Behavioural Type-Based Static Verification Framework for Go

Julien Lange, **Nicholas Ng**,
Bernardo Toninho, Nobuko Yoshida

The Go Programming Language

- Developed by Google for multicore programming
- Statically typed, natively compiled, **concurrent** PL
- Supports channel-based message passing for concurrency

In use by major technology companies

docker kubernetes UBER twitter NETFLIX etc..

Concurrency in Go

Basic primitives and philosophy

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

- Message-passing concurrency primitives
  - Buffered I/O communication over *channels*
  - Lightweight thread spawning (goroutines)
  - Non-deterministic selection construct
- Inspired by Hoare's CSP/process calculi
- **Encourages** message-passing over locking

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

```
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

```
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" pa
func main() {
      ch := make(chan int)
      send(ch)
      print(<-ch)
}
func send(ch chan int) { ch <- 1 }
```

Add benign import

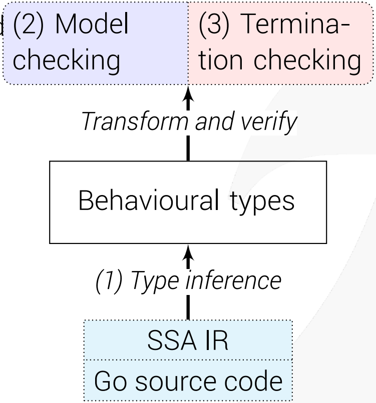
Deadlock **NOT** detected

Verification framework for Go

Overview

Check safety and liveness

Create input model and formula



Address type and process gap

Pass to termination prover

Behavioural Types

Types for process calculi, e.g.

- CCS, π-calculus (Milner 1980, 1992)
- CSP (Hoare 1978)

Model concurrent systems **behaviours**

- e.g. Process (thread) creations
- e.g. (a)sync. send/recv message passing
- Guarantees free of deadlocks etc.

Typically powerful but **complex**

This work instead aims to make behavioural type accessible

Type Abstraction

Program/Process

Analyse “directly”

- e.g. send(`x: int`)
- Evaluate expressions

Accurate but Expensive

Check `x == 1`
Check `x == 2`
Check `x == …`
→ State Explosion

Types

Analyse *Types*
+ relate Process ↔ Types

- Data abstracted away
- e.g. send `int`/`bool`

Data needed in *some cases*!

- Process/types mismatch
- *3 classes* of processes
  → (POPL'17)

More concrete ⟷ More abstract

Type Abstraction

Program/Process

Analyse "directly"

- e.g. send(`x: int`)
- Evaluate expressions

Accurate but Expensive

Check `x == 1`
Check `x == 2`
Check `x == …`
→ State Explosion

Types

Analyse *Types*
+ **termination check**

- Data abstracted away
- e.g. send `int/bool`

Data needed in *some cases*!

- Process/types mismatch
- *3 classes* of processes
  → (POPL'17)

← More concrete — More abstract →

Abstracting Go with Behavioural Types

Type syntax

$$\alpha \;:=\; \overline{u} \mid u \mid \tau$$

$$T, S \;:=\; \alpha; T \mid T \oplus S \mid \&\{\alpha_i; T_i\}_{i\in I} \mid (T \mid S) \mid \mathbf{0}$$
$$\mid\; (\mathsf{new}\, a)T \mid \mathtt{close}\, u; T \mid \mathbf{t}\langle\tilde{u}\rangle$$

$$\mathbf{T} \;:=\; \{\mathbf{t}(\tilde{y}_i) = T_i\}_{i\in I} \;\mathsf{in}\; S$$

- Types of a CCS-like process calculi
- Abstracts Go concurrency **primitives**
  - Send/Recv, new (channel), parallel composition (spawn)
  - Go-specific: Close channel, Select (guarded choice)

Verification framework for Go (1)

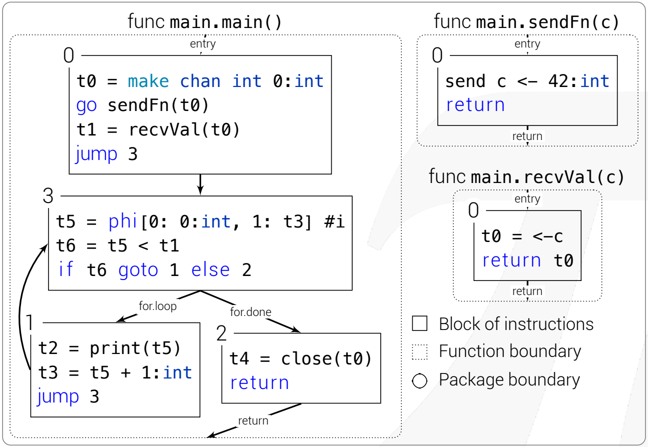
Type inference by example

```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 channel c
func recvVal(c chan int) int { return <-c } // Receive from c
```

Verification framework for Go (1)

Program in Static Single Assignment (SSA) form

package main



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

Verification framework for Go

Types inferred from 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 channel c
func recvVal(c chan int) int { return <-c } // Receive from c
```

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

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

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

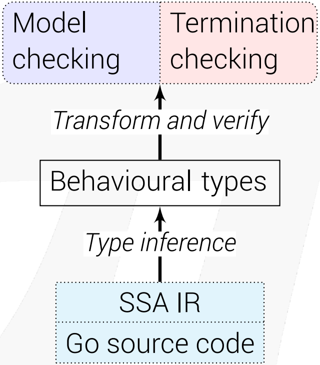
- Type: Live
- Program: NOT live

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