.ac.uk
Concurrency in Go 🚪

Deadlock detection

Go’s runtime deadlock detector

- Checks if **all** goroutines are blocked (‘global’ deadlock)
- Print message then crash
- Some packages disable it (e.g. net)

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

A Static Verification Framework for Message Passing in Go using Behavioural Types

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

Deadlock detection

```go
import _ "net" // unused

func main() {
    ch := make(chan string)
    send(ch) // Oops
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent"
}
```

- Missing 'go' keyword
- Import unused, unrelated package
Concurrency in Go 🧢

Deadlock detection

- Only one (main) goroutine
- Send without receive - blocks

Output:

```
$ go run deadlock2.go
Hangs: Deadlock NOT detected
```

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
Our goal

Check liveness/safety properties in addition to global deadlocks

- Apply process calculi techniques to Go
- Use model checking to statically analyse Go programs
Concurrency in Go

Behavioural type inference

Abstract Go communication as Behavioural Types

1. Type inference
   SSA IR
   Go source code

2. Model checking
   mCRL2 model checker
   Check safety and liveness

3. Termination checking
   KITTeL termination prover
   Address type ↔ program gap

Summary

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types

mrg.doc.ic.ac.uk
Infer Behavioural Types from Go program

Go source code

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

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

Behavioural Types

Types of CCS-like [Milner ’80] process calculus

- Send/Receive
- new (channel)
- parallel composition (spawn)

Go-specific

- Close channel
- Select (guarded choice)
Infer Behavioural Types from Go program

Go source code

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

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

Inferred Behavioural Types

```
main() = (new ch); (send(ch) | ch; close ch),
```

```
send(ch) = ch
```
Infer Behavioural Types from Go program

Go source code

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

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

Inferred Behavioural Types

```
main() = (new ch);
spawn (send[ch] | ch;
receive close ch),
send(ch) = ch
```

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
mrg.doc.ic.ac.uk
Infer Behavioural Types from Go program

```go
func main() {
    ch := make(chan int) // Create channel
    go sendFn(ch)        // Run as goroutine
    x := recvVal(ch)     // Function call
    for i := 0; i < x; i++ {
        print(i)
    }
    close(ch) // Close channel
}

func sendFn(c chan int) { c <- 3 } // Send to c
func recvVal(c chan int) int { return <-c } // Recv from c
```

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

A Static Verification Framework for Message Passing in Go using Behavioural Types

mrg.doc.ic.ac.uk 27/46
Infer Behavioural Types from Go program

package main

func main.main()

entry

0

t0 = make chan int 0:int
go sendFn(t0)
t1 = recvVal(t0)
jump 3

3

t5 = phi[0: 0:int, 1: t3] #i
t6 = t5 < t1
if t6 goto 1 else 2

1

t2 = print(t5)
t3 = t5 + 1:int
jump 3

2

t4 = close(t0)
return

func main.sendFn(c)

entry

0

send c <- 42:int
return

func main.recvVal(c)

entry

0

t0 = <-c
return

SSA representation of input program

- Only inspect communication primitives
- Distinguish between unique channels
Model checking behavioural types

From behavioural types to model and property specification

1. Type inference

Behavioural Types

SSA IR
Go source code

Transform and verify

Model checking

mCRL2 model checker
Check safety and liveness

Termination checking

KITTeL termination prover
Address type ↔ program gap

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
Model checking behavioural types

\[ M \models \phi \]

- **LTS model**: inferred type + type semantics
- **Safety/liveness properties**: $\mu$-calculus formulae for LTS
- Check with mCRL2 model checker
  - mCRL2 constraint: *Finite control* (no spawning in loops)
- Global deadlock freedom
- Channel safety (no send/c\texttt{lose} on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Behavioural Types as **LTS model**

Standard CS semantics, i.e.

\[ \overline{a}; T \xrightarrow{a} T \]

