Behavioural Type-Based Static Verification Framework for Go

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

May 14, 2018
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

etc..
Go and concurrency
Approach and philosophy

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

Inspired by Hoare’s CSP/process calculi

- **Goroutines**: lightweight threads
- **Channels**: typed FIFO queues

Encourages message passing over locking
Overview

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 term. prover

Address type ↔ process gap
Concurrency in Go

Concurrency primitives

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

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

Creates new channel.
Use `make(chan int, 1)` for buffered channel (capacity=1)
Concurrent in Go

Concurrency primitives

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

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

Spawns `send` function as new goroutine (thread)
Concurrency in Go 🌸

Concurrency primitives

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

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

- Receive from channel `ch`
- Send to channel `ch`
Concurrency in Go

Concurrent primitives

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

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

Close channel ch
Concurrency in Go

Concurrency primitives

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

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

Also available: `select-case`
Wait on multiple communication operations
(like `switch-case` for communication)
Concurrent in Go

Concurrency primitives

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

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

$ go run main.go
Hej ICSE!
$
Overview  Concurrency in Go  Behavioural type inference  Model checking behavioural types  Termination checking  Summary

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

$ go run main.go
fatal error: all goroutines are asleep - deadlock!
$
Concurrency in Go 🧢

Deadlock detection

Go’s runtime deadlock detector

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

```
// import _ "net"

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

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

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

Behavioural Type-Based Static Verification Framework for Go 🐒
Overview
Concurrency in Go
Behavioural type inference
Model checking behavioural types
Termination checking
Summary

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"
}
```

$ go run main.go

Hangs: Deadlock NOT detected
Concurrency in Go

Deadlock detection

Our goal

Check liveness/safety properties in addition to global deadlocks by statically analysing Go code e.g. model checking
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 *term. prover*
   - Address type ↔ process gap
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
    t0 = make chan int 0:int
    go sendFn(t0)
    t1 = recvVal(t0)
    jump 3

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

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

```
func main.sendFn(c)
    entry
    send c <- 42:int
    return
```

```
func main.recvVal(c)
    entry
    t0 = <-c
    return t0
```

Analyse in

Single Static Assignment

SSA representation of input program

- Only inspect `communication` primitives
- Distinguish channels: context-sensitive static analysis
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() = \( \text{new } ch \); \( \text{send}(ch) \mid \text{ch}; \text{close } ch \),
- send(ch) = ch

- create channel
- spawn
- receive
- close
- send
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 term. prover
   - Address type ↔ process gap
Model checking behavioural types

\[ M \models \phi \]

1. **LTS model**: inferred type + type semantics
2. **Safety/liveness properties**: \( \mu \)-calculus formulae for LTS
3. 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 to LTS model

- Standard CCS reduction, i.e.

$$
\bar{a}; T \xrightarrow{\bar{a}} T
$$

$$
T \xrightarrow{a} T' \quad S \xrightarrow{a} S'
\quad T \mid S \xrightarrow{\tau} T' \mid S'
\quad a; T \xrightarrow{a} T
$$

- Send on channel $a$
- Synchronise on $a$
- Receive on channel $a$
Behavioural Types as LTS model

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

func send(c chan int) { 
    c <- 1 
}
```
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** ($T \downarrow_0$)

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

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

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

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

\[ \overline{ch} | \ ch; \ close \ ch \]

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

\[
\overline{ch} \mid \text{close } ch
\]

\[
\tau
\]

\[
\text{close } ch
\]

\[
\tau
\]

\[
\emptyset
\]

\[
ch^*
\]

\[
\downarrow \{ \overline{ch}, ch, \tau \}
\]

\[
\downarrow \{ \text{close } ch, \tau \}
\]

\[
\downarrow ch^*
\]

Ready to Send on \( ch \)

Ready to Receive on \( ch \)

Ready to Synchronise on \( ch \)

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/closure on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Property: Global deadlock freedom

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

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

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

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 \overline{A} = \) set of all labels
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)

\(A = \text{set of all initialised channels}\) \hspace{1cm} \(A = \text{set of all labels}\)

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

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

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

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

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

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

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

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

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

```go
func main() {
    ch := make(chan int)
    go looper() // something is always happening here
    <-ch // No matching send
}

func looper() {
    for {
    }
}
```
Property: Liveness (partial deadlock freedom)

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

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 \texttt{receiver}/recv for \texttt{sender})
Property: Liveness (partial deadlock freedom)

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

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 will become synchronised \((\tau_a)\)
Model checking behavioural types

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
  - “Reachable”: Require additional guarantees
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 term. prover**
   - Address type ↔ process gap
Model checking behavioural types

Termination checking with KITTeL

Type inference do not consider *data* in program

- Type liveness $\neq$ program liveness
- Especially when involving iteration

$\Rightarrow$ Check for loop termination

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

- Type: Live
- Program: NOT live
Model checking behavioural types

Termination checking with KITTeL

Type inference do not consider data in program

- Type liveness ≠ program liveness
- Especially when involving iteration

⇒ Check for loop termination
   (if program terminates, then type liveness = program liveness)

✓ Global deadlock freedom
✓ Channel safety (no send/close on closed channel)
✓ Liveness (partial deadlock freedom)
✓ Eventual reception
Overview  Concurrency in Go  Behavioural type inference  Model checking behavioural types  Termination checking  Summary

Tool

