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

Julien Lange\(^1\), Nicholas Ng\(^2\), Bernardo Toninho\(^3\), Nobuko Yoshida\(^2\)

\(^1\)University of Kent \quad \(^2\)Imperial College London
\(^3\)Universidade Nova de Lisboa
The Go Programming Language

- Developed at Google for multicore programming
- Statically typed, natively compiled, **concurrent**
- Channel-based message passing for concurrency
- Used by major technology companies, e.g.
Go and concurrency
Approach and philosophy

*Do not communicate by sharing memory; Instead, share memory by communicating*
— Go language proverb

Encourages message passing over locking
- **Goroutines**: lightweight threads
- **Channels**: typed FIFO queues

Inspired by Hoare’s CSP/process calculi
Overview

Concurrency in Go

Behavioural type inference

Model checking behavioural types

Termination checking

Summary

Static verification framework for Go

Overview

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
Concurrent in Go 🌧️

Goroutines

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

func send(ch chan string) {
    ch <- "Hej ICSE!"
}
```

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

Channels

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

func send(ch chan string) {
  ch <- "Hej ICSE!"
}
```

- 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 <- "Hej ICSE!"
}
```

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 <- "Hej ICSE!"
}
```

- Send message thru channel
- Print message on screen

Output:

```
$ go run hello.go
Hej ICSE!
$
```
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
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 <- "Hej ICSE"
}
```

Import unused, unrelated package
Concurrency in Go 🦇

Deadlock detection

Missing 'go' keyword

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

func send(ch chan string) {
    ch <- "Hej ICSE"
}
```

- 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
   - 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

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**

- `main()` = `(new ch); (send(ch) | ch; 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
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

```
create channel

main() = (new ch);
spawn
(send(ch) | ch;
receive
close
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
Infer Behavioural Types from Go Program

package main

func main.main()

0

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

3

\[
t5 = \text{phi}[0: 0: \text{int}, 1: t3]\#it6 = t5 < t1
\]
\[
\text{if } t6 \text{ goto 1 else 2}
\]

1

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

2

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

func main.sendFn(c)

0

\[
\text{send c <- 42: int}
\]
\[
return
\]

func main.recvVal(c)

0

\[
t0 = <-c
\]
\[
return t0
\]

Analyse in

Static Single Assignment

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

2. Transform and verify

Model checking

mCRL2 model checker
Check safety and liveness

3. 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: \textit{Finite control} (no spawning in loops)
- Global deadlock freedom
- Channel safety (no send/\texttt{close} on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Behavourial Types as \textbf{LTS model}

Standard CCS reduction, i.e.

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

Send on channel \(a\) 

Synchronise on \(a\) 

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

Standard CCS reduction, 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 \(a\)
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** \((T \downarrow_\circ)\)
- Predicates of state/type \(T\)
- Holds when \(T\) is ready to fire action \(\circ\)
Specifying **properties** of model

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

\[ \frac{T \downarrow \overline{a} \quad T' \downarrow \overline{a}}{T \parallel T' \downarrow \tau \overline{a}} \]

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

Ready to send \quad Ready to synchronise \quad 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

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

- 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 $\mu$-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: Liveness (partial deadlock freedom)

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

\[A = \text{set of initialised channels}\]

If a channel is ready to receive or send, then \textbf{eventually} it will become synchronised (\(\tau_a\))

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

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

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 will become synchronised (\(\tau_a\))
Property: Liveness (partial deadlock freedom)

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

```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)

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

What about this one?

- Type: Live
- Program: NOT live

Needs additional guarantees
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

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

Type inference does not consider *program data*

- Type liveness $\neq$ Program liveness if program non-terminating
- Especially when involving iteration
  $\Rightarrow$ 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>
Overview  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
mrg.doc.ic.ac.uk
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_e)</th>
<th>Godel Checker</th>
<th>dingo-hunter [36]</th>
<th>gopherlyzer [40]</th>
<th>Golfer/Gong [30]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Live</td>
<td>DF</td>
<td>Live</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td>mismatch</td>
<td>38</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>fixed</td>
<td>36</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>finan</td>
<td>39</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>slice</td>
<td>43</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>phi</td>
<td>41</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>dinephi13</td>
<td>55</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>starvephi13</td>
<td>47</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>sel</td>
<td>22</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>selFixed</td>
<td>22</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>jobsched</td>
<td>43</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>forselect</td>
<td>42</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>cond-recurs</td>
<td>37</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>concsxs</td>
<td>118</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>alt-bit</td>
<td>70</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>prod-cons</td>
<td>28</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>nonlive</td>
<td>16</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>double-close</td>
<td>15</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>stuckmsg</td>
<td>8</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</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)
Property: Global deadlock freedom

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

import _ "net" // unused

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

func send(ch chan string) {
    ch <- "Hej ICSE"
}
Property: Global deadlock freedom

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

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

\( A = \text{set of all initialised channels} \quad A = \text{set of all labels} \)
Property: Global deadlock freedom

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

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

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

\( \implies \) Ready receive/send = not end of program.
Property: Channel safety

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

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

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

1. func main() {
   2. ch := make(chan int)
   3. go func(ch chan int) {
      4. ch <- 1 // is ch closed?
   5. }(ch)
   6. close(ch)
   7. <-ch
3. /6

- \( \downarrow \text{clo ch} \) when close(ch)
- \( \downarrow \text{ch}^* \) fires after closed
- Send (\( \downarrow \overline{\text{ch}} \): line 4)
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: Liveness (select)

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

"If one of the channels in select is ready to receive or send, then eventually it will synchronise (\(\tau_a\)) (i.e. there's corresponding send for receiver/recv for sender)"
Property: Eventual reception

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

“If an item is sent to a buffered channel \((a \bullet)\), Then \textit{eventually} it will be consumed/synchronised \((\tau_a)\) (i.e. no orphan messages)
Behavioural Types for Go

Type syntax

\[
\begin{align*}
\alpha & := \overline{u} | u | \tau \\
T, S & := \alpha; T | T \oplus S | \&\{\alpha_i; T_i\}_{i \in I} | (T | S) | 0 \\
& | (\text{new } a)T | \text{close } u; T | t(\tilde{u}) | [u]^n_k | \text{buf}[u]_{\text{closed}} \\
T & := \{t(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S
\end{align*}
\]

- 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)