Behavioural Type-Based Static Verification Framework for GO

Julien Lange  Nicholas Net  Bernardo Toninho  Nobuko Yoshida
The paper *Multiparty asynchronous session types* by Kohei Honda, Nobuko Yoshida, and Marco Carbone, published in POPL 2008 has been awarded the ACM SIGPLAN Most Influential POPL Paper Award today at POPL 2018.

10 Jan 2018

Estatet has published a page on their usage of the Scribble language developed in our group with RedHat and other industry partners.

25 Sep 2017

Nick spoke at Golang UK 2017 on applying behavioural types to verify concurrent Go programs.

**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
POPL 2008 Most Influential Paper Award

Kohei Honda, Nobuko Yoshida and Marco Carbone

Multiparty asynchronous session types
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.
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.

www.estafet.com

Estafet Managed Service
End-to-End Switching Programme by DCC

Caveats:
1. Using earlier implementation of Scribble (CDL), because we already have those tools
2. Using earlier plugin to Eclipse - we’d want to improve this
3. We’re not going via OpenTracing - this is part of the bid costs

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
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 Abdeljalil
Imperial College London
United Kingdom

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

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

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

shots fired @zeeshanlakhani · Mar 12
Replying to @graydon_pub @dsyme
Awesome!

Brendan Zabarauskas @brendanzab · Mar 12
Replying to @graydon_pub
This stuff fills me with hope!

Ryan Riley @panesofglass · Mar 12
Replying to @graydon_pub
This is amazing! I guess I need to switch
Selected Publications 2017/2018

- [CC’18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
[CC’18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.


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

JANUARY 25, 2018

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

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

We're jumping ahead to ICSE ’18 now, and a paper that has been accepted for publication there later this year. It fits with the theme we've been exploring this week though, so I thought I'd cover it now. We've seen verification techniques applied in the context of Rust and JavaScript, looked at the integration of linear types in Haskell, and today it is the turn of Go!
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 behavioural types in process calculi.
  e.g. [ACM Survey, 2016] 185 citations, 6 pages

- Dynamic channel creations, unbounded thread creators...

- Scalable (synchronous/asynchronous) Modular, rerunnable

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
Overview

1. Type inference

Behavioural Types

Transform and verify

Model checking

mCRL2 model checker
Check safety and liveness

Termination checking

KITTeL termination prover
Address type ↔ program gap

Static verification framework for Go

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

Goroutines

```go
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
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!"
}
```

- 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
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!"
}
```

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

Deadlock detection

```
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

### Missing 'go' keyword

```go
// import _ "net"
func main() {
    ch := make(chan string)
    send(ch) // Oops
    print(<-ch)
    close(ch)
}
func send(ch chan string) {
    ch <- "Hello Kent!"
}
```

- 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
Concurrency in Go 🐧

Deadlock detection

```go
// import _ "net"

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

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

Go’s runtime deadlock detector
- Checks if all goroutines are blocked (‘global’ deadlock)
- Print message then crash
- Some packages disable it (e.g. net)
Concurrency in Go

Deadlock detection

Missing 'go' keyword

```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"
}
```

Import unused, unrelated package
Concurrency in Go 🚪

Deadlock detection

Missing 'go' keyword

```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"
}
```

- 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

Model checking behavioural types

Termination checking

Summary

Behavioural type inference

Abstract Go communication as Behavioural Types

Behavioural Types

Type inference

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
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

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

send(ch) = \overline{ch}
```
Infer Behavioural Types from Go program

Go source code

1  func main() {  
2     ch := make(chan int)  
3     go send(ch)  
4     print(<-ch)  
5     close(ch)  
6  }
7  
8  func send(c chan int) {  
9     c <- 1  
10  }

Inferred Behavioural Types

create channel \rightarrow \text{main()} = (\text{new ch});

spawn \rightarrow \text{send}(\text{ch}) | \text{ch};

receive \rightarrow \text{close}(\text{ch});

