Behavioural Type-Based Static Verification Framework for GO

Julien Lange Nicholas Net
Bernardo Toninho Nobuko Yoshida
http://mrg.doc.ic.ac.uk

Mobility Research Group

NEWS

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.

» more

10 Jan 2018

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

» more

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 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.
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

Scope of the demo:
7. Generate exception report and send back to DCC

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
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 Abdelljalall  
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...
Selected Publications 2017/2018

- **[LICS’18]** Romain Demangeon, NY: Casual Computational Complexity of Distributed Processes.
- **[CC’18]** Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
- **[ICSE’18]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY: A Static Verification Framework for Message Passing in Go using Behavioural Types.
- **[ECOOP’17]** Alceste Scala, Raymond Hu, Ornella Darda, NY: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming.
- **[COORDINATION’17]** Keigo Imai, NY, 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’18] Rumyana Neykova, Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.


Go concurrency verification research at DoC grabs headline

the morning paper

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, one thread, 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 calculus
  
  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 creations, ...
- Scalable (synchronous/asynchronous) Modular, retrainable

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
  - 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
   - Go source code
   - SSA IR

2. Model checking
   - mCRL2 model checker
   - Check safety and liveness

3. Termination checking
   - KITTeL termination prover
   - Address type ↔ program gap
Concurrency 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
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)
}
```

- Send message thru channel
- Print message on screen

Output:

```
$ go run hello.go
Hello Kent!
$
```
Concurrent in Go

Deadlock detection

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

Output:

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

```
 Julio 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**

### 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!"
}
```

**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

```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

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

```
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
```
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
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
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);`
- `send(ch) = ch`
- `create channel`
- `spawn`
- `receive`
- `close`

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

```
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()

0

t0 = make chan int 0:int

0

go sendFn(t0)

t1 = recvVal(t0)

0

jump 3

3


t5 = phi[0: 0:int, 1: t3] #i

t6 = t5 < t1

0

if t6 goto 1 else 2

1


t2 = print(t5)

t3 = t5 + 1:int

3

jump 3

2


t4 = close(t0)

1

return

for.loop

for.done


func main.sendFn(c)

0

entry

0

send c <- 42:int

0

return

func main.recvVal(c)

0

entry

0

t0 = <-c

0

return


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

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

mrg.doc.ic.ac.uk

Only inspect communication primitives

Distinguish between unique channels

Analyze in

Static Single Assignment

SSA representation of input program
Model checking behavioural types

From behavioural types to model and property specification

1. Type inference
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/close on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Behavioural Types as **LTS model**

Standard CS semantics, i.e.

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

Send on channel \(a\)

Synchronise on \(a\)

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

Standard CS semantics, i.e.

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

Send on channel \( a \)

\[ T \overset{\overline{a}}{\rightarrow} T' \quad S \overset{a}{\rightarrow} S' \]

Synchronise on \( a \)

\[ T \parallel S \overset{\tau a}{\rightarrow} T' \parallel S' \]

Receive on channel \( a \)

\[ a; T \overset{a}{\rightarrow} T \]
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

\[ \overline{a}; \ T \downarrow^a \]

\[ \frac{T \downarrow^a}{\ T | \ T' \downarrow^{\tau_a}} \]

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 \( o \)
Specifying properties of model

\( \bar{a}; T \downarrow \bar{a} \)

\[
\frac{T \downarrow \bar{a} \quad T' \downarrow a}{T | T' \downarrow T_{\bar{a}}}
\]

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 \(o\)
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

Channel Safety

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

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

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

Never closing

Never send

Barb
[Milner 8
Sangiorgi 92]
Migo Liveness / Safety

- Liveness

All reachable actions are eventually performed

\[ P \text{ is live if } P \xrightarrow{*(v,c)} Q \]

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

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

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

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

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

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

\[ R_1 = a? \]

Time Out

If P is live

P1 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
\( P_1 \) is live

\( P_2 \) is not live
\( P_2 | R_2 \) is
Select

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

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

Barb \[ \downarrow \bar{a} \]

\[ \pi_t \downarrow Q_e \]

select \( \{ \pi_t, P_i \} \downarrow \bar{a} \)

Time Out

if \( P \) is live

\( P_1 \) is live

\( P_2 \) is not live

\( P_2 \mid R_2 \) is

\[ R_1 = a? \]

Liveness \( Q \downarrow \bar{a} \Rightarrow Q \downarrow z \) at \( ai \)
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/`close` on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Property: Global deadlock freedom

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

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

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

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

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

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

\[(\forall a \in \mathcal{A}) \quad (\downarrow a^*) \implies \neg (\downarrow \bar{a} \lor \downarrow \text{clo } a)\]

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

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