**Send** on channel \( a \)

\[ T \mid S \xrightarrow{\tau a} T' \mid S' \]

**Synchronise** on \( a \)

\[ a; T \xrightarrow{a} T \]

**Receive** on channel \( a \)
Behavioural Types as \textbf{LTS model}

Standard CS semantics, i.e.

\[ \overline{a}; T \xrightarrow{a} T \]

\[ T \rightarrow T' \quad S \xrightarrow{a} S' \]

\[ T \mid S \xrightarrow{\tau_a} T' \mid S' \]

- Send on channel \( a \)
- Synchronise on \( a \)
- Receive on channel \( a \)
Specifying **properties** of model

**Barbs** (predicates at each state) describe property at state

- Concept from process calculi [Milner ’88, Sangiorgi ’92]
- µ-calculus **properties** specified in terms of barbs

**Barbs** \( T \Downarrow_o \)

- Predicates of state/type \( T \)
- Holds when \( T \) is ready to fire action \( o \)
Specifying properties of model

\[ \bar{a}; \ T \downarrow \bar{a} \]

\[ \frac{T \downarrow \bar{a} \quad T' \downarrow a}{T \mid T' \downarrow_{\tau_a}} \]

Ready to send \hspace{1cm} Ready to synchronise \hspace{1cm} Ready to receive

Barbs \( T \downarrow_o \)

- Predicates of state/type \( T \)
- Holds when \( T \) is ready to fire action \( o \)
Specifying **properties** of model

\[
\bar{a}; \ T \downarrow \bar{a} \quad \frac{\ T \downarrow \bar{a} \quad \ T' \downarrow a}{\ T \mid \ T' \downarrow \tau a} \quad a; \ T \downarrow a
\]

Ready to **send**  Ready to **synchronise**  Ready to **receive**

### Barbs \((T \downarrow o)\)
- Predicates of state/type \(T\)
- Holds when \(T\) is ready to fire action \(o\)
Mi Go  Liveness / Safety

Channel Safety

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

Barb [Milner 8 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.

\[ P \text{ is channel safe if } P \xrightarrow{*} (v \bar{c}) Q \text{ and } Q \downarrow \text{close}(a) \]

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

never closing never send
Migo Liveness / Safety

- Liveness

All reachable actions are eventually performed

\[ P \text{ is live if } P \rightarrow^{*} (\forall c) Q \]

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

\[ Q \downarrow \bar{a} \Rightarrow Q \downarrow z \text{ at } a \]

Reduction (tau) at a
Select

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

\[ P_2 = \text{select } \{ a!, b? \} \quad R_1 = a? \]
Select

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

\[ P_2 = \text{select } \{ a!, b? \} \quad R_1 = a? \]

Time Out
if \( P \) is live
\( P_1 \) is live
Select

$p_1 = \text{select } \{a!, b?, z. P\}$

$p_2 = \text{select } \{a!, b?\}$

$R_1 = a?$

Time Out

if $P$ is live

if $P_1$ is live

$P_2$ is not live

$P_2 | R_2$ is
Select

\[ P_1 = \text{select \{a!, b?, z \cdot P\}} \]

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

Barb \Downarrow \tilde{a}

\[ \pi_1 \Downarrow Q \tilde{z} \]

\[ \text{select \{\pi_1, P_i\}} \Downarrow \tilde{a} \]

\[ P \Downarrow \tilde{a}, Q \Downarrow \tilde{a} \]

\[ P \mid Q \Downarrow [\tilde{a}] \]

Liveness \[ Q \Downarrow \tilde{a} \Rightarrow Q \Downarrow z \text{ at } a_i \]

Time Out

if \( P \) is live

\( P_1 \) is live

\( P_2 \) is not live

\( P_2 \mid R_2 \) is
Specifying **properties** of model

Given

- **LTS model** from inferred behavioural types
- **Barbs** of the LTS model

Express **safety/liveness properties**

- As $\mu$-calculus formulae
- In terms of the **model** and the **barbs**

- Global deadlock freedom
- Channel safety (no send/\textit{close} on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Property: Global deadlock freedom

\[ \left( \bigwedge_{a \in A} \downarrow a \lor \downarrow \bar{a} \right) \implies \langle A \rangle \text{true} \]

