Behavioural Type-Based Static Verification Framework for GO

Julian Lange  Nicholas Ng  Bernardo Toninho  Nobuko Yoshida
Go programming language at Google (2009)

- Message-Passing based multicore PL, successor of C

- Do not communicate by shared memory; instead, share memory by communicating

- 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, online, 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

- **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, retinable

  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
Verification framework for Go

Overview

1. Type inference
2. Model checking
3. Termination checking

Transform and verify

Behavioural types

Check safety and liveness
Create input model and formula

Address type and process gap
Pass to termination prover

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

Behavioural Type-Based Static Verification Framework for Go

mrg.doc.ic.ac.uk
Concurrency in Go
Concurrency primitives

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

func send(ch chan int) {
    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

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

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

Missing 'go' keyword
Concurrenci 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!
```
Concurrenc 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 }
```
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 }
```

Deadlock **NOT** detected
Abstracting Go with Behavioural Types

Type syntax

\[ \alpha \ := \ \overline{u} \mid u \mid \tau \]
\[ T, S \ := \ \alpha ; T \mid T \oplus S \mid \&\{\alpha_i, T_i\}_{i \in I} \mid (T \mid S) \mid 0 \]
\[ \mid (\text{new } a) T \mid \text{close } u; T \mid t\langle\tilde{u}\rangle \]
\[ T \ := \ \{t(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S \]

- 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)
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

- 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
```

\[
\begin{align*}
\text{main}() & = (\text{new } t0)(\text{sendFn}⟨t0⟩ | \text{recvVal}⟨t0⟩; \text{main}_3⟨t0⟩) \\
\text{main}_1(t0) & = \text{main}_3⟨t0⟩ \\
\text{main}_2(t0) & = \text{close } t0; 0 \\
\text{main}_3(t0) & = \text{main}_1⟨t0⟩ ⊕ \text{main}_2⟨t0⟩ \\
\text{sendFn}(c) & = \mathcal{C}; 0 \\
\text{recvVal}(c) & = c; 0
\end{align*}
\]
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/closed 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 $\neq$ program liveness
  - Especially when involving iteration
  - Check for loop termination
- Properties:
  - ✓ Global deadlock
  - ✓ Channel safety (no send/cancel 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
}
```

- **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