send \rightarrow \text{send}(\text{ch}) = \overline{\text{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
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
```
Infer Behavioural Types from Go program

package main

func main.main()
entry
0

\[ t0 = \text{make} \; \text{chan} \; \text{int} \; 0 : \text{int} \]
\[ \text{go} \; \text{sendFn}(t0) \]
\[ t1 = \text{recvVal}(t0) \]
jump 3

func main.sendFn(c)
entry
0

\[ \text{send} \; c \leftarrow 42 : \text{int} \]
return

func main.recvVal(c)
entry
0

\[ t0 = \llcorner c \lrcorner \]
return

0

\[ t5 = \phi_i[0 : 0 : \text{int}, 1 : t3] \]
\[ t6 = t5 < t1 \]
if t6 goto 1 else 2

1

\[ t2 = \text{print}(t5) \]
\[ t3 = t5 + 1 : \text{int} \]
jump 3

2

\[ t4 = \text{close}(t0) \]
return

3

\[ 0 \]

Analyse in

\textbf{Static Single Assignment}

SSA representation of input program

- Only inspect \textit{communication} primitives
- Distinguish between unique channels
Model checking behavioural types

From behavioural types to model and property specification

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
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/\texttt{close} on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Behavioural Types as \textbf{LTS model}

Standard CS semantics, i.e.

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

Send on channel \( a \)

Synchronise on \( a \)

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

Standard CS semantics, i.e.

\[
\begin{align*}
\overline{a}; T \xrightarrow{\overline{a}} T \\
T \mid S \xrightarrow{T_a} T' \mid S' \\
a; T \xrightarrow{a} T
\end{align*}
\]

Send on channel a \hspace{2cm} Synchronise on a \hspace{2cm} 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]
- $\mu$-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

\[
\bar{a}; T \downarrow_a \quad T \downarrow_a \quad T' \downarrow_a
\]

Barbs \((T \downarrow_o)\)

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

\[
\begin{align*}
\bar{a}; T & \downarrow a \\
T & \downarrow \bar{a} \quad T' & \downarrow a \\
T \mid T' & \downarrow \tau_a \\
\end{align*}
\]

- 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 \(o\)
Specifying properties of model

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

Express **safety/liveness properties**
- As μ-calculus formulae
- In terms of the **model** and the **barbs**

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

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

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

\(A = \) set of all initialised channels \quad \mathbb{A} = \) set of all labels

\(\Rightarrow\) Ready receive/send = not end of program.
Property: Global deadlock freedom

\[(\bigwedge_{a \in \mathcal{A}} \downarrow a \lor \downarrow \bar{a}) \iff \langle \mathcal{A} \rangle \text{true}\]

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

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

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

\[(\bigwedge_{a \in A} \downarrow a^{\bullet}) \implies \neg (\downarrow a \lor \downarrow \text{clo } a)\]

```go
func main() {
    ch := make(chan int)
    go func(ch chan int) {
        ch <- 1 // is ch closed?
    }(ch)
    close(ch)
    <-ch
}
```

- \(\downarrow \text{clo } ch\) when \(\text{close}(ch)\)
- \(\downarrow ch^{\bullet}\) fires after closed
- Send (\(\downarrow \text{ch}\): line 4)
Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

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

If a channel is ready to receive or send, then 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

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

where:

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

If a channel is ready to receive or send, then for some reachable state it can synchronise \( \left( \tau_a \right) \).
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 \texttt{select} is ready to receive or send, Then \textbf{eventually} it will synchronise \((\tau_a)\)
Property: Liveness (partial deadlock freedom)

Liveness for Select

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

\[P_1 = \text{select}\{\bar{a}, b, \tau.P\}\]
\[P_2 = \text{select}\{\bar{a}, b\}\]
\[R_1 = a\]

\[P_1\] is live if \(P\) is ✓
\[P_2\] is not live ×
\(\left( P_2 | R_1 \right)\) is live ✓
Property: Liveness (partial deadlock freedom)

Liveness for Select

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

\[P_1 = \text{select}\{a, b, \tau.P\}\]
\[P_2 = \text{select}\{a, b\}\]
\[R_1 = a\]

\[P_1 \text{ is live if } P \text{ is } \checkmark\]
\[P_2 \text{ is not live } \times\]
\[(P_2 \mid R_1) \text{ is live } \checkmark\]
Property: Liveness (partial deadlock freedom)

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

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

```
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)

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

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

```go
1 func main() {
2     ch := make(chan int)
3     go loopSend(ch)
4     <-ch
5 }
6 func loopSend(ch chan int) {
7     for i := 0; i < 10; i-- {
8         // Does not terminate
9     }
10     ch <- 1
11 }
```

What about this one?