If a channel \( a \) is ready to receive or send, then there must be a next state (i.e. not stuck)

\[ A = \text{set of all initialised channels} \quad \bar{A} = \text{set of all labels} \]

\[ \Rightarrow \text{Ready receive/send} = \text{not end of program.} \]
Property: Global deadlock freedom

\[(\bigwedge_{a \in A} \downarrow a \lor \downarrow \overline{a}) \implies \langle A \rangle \text{true}\]

```go
import _ "net" // unused

func main() {
    ch := make(chan string)
    send(ch)  // Oops
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent"
}
```

- Send (\(\downarrow_{\text{ch}}\): line 10)
- No synchronisation
- No more reduction
Property: Channel safety

\[
\left( \bigwedge_{a \in \mathcal{A}} \downarrow a^* \right) \implies \neg (\downarrow \overline{a} \lor \downarrow\text{clo } a)
\]

Once a channel \( a \) is closed \( (a^*) \), it will not be sent to, nor closed again \( (\text{clo } a) \).
Property: Channel safety

\[
(\bigwedge_{a \in A} \downarrow a^\ast) \implies \neg (\downarrow \overline{a} \lor \downarrow \text{clo } a)
\]
Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

\[
(\forall a \in A \left( \downarrow a \lor \downarrow \neg a \right) \implies \text{eventually } \left( \langle \tau_a \rangle \text{true} \right))
\]

If a channel is ready to receive or send, then \textit{eventually} it can synchronise \((\tau_a)\)

(i.e. there's corresponding send for receiver/recv for sender)
Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

\[ \left( \bigwedge_{a \in A} \downarrow a \lor \downarrow \bar{a} \right) \implies \text{eventually} \left( \langle \tau_a \rangle \text{true} \right) \]

where:

\[ \text{eventually} \left( \phi \right) \overset{\text{def}}{=} \mu y. \left( \phi \lor \langle A \rangle y \right) \]

If a channel is ready to receive or send, then for some reachable state it can synchronise \( \tau_a \)
Property: Liveness (partial deadlock freedom)

Liveness for Select

\[ \left( \bigwedge_{\tilde{a} \in \mathcal{P}(A)} \downarrow \tilde{a} \right) \implies \text{eventually} \left( \langle \{ \tau_a \mid a \in \tilde{a} \} \rangle \text{true} \right) \]

If one of the channels in `select` is ready to `receive` or `send`, then `eventually` it will synchronise `\( \tau_a \)`.
Property: Liveness (partial deadlock freedom)

Liveness for Select

\[
\left( \bigwedge_{\tilde{a} \in \mathcal{P}(A)} \downarrow \tilde{a} \right) \implies \text{eventually } \left( \langle \{ \tau_a \mid a \in \tilde{a} \} \rangle \text{true} \right)
\]

\[
P_1 = \text{select}\{\bar{a}, b, \tau.P\}
\]

\[
P_2 = \text{select}\{\bar{a}, b\}
\]

\[
R_1 = a
\]

\[P_1\text{ is live if } P \text{ is } \checkmark\]

\[P_2\text{ is not live } \times\]

\[(P_2 \mid R_1)\text{ is live } \checkmark\]
Property: Liveness (partial deadlock freedom)

Liveness for Select

\[
\big( \bigwedge_{\tilde{a} \in \mathcal{P}(\mathcal{A})} \downarrow \tilde{a} \big) \implies \text{eventually} \left( \langle \{ \tau_{a} \mid a \in \tilde{a} \} \rangle \text{true} \right)
\]

\[
P_1 = \text{select}\{\tilde{a}, b, \tau.P\}
\]

\[
P_2 = \text{select}\{\tilde{a}, b\}
\]

\[
R_1 = a
\]

\[
P_1 \text{ is live if } P \text{ is } \checkmark
\]

\[
P_2 \text{ is not live } \times
\]

\[
(P_2 \mid R_1) \text{ is live } \checkmark
\]

