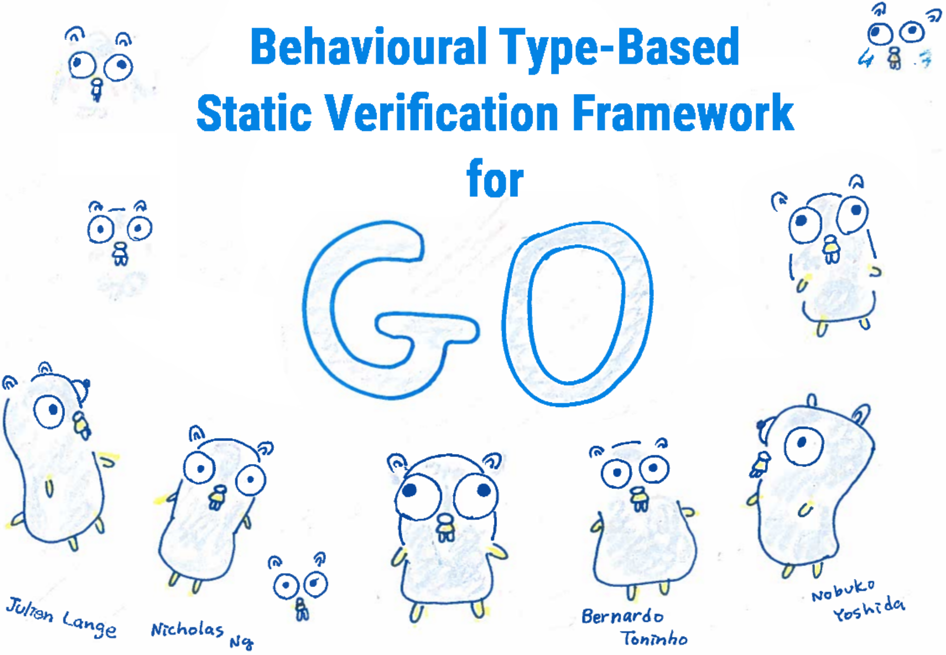
Behavioural Type-Based Static Verification Framework for



Julien Lange

Nicholas Ng

Bernardo Toninho

Nobuko Yoshida

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 — gorounines
  - Selective send/receive

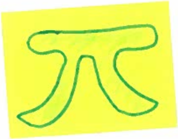
CSP 80'

FUN Dropbox, Netfix, 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

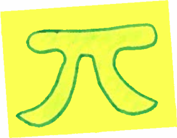
- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?



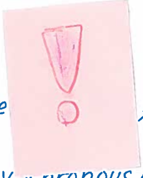
- Use behavioural types in process calculi e.g. [ACM Survey, 2016] 185 citations, 6 pages

- 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



- Dyn ne , unbounded thread creations, recursions,..
- Scalable (sy nchronous / asynchronous) Modular, Refinable

- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?

π

- Use behavioural types in process calculi

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

186 ??

- hannel 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?

π

- Use behavioural types in process calculi e.g. [ACM Survey, 2016] **185** citations, 6 pages

- Dynamic channel creations, unbounded thread creation ons, ..
- Scalable (synchronous/asynchronous) Modular, Refinable

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)

↑ GAP ↓

STEP 3

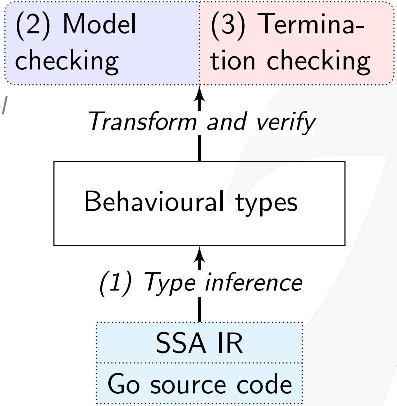
- Relate Safety / Liveness of Behavioural Types and GO Programs
  - 3 Classes [POPL'17]
  - Termination Check

Verification framework for Go

Overview

Check safety and liveness

Create input model and formula



(2) Model checking

(3) Termination checking

Transform and verify

Behavioural types

(1) Type inference

SSA IR

Go source code

Address type and process gap

Pass to termination prover

Concurrency in Go

Concurrency primitives

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

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

- Send/receive blocks goroutines if channel full/empty resp.
- Channel buffer size specified at creation: `make(chan int, 1)`
- Other primitives:
  - Close a channel `close(ch)`
  - Guarded choice `select { case <-ch:; case <-ch2: }`

Concurrency in Go

Deadlock detection

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

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

Missing 'go' keyword

Concurrency in Go

Deadlock detection

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

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

Run program:

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

Concurrency in Go

Deadlock detection

- Go has a runtime deadlock detector, panics (crash) if deadlock
- Deadlock if all goroutines are blocked
- Some packages (e.g. `net` for networking) **disables** it

```go
import _ "net"  // Load "net" package
func main() {
        ch := make(chan int)
        send(ch)
        print(<-ch)
}
func send(ch chan int) { ch <- 1 }
```