```go
def main() {
    ch := make(chan int)
    go func(ch chan int) {
        ch <- 1 // is ch closed?
    }(ch)
    close(ch)
    <-ch
}
```
Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

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

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

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

where:

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

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

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

\[\begin{align*}
P_1 &= \text{select}\{\bar{a}, b, \tau.P\} \\
P_2 &= \text{select}\{\bar{a}, b\} \\
R_1 &= a
\end{align*}\]

\(P_1\) is live if \(P\) is \(\checkmark\)

\(P_2\) is not live \(\times\)

\((P_2 \mid R_1)\) is live \(\checkmark\)
Property: Liveness (partial deadlock freedom)

Liveness for Select

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

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

P₁ is live if P is √
P₂ is not live ×
\((P₂ \mid R₁)\) is live √
Property: Liveness (partial deadlock freedom)

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

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

```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)

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

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

```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

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

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

(i.e. no orphan messages)
Termination checking

Addressing the program-type abstraction gap

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

SSA IR

Go source code

Transform and verify
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></th>
<th>Program terminates</th>
<th>Program does not terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type live</td>
<td>✓ Program live</td>
<td>?</td>
</tr>
<tr>
<td>Type not live</td>
<td>✗ Program not live</td>
<td>✗ Program not live</td>
</tr>
</tbody>
</table>
Concurrency in Go  Behavioural type inference  Model checking behavioural types  Termination checking

Summary

Tool: Godel-Checker

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

Understanding Concurrency with Behavioural Types

GolangUK Conference 2017

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

Conclusion
In the paper

See our paper for omitted topics in this talk:

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

<table>
<thead>
<tr>
<th>Programs</th>
<th>LoC</th>
<th># states</th>
<th>$\psi_q$</th>
<th>$\psi_I$</th>
<th>$\psi_T$</th>
<th>$\psi_s$</th>
<th>$\psi_c$</th>
<th>Infer</th>
<th>Live</th>
<th>Live+CS</th>
<th>Term</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>$\times$</td>
<td>$\times$</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>
<td>$\times$</td>
<td>639.4</td>
<td>$\times$</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>$\checkmark$</td>
<td>624.4</td>
<td>996.5</td>
<td>996.3</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>603.1</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>$\checkmark$</td>
<td>631.1</td>
<td>996.2</td>
<td>996.2</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>608.9</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>sieve</td>
<td>43</td>
<td>$\infty$</td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
<td>-</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>34.2</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>philo</td>
<td>41</td>
<td>65</td>
<td>$\times$</td>
<td>$\times$</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>
<td>$\times$</td>
<td>32.7</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>dinephill</td>
<td>55</td>
<td>3838</td>
<td>$\checkmark$</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>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
</tr>
<tr>
<td>starvphil</td>
<td>47</td>
<td>3151</td>
<td>$\times$</td>
<td>$\times$</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>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
</tr>
<tr>
<td>sel</td>
<td>22</td>
<td>103</td>
<td>$\times$</td>
<td>$\times$</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>
<td>$\times$</td>
<td>15.3</td>
<td>$\times$</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>$\checkmark$</td>
<td>4.0</td>