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
mrg.doc.ic.ac.uk
Property: Liveness (partial deadlock freedom)

\[
\left( \bigwedge_{a \in A} \downarrow a \lor \downarrow \bar{a} \right) \implies \text{eventually} \left( \langle \tau_a \rangle \text{true} \right)
\]

\[
\left( \bigwedge_{\bar{a} \in \mathcal{P}(A)} \downarrow \bar{a} \right) \implies \text{eventually} \left( \langle \{\tau_a \mid a \in \bar{a}\} \rangle \text{true} \right)
\]

```go
func main() {
    ch := make(chan int)
    go looper() // !!!
    <-ch       // No matching send
}

func looper() {
    for {
    }
}
```

× Runtime detector: Hangs
✓ Our tool: NOT live
Property: Liveness (partial deadlock freedom)

\[
\left( \bigwedge_{a \in A} \downarrow a \lor \downarrow \bar{a} \right) \implies \text{eventually} \left( \langle \tau_a \rangle \text{true} \right)
\]

\[
\left( \bigwedge_{\bar{a} \in \mathcal{P}(A)} \downarrow \bar{a} \right) \implies \text{eventually} \left( \langle \{ \tau_a | a \in \bar{a} \} \rangle \text{true} \right)
\]

```go
func main() {
    ch := make(chan int)
    go loopSend(ch)
    <-ch
}

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

What about this one?

- Type: Live
- Program: NOT live

Needs additional guarantees
Property: Eventual reception

\[
(\bigwedge_{a \in A} \downarrow a^\bullet) \implies \text{eventually } (\langle \tau_a \rangle \text{true})
\]

If an item is sent to a buffered channel \((a^\bullet)\),
Then \textbf{eventually} it can be consumed/synchronised \((\tau_a)\)

(i.e. no orphan messages)
Termination checking

Addressing the program-type abstraction gap
Termination checking with KITTeL

Type inference does not consider *program data*

- Type liveness \( \neq \) Program liveness if program non-terminating
- Especially when involving iteration

⇒ Check for loop termination
- If terminates, type liveness = program liveness

<table>
<thead>
<tr>
<th>Program terminates</th>
<th>Program does not terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type live</td>
<td>✗ Program live</td>
</tr>
<tr>
<td>Type not live</td>
<td>✗ Program not live</td>
</tr>
</tbody>
</table>
Tool: Godel-Checker

https://github.com/nickng/gospal
https://bitbucket.org/MobilityReadingGroup/godel-checker

Understanding Concurrency with Behavioural Types

GolangUK Conference 2017
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
In the paper

See our paper for omitted topics in this talk:

- Behavioural type inference algorithm
- Treatment of buffered (asynchronous) channels
- The select (non-deterministic choice) primitive
- Definitions of behavioural type semantics/barbs

Table 3: Go programs verified by our framework and comparison with existing static deadlock detection tools.

<table>
<thead>
<tr>
<th>Programs</th>
<th>LoC</th>
<th># states</th>
<th>(\psi_g)</th>
<th>(\psi_I)</th>
<th>(\psi_s)</th>
<th>(\psi_a)</th>
<th>Godel Checker</th>
<th>dingo-hunter [36]</th>
<th>gopherlyzer [40]</th>
<th>Golfer/Gong [30]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mismatch</td>
<td>29</td>
<td>53</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>620.7</td>
<td>996.8</td>
<td>996.7</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>fixed</td>
<td>27</td>
<td>16</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>624.4</td>
<td>996.5</td>
<td>996.3</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>fanin</td>
<td>41</td>
<td>39</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>631.1</td>
<td>996.2</td>