Concurrency in Go

Deadlock detection

- Go has a runtime deadlock detector, panics (crash) if deadlock
- Deadlock if all goroutines are blocked
- Some packages (e.g. `net` for networking) **disables** it

```
import _ "net"  // Load "net" p
func main() {
        ch := make(chan int)
        send(ch)
        print(<-ch)
}
func send(ch chan int) { ch <- 1 }
```

Add benign import

Deadlock **NOT** detected

Abstracting Go with Behavioural Types

Type syntax

$$
\begin{array}{rcl}
\alpha & := & \overline{u} \mid u \mid \tau \\
T, S & := & \alpha; T \mid T \oplus S \mid \&\{\alpha_i; T_i\}_{i \in I} \mid (T \mid S) \mid \mathbf{0} \\
 & \mid & (\mathtt{new}\, a) T \mid \mathtt{close}\, u; T \mid \mathbf{t}\langle \tilde{u} \rangle \\
\mathbf{T} & := & \{\mathbf{t}(\tilde{y}_i) = T_i\}_{i \in I} \;\mathtt{in}\; S
\end{array}
$$

- 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)

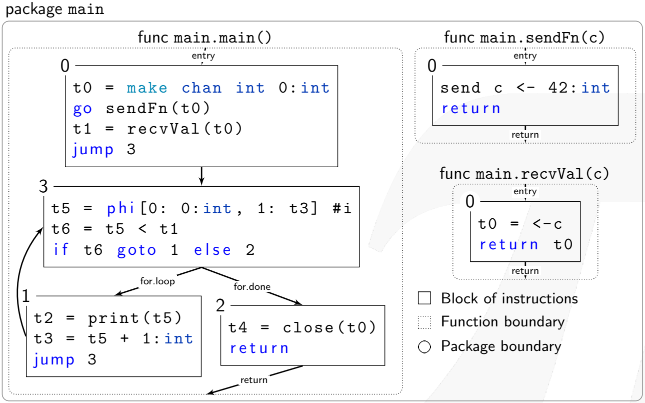
Verification framework for Go (1)

Type inference by example

```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
```

Verification framework for Go (1)

Program in Static Single Assignment (SSA) form



package main

func main.main()

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

```
3
t5 = phi [0: 0:int, 1: t3] #i
t6 = t5 < t1
if t6 goto 1 else 2
```

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

```
2
t4 = close(t0)
return
```

func main.sendFn(c)

```
0
send c <- 42:int
return
```

func main.recvVal(c)

```
0
t0 = <-c
return t0
```

□ Block of instructions

⬚ Function boundary

○ Package boundary

- Context-sensitive analysis to distinguish channel variables
- Skip over non-communication code

Verification framework for Go

Types inferred from 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
```

$$
\begin{aligned}
\mathbf{main}() &= (\mathtt{new}\ t0)(\mathbf{sendFn}\langle t0\rangle \mid \mathbf{recvVal}\langle t0\rangle; \mathbf{main\_3}\langle t0\rangle) \\
\mathbf{main\_1}(t0) &= \mathbf{main\_3}\langle t0\rangle \\
\mathbf{main\_2}(t0) &= \mathtt{close}\ t0; \mathbf{0} \\
\mathbf{main\_3}(t0) &= \mathbf{main\_1}\langle t0\rangle \oplus \mathbf{main\_2}\langle t0\rangle \\
\mathbf{sendFn}(c) &= \overline{c}; \mathbf{0} \\
\mathbf{recvVal}(c) &= c; \mathbf{0}
\end{aligned}
$$

Verification framework for Go (2)

Model checking with mCRL2

Generate LTS model and formulae from types

- Finite control (no parallel composition in recursion)
- Properties (formulae for model checker):
  - ✓ Global deadlock
  - ✓ Channel safety (no send/`close` on closed channel)
  - ✗ Liveness (partial deadlock)
  - ✗ Eventual reception
    - Require additional guarantees

Verification framework for Go (3)

Termination checking with KITTeL

- Extracted types do not consider *data* in process
- Type liveness != program liveness
  - Especially when involving iteration
  - Check for loop termination
- Properties:
  - ✓ Global deadlock
  - ✓ Channel safety (no send/`close` on closed channel)
  - ✓ Liveness (partial deadlock)
  - ✓ Eventual reception

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

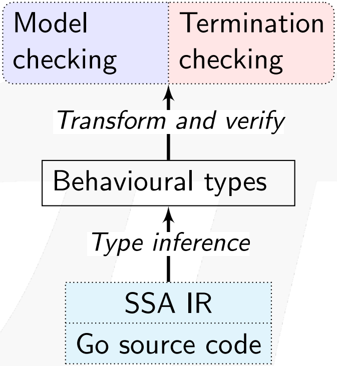
- Type: Live
- Program: NOT live

Tool demo

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



Future work

- Extend framework to support more properties
- Unlimited possibilities!
  - Different verification techniques
    - e.g. [POPL'17], Choreography synthesis [CC'15]
  - Different concurrency issues
    - Other synchronisation mechanisms
    - Race conditions