- **Type:** Live
- **Program:** NOT live

Needs additional guarantees
Property: Eventual reception

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

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

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
Termination checking with KITTeL

Type inference does not consider *program data*

- Type liveness ≠ 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>
<tr>
<td></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.
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)
**Behavioural Types for Go**

**Type syntax**

\[
\begin{align*}
\alpha & := \bar{u} \mid u \mid \tau \\
T, S & := \alpha; T \mid T \oplus S \mid \{\alpha_i; T_i\}_{i \in I} \mid (T \mid S) \mid 0 \\
& \quad \mid (\text{new } a) T \mid \text{close } u; T \mid t\langle \tilde{u} \rangle \mid \lfloor u \rfloor_k \mid \text{buf}[u]_{\text{closed}} \\
T & := \{t(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S
\end{align*}
\]

- 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

\[
\begin{align*}
\text{SND} & \quad \overline{a}; T \xrightarrow{\bar{a}} T \quad \text{RCV} & \quad a; T \xrightarrow{a} T \\
\text{TAU} & \quad \tau; T \xrightarrow{\tau} T \\
\text{END} & \quad \text{close} a; T \xrightarrow{\text{clo}a} T \\
\text{BUF} & \quad \lfloor a \rfloor^k_k \xrightarrow{\text{clo}a} \text{buf}[a]_{\text{closed}} \\
\text{CLD} & \quad \text{buf}[a]_{\text{closed}} \xrightarrow{a} \text{buf}[a]_{\text{closed}} \\
\text{PAR} & \quad T \xrightarrow{\alpha} T' \\
\text{SEQ} & \quad T; S \xrightarrow{\alpha} T'; S \\
\text{TERM} & \quad 0; S \xrightarrow{\tau} S \\
\text{COM} & \quad \alpha \in \{a, a^*, a^\cdot\} \\
\text{EQ} & \quad T \equiv_\alpha T' \quad T \xrightarrow{\alpha} T'' \\
\text{DEF} & \quad T \{ \bar{a}/\bar{x} \} \xrightarrow{\alpha} T' \quad t(\bar{x}) = T \\
\text{CLOSE} & \quad T \xrightarrow{\text{clo}a} T' \quad S \xrightarrow{\text{clo}a} S' \\
\text{IN} & \quad k < n \\
\text{OUT} & \quad k \geq 1
\end{align*}
\]
Barb predicates for types

\[ \begin{align*}
    & a; T \downarrow_a \quad \text{close } a; T \downarrow_{\text{clo } a} \\
    & \bar{a}; T \downarrow \bar{a} \quad \text{buf}[a]_{\text{closed}} \downarrow_{a^*} \\
    & \forall i \in \{1, \ldots, n\} : \alpha_i \downarrow_{o_i} \\
    & \&\{\alpha_i; T\}_{i \in \{1, \ldots, n\}} \downarrow_{\alpha_1 \ldots \alpha_n} \\
    & T \downarrow_o \quad T \downarrow_a \quad T' \downarrow_{\bar{a}} \text{ or } T' \downarrow_{a^*} \\
    & T \downarrow_a \quad \{\bar{a}/\bar{x}\} \downarrow_o \quad t(\bar{x}) = T \\
    & T \downarrow_a \quad \alpha_i \downarrow_{\bar{a}} \\
    & \&\{\alpha_i; S_i\}_{i \in I} \downarrow_{\tau_a} \\
    & k < n \quad k \geq 1 \\
    & \{a\}^n_k \downarrow_{a^*} \quad \{a\}^n_k \downarrow_{\bar{a}^*} \\
    & T \downarrow_{\bar{a}} \quad T' \downarrow_{\bar{a}^*} \\
    & T \downarrow_{a^*} \quad \alpha_i \downarrow_{a} \\
    & T \downarrow_{\bar{a}} \quad \alpha_i \downarrow_{\bar{a}} \\
    & \&\{\alpha_i; S_i\}_{i \in I} \downarrow_{\tau_a} \\
    & T \downarrow_{\bar{a}} \quad T' \downarrow_{\tau_a} \\
    & T \downarrow_o \quad T \downarrow_{\bar{a}} \quad \bar{a}; T \downarrow_{\bar{a}} \\
    & \{a\}^n_k \downarrow_{a^*} \quad \{a\}^n_k \downarrow_{\bar{a}^*} \\
    & T \downarrow_{\bar{a}} \quad T' \downarrow_{\bar{a}^*} \\
    & T \downarrow_{a^*} \quad \alpha_i \downarrow_{a} \\
    & T \downarrow_{\bar{a}} \quad \alpha_i \downarrow_{\bar{a}} \\
    & \&\{\alpha_i; S_i\}_{i \in I} \downarrow_{\tau_a} \\
    & T \downarrow_{\bar{a}} \quad T' \downarrow_{\tau_a} \\
    & T \downarrow_o \quad T \equiv T' \\
    & T \downarrow_o
\end{align*} \]

Figure: Barb predicates for types.