<td>996.2</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>sieve</td>
<td>43</td>
<td>(\infty)</td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>n/a</td>
<td>608.9</td>
<td>(\checkmark)</td>
<td>19.8</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>phi</td>
<td>41</td>
<td>65</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>6.1</td>
<td>996.5</td>
<td>996.6</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>dinephi13</td>
<td>55</td>
<td>3838</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>645.2</td>
<td>996.4</td>
<td>996.3</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>starvphi13</td>
<td>47</td>
<td>3151</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>628.2</td>
<td>996.5</td>
<td>996.5</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>sel</td>
<td>46</td>
<td>103</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>4.2</td>
<td>996.7</td>
<td>996.6</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>selfixed</td>
<td>22</td>
<td>20</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>4.0</td>
<td>996.3</td>
<td>996.4</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>jobsched</td>
<td>43</td>
<td>43</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>632.7</td>
<td>996.7</td>
<td>1996.1</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>forselect</td>
<td>42</td>
<td>26</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>623.3</td>
<td>996.4</td>
<td>996.3</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>cond-recur</td>
<td>37</td>
<td>12</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>4.0</td>
<td>996.2</td>
<td>996.2</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>concysys</td>
<td>118</td>
<td>15</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>549.7</td>
<td>996.5</td>
<td>996.4</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>alt-bit</td>
<td>70</td>
<td>112</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>634.4</td>
<td>996.3</td>
<td>996.3</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>prod-cons</td>
<td>28</td>
<td>106</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>4.1</td>
<td>996.4</td>
<td>1996.2</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>nonlive</td>
<td>16</td>
<td>8</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>630.1</td>
<td>996.6</td>
<td>996.6</td>
<td>timeout</td>
</tr>
<tr>
<td>double-close</td>
<td>15</td>
<td>17</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>3.5</td>
<td>996.6</td>
<td>1996.6</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>stuckmsg</td>
<td>8</td>
<td>4</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
<td>3.5</td>
<td>996.6</td>
<td>996.6</td>
<td>(\checkmark)</td>
</tr>
</tbody>
</table>

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
Future and related work

Extend framework to support more safety properties
Different verification approaches
  - Godel-Checker model checking [ICSE’18] (this talk)
  - Gong type verifier [POPL’17]
  - Choreography synthesis [CC’15]
Different concurrency issues (e.g. data races)
Distributed Programming using Role-Parametric Session Types in Go

David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, Nobuko Yoshida
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.
Scribble-Go workflow

1. Write a role-parametric global protocol
2. Select endpoint role variant to implement (e.g. Fetcher)
3. Use Scribble-Go to project and generate Endpoint API
4. Implement endpoint (e.g. Fetcher[3]) using the Endpoint API