<td>996.3</td>
<td>996.4</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>14.9</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>$\checkmark$</td>
<td>632.7</td>
<td>996.7</td>
<td>1966.1</td>
<td>$\checkmark$</td>
<td>n/a</td>
<td>-</td>
<td>n/a</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>$\checkmark$</td>
<td>623.3</td>
<td>996.4</td>
<td>996.3</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>618.8</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>$\checkmark$</td>
<td>4.0</td>
<td>996.2</td>
<td>996.2</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>9.4</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>concsys</td>
<td>118</td>
<td>15</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>549.7</td>
<td>996.5</td>
<td>996.4</td>
<td>$\checkmark$</td>
<td>n/a</td>
<td>-</td>
<td>n/a</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>$\checkmark$</td>
<td>634.4</td>
<td>996.3</td>
<td>996.3</td>
<td>$\checkmark$</td>
<td>n/a</td>
<td>-</td>
<td>n/a</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>$\checkmark$</td>
<td>4.1</td>
<td>996.4</td>
<td>1996.2</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td>10.1</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>$\checkmark$</td>
<td>630.1</td>
<td>996.6</td>
<td>996.5</td>
<td>timeout</td>
<td>613.6</td>
<td>n/a</td>
<td>-</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>$\checkmark$</td>
<td>3.5</td>
<td>996.6</td>
<td>1996.6</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>8.7</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>stickmsg</td>
<td>8</td>
<td>4</td>
<td>$\checkmark$</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>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
</tr>
</tbody>
</table>
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

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

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

\[
\mid (\text{new } a)T \mid \text{close } u; T \mid t(\tilde{u}) \mid u_k^n \mid \text{buf}[u]_{\text{closed}}
\]

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

- 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

\[
\begin{align*}
\text{SND} & \quad \bar{a}; T \xrightarrow{\bar{a}} T \\
\text{RCV} & \quad a; T \xrightarrow{a} T \\
\text{TAU} & \quad \tau; T \xrightarrow{\tau} T \\
\text{END} & \quad \text{close } a; T \xrightarrow{\text{clo } a} T \\
\text{BUF} & \quad \lfloor a \rfloor^k_k \xrightarrow{\text{clo } a} \text{buf}[a]_{\text{closed}} \\
\text{CLD} & \quad \text{buf}[a]_{\text{closed}} \xrightarrow{a} \text{buf}[a]_{\text{closed}} \\
\text{SEL} & \quad i \in \{1,2\} \\
\frac{T_1 \oplus T_2 \xrightarrow{\tau} T_i}{T} \\
\text{PAR} & \quad T \xrightarrow{\alpha} T' \\
\frac{T \mid S \xrightarrow{\alpha} T' \mid S}{T \mid S} \\
\text{SEQ} & \quad T \xrightarrow{\alpha} T' \\
\frac{T; S \xrightarrow{\alpha} T'; S}{T; S} \\
\text{TERM} & \quad 0; S \xrightarrow{\tau} S \\
\text{COM} & \quad \alpha \in \{\bar{a}, a^*, a^\circ\} \\
\frac{T \equiv \alpha \quad T \xrightarrow{\alpha} T''}{T \xrightarrow{\alpha} T''} \\
\text{EQ} & \quad T \xrightarrow{\text{clo } a} T' \\
\frac{T' \xrightarrow{\alpha} T''}{T' \xrightarrow{\alpha} T''} \\
\text{DEF} & \quad T \{\bar{a}/x\} \xrightarrow{\alpha} T' \\
\frac{t(\bar{a}) \xrightarrow{\alpha} T'}{t(\bar{a}) \xrightarrow{\alpha} T'} \\
\text{IN} & \quad k < n \\
\frac{\lfloor a \rfloor^k_k \xrightarrow{a} \lfloor a \rfloor^{n+1}_k}{\lfloor a \rfloor^k_k} \\
\text{OUT} & \quad k \geq 1 \\
\frac{\lfloor a \rfloor^n_k \xrightarrow{a} \lfloor a \rfloor^{n-k-1}_k}{\lfloor a \rfloor^n_k}
\end{align*}
\]
Barb predicates for types

\[
\begin{align*}
a; T \Downarrow_a & \quad \text{close } a; T \Downarrow_{\text{clo}} a \\
\bar{a}; T \Downarrow_{\bar{a}} & \quad \text{buf}[a]_{\text{closed}} \Downarrow_{a^*} \\
T \Downarrow_o & \quad T \Downarrow_a \quad T' \Downarrow_{\bar{a}} \quad \text{or} \quad T' \Downarrow_{a^*} \\
T \Downarrow_{\bar{a}} & \quad T \Downarrow_o \quad \alpha_i \Downarrow_{\bar{a}} \\
k < n & \quad k \geq 1 \\
[a]_k \Downarrow_{\text{•}a} & \quad [a]_k \Downarrow_{a^*} \\
T \Downarrow_o & \quad T \Downarrow_o \quad a \notin \text{fn}(o) \\
T \Downarrow_o \quad (\text{new}^n a); T \Downarrow_o & \quad T \Downarrow_o \quad T \equiv T'
\end{align*}
\]

Figure: Barb predicates for types.