```go
package main
import "fmt"

//import _ "net" // Load "net" package

func main() {
    ch := make(chan int) // Create channel.
    send(ch)              // Spawn as goroutine.
    print(<-ch)           //Recv from channel.
}

func send(ch chan int) { // Channel as parameter.
    fmt.Println("Waiting to send...")
    ch <- 1           // Send to channel.
    fmt.Println("Sent")
}
```

---

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

*Behavioural Type-Based Static Verification Framework for Go*
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>Golnfer/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>Live</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>Live</td>
<td>Time</td>
</tr>
<tr>
<td>mismatch</td>
<td>29</td>
<td>53</td>
<td>×</td>
<td>×</td>
<td>√</td>
<td>√</td>
<td>620.7</td>
<td>996.8</td>
<td>996.7</td>
<td>√</td>
</tr>
<tr>
<td>fixed</td>
<td>27</td>
<td>16</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>624.4</td>
<td>996.5</td>
<td>996.3</td>
<td>√</td>
</tr>
<tr>
<td>fanin</td>
<td>41</td>
<td>39</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>631.1</td>
<td>996.2</td>
<td>996.2</td>
<td>√</td>
</tr>
<tr>
<td>sieve</td>
<td>43</td>
<td>∞</td>
<td>n/a</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td>philo</td>
<td>41</td>
<td>65</td>
<td>×</td>
<td>×</td>
<td>√</td>
<td>√</td>
<td>6.1</td>
<td>996.5</td>
<td>996.6</td>
<td>√</td>
</tr>
<tr>
<td>dinephil13</td>
<td>55</td>
<td>3838</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>645.2</td>
<td>996.4</td>
<td>996.3</td>
<td>√</td>
</tr>
<tr>
<td>starvephi13</td>
<td>47</td>
<td>3151</td>
<td>×</td>
<td>×</td>
<td>√</td>
<td>√</td>
<td>628.2</td>
<td>996.5</td>
<td>996.5</td>
<td>√</td>
</tr>
<tr>
<td>sel</td>
<td>22</td>
<td>103</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
<td>4.2</td>
<td>996.7</td>
<td>996.6</td>
<td>√</td>
</tr>
<tr>
<td>selfixed</td>
<td>22</td>
<td>20</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>4.0</td>
<td>996.3</td>
<td>994.4</td>
<td>√</td>
</tr>
<tr>
<td>jobsched</td>
<td>43</td>
<td>43</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>632.7</td>
<td>996.7</td>
<td>1996.1</td>
<td>√</td>
</tr>
<tr>
<td>forselect</td>
<td>42</td>
<td>26</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>623.3</td>
<td>996.4</td>
<td>996.3</td>
<td>√</td>
</tr>
<tr>
<td>cond-recur</td>
<td>37</td>
<td>12</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>4.9</td>
<td>996.2</td>
<td>996.2</td>
<td>√</td>
</tr>
<tr>
<td>concsys</td>
<td>118</td>
<td>15</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
<td>549.7</td>
<td>996.5</td>
<td>996.4</td>
<td>√</td>
</tr>
<tr>
<td>alt-bit</td>
<td>70</td>
<td>112</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>634.4</td>
<td>996.3</td>
<td>996.3</td>
<td>√</td>
</tr>
<tr>
<td>prod-cons</td>
<td>28</td>
<td>106</td>
<td>×</td>
<td>×</td>
<td>√</td>
<td>√</td>
<td>4.1</td>
<td>996.4</td>
<td>1996.2</td>
<td>√</td>
</tr>
<tr>
<td>nonlive</td>
<td>16</td>
<td>8</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>630.1</td>
<td>996.6</td>
<td>996.5</td>
<td>timeout</td>
</tr>
<tr>
<td>double-close</td>
<td>17</td>
<td>17</td>
<td>×</td>
<td>√</td>
<td>×</td>
<td>√</td>
<td>3.5</td>
<td>996.6</td>
<td>1996.6</td>
<td>√</td>
</tr>
<tr>
<td>stuckmsg</td>
<td>8</td>
<td>4</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>3.5</td>
<td>996.6</td>
<td>996.6</td>
<td>√</td>
</tr>
</tbody>
</table>
Future work

Extend framework to support more safety properties
Unlimited possibilities!

- 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: Eventual reception

\[(\bigwedge \downarrow a^\bullet) \implies \text{eventually } (\langle \tau \rangle \text{true})\]

“If an item is sent to a buffered channel \((a^\bullet)\),
Then \textbf{eventually} it will be consumed/synchronised \((\tau)\)
(i.e. no orphan messages)
Property: Liveness (select)

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

“If one of the channels in \texttt{select} is ready to receive or send, then \textit{eventually} it will become synchronised (\(\tau_a\))

(i.e. there’s corresponding send for \texttt{receiver}/recv for \texttt{sender})