Role variant

Role variant are *unique kinds* of endpoints

\{ M, F[1..n], Server \}

If F[1] sends an extra request

HTTP HEAD to Server to get total size

Then acts as a normal F

The role variants are:

\{ M, F[1], F[2..n], Server \}

→ F[1] and F[2..n] are different endpoints

Inference of role variants (indices): formulated as SMT constraints for Z3
Endpoint API generation and usage

FSMs from local protocols → Message passing API

- Fluent-style
  - Every state is a unique type (struct)
  - Method calls (communication) returns next state
- Type information can be leveraged by IDEs
  - “dot-driven” content assist & auto complete
Behavioural Types for Go

Type syntax

$$\alpha \ := \ \overline{u} \ | \ u \ | \ \tau$$

$$T, S \ := \ \alpha; T \ | \ T \ \oplus\ S \ | \ \&\{\alpha_i; T_i\}_{i \in I} \ | \ (T \ | \ S) \ | \ 0 \ | \ (new\ a)T \ | \ close\ u; T \ | \ t(\overline{u}) \ | \ \lfloor u \rfloor^n \ | \ buf[u]_{\text{closed}}$$

$$T \ := \ \{t(\tilde{y}_i) = T_i\}_{i \in I} \ \text{in} \ S$$

- 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)
Semantics of types

\[
\begin{align*}
\text{SND} & \quad \bar{a}; T \xrightarrow{\bar{a}} T \\
\text{RCV} & \quad a; T \xrightarrow{a} T \\
\text{TAU} & \quad \tau; T \xrightarrow{\tau} T \\
\text{END} & \quad \text{close } a; T \xrightarrow{\text{clo } a} T \\
\text{BUF} & \quad \lceil a \rceil^k \xrightarrow{\text{clo } a} \text{buf}[a]_{\text{closed}} \\
\text{CLD} & \quad \text{buf}[a]_{\text{closed}} \xrightarrow{a^*} \text{buf}[a]_{\text{closed}} \\
\text{SEL} & \quad i \in \{1, 2\} \\
& \quad \frac{T_1 \oplus T_2 \xrightarrow{\tau} T_i}{T \xrightarrow{\alpha} T'} \\
\text{PAR} & \quad \frac{T \mid S \xrightarrow{\alpha} T' \mid S}{T \xrightarrow{\alpha} T'} \\
\text{SEQ} & \quad \frac{T \xrightarrow{\alpha} T' \quad S \xrightarrow{\beta} S'}{T; S \xrightarrow{\alpha} T' ; S \xrightarrow{\beta} S'} \\
\text{TERM} & \quad 0; S \xrightarrow{\tau} S \\
\text{COM} & \quad \alpha \in \{\bar{a}, a^*, a.^*\} \\
& \quad \frac{T \xrightarrow{\alpha} T' \quad S \xrightarrow{\beta} S'}{T \mid S \xrightarrow{\tau_\alpha} T' \mid S'} \\
\text{EQ} & \quad T \equiv_\alpha T' \quad T \xrightarrow{\alpha} T'' \\
& \quad \frac{T' \xrightarrow{\alpha} T''}{T \xrightarrow{\alpha} T''} \\
\text{DEF} & \quad T \{\bar{a}/\bar{x}\} \xrightarrow{\alpha} T' \quad t(\bar{x}) = T \\
& \quad \frac{t(\bar{a}) \xrightarrow{\alpha} T'}{t(\bar{a}) \xrightarrow{\alpha} T'} \\
\text{CLOSE} & \quad \frac{T \xrightarrow{\text{clo } a} T' \quad S \xrightarrow{\text{clo } a} S'}{T \mid S \xrightarrow{\tau} T' \mid S'} \\
\text{IN} & \quad k < n \\
& \quad \frac{|a|^k \xrightarrow{\cdot a} |a|^{k+1}}{\text{OUT}} \\
& \quad k \geq 1 \\
& \quad \frac{|a|^k \xrightarrow{a.^* \cdot a} |a|^k_{k-1}}{\text{OUT}}
\end{align*}
\]
Barb predicates for types

\[
\begin{align*}
\text{a; } T \downarrow_a & \quad \text{close a; } T \downarrow_{\text{clo}} a \\
\bar{a}; T \downarrow_{\bar{a}} & \quad \text{buf}[a]_{\text{closed}} \downarrow_{a^*} \\
T \downarrow_o & \quad T \downarrow_a T' \downarrow_{\bar{a}} \text{ or } T' \downarrow_{a^*} \\
T \downarrow_a & \quad \alpha_i \downarrow_{\tau_a} \\
T \downarrow_{\bar{a}} & \quad \text{or } T \downarrow_{a^*} \quad \alpha_i \downarrow_a \\
k < n & \quad [a]_k^n \downarrow_{\text{new}} a \\
T \downarrow_o & \quad T \downarrow_o \quad a \notin \text{fn}(o) \\
T \downarrow_o & \quad T \equiv T' \\
T \downarrow_o & \quad T \equiv T' 
\end{align*}
\]

Figure: Barb predicates for types.