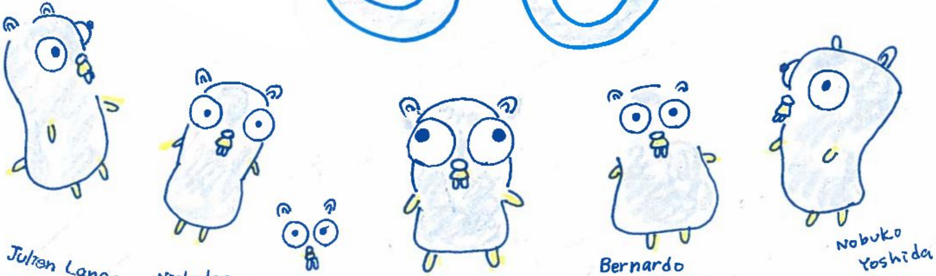
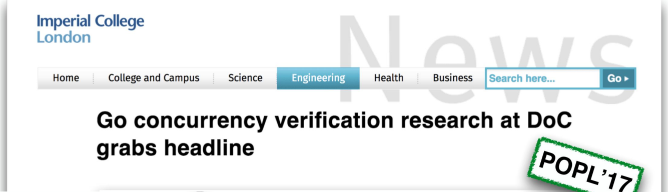
Behavioural Type-Based Static Verification Framework for GO

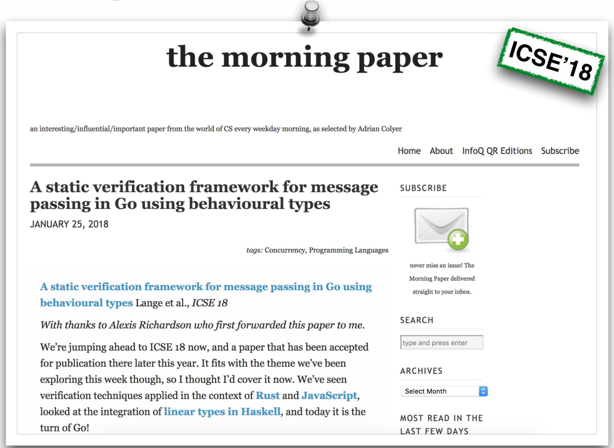






Julien Lange
Nicholas Ng
Bernardo Toninho
Nobuko Yoshida





the morning paper

an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

Home About InfoQ QR Editions Subscribe

A static verification framework for message passing in Go using behavioural types

JANUARY 25, 2018

tags: Concurrency, Programming Languages

A static verification framework for message passing in Go using behavioural types Lange et al., *ICSE 18*

With thanks to Alexis Richardson who first forwarded this paper to me.

We're jumping ahead to ICSE 18 now, and a paper that has been accepted for publication there later this year. It fits with the theme we've been exploring this week though, so I thought I'd cover it now. We've seen verification techniques applied in the context of **Rust** and **JavaScript**, looked at the integration of **linear types in Haskell**, and today it is the turn of Go!

SUBSCRIBE

never miss an issue! The Morning Paper delivered straight to your inbox.

SEARCH

type and press enter

ARCHIVES

Select Month

MOST READ IN THE LAST FEW DAYS

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

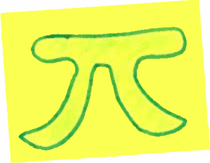
- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?

π

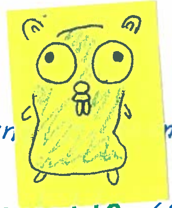
- Use behavioural types in process calculi
  e.g. [ACM Survey, 2016] 185 citations, 6 pages

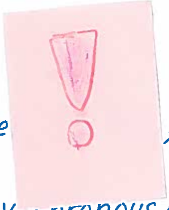
- Dynamic channel creations, unbounded thread creations, recursions,..
- Scalable (synchronous/asynchronous) Modular, Refinable

- 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 creatio 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)

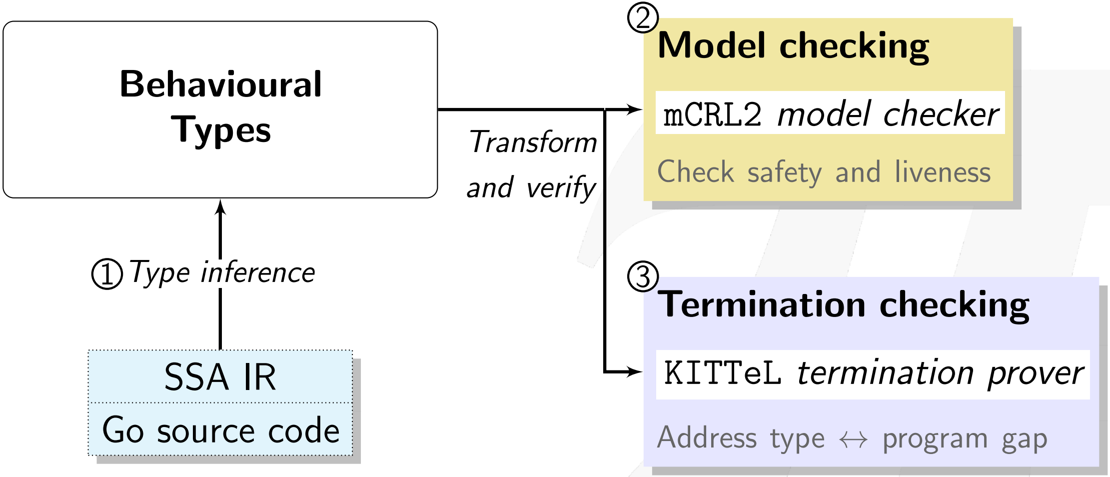
GAP

STEP 3

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

Static verification framework for Go

Overview



Concurrency in Go

Goroutines

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

func send(ch chan string) {
  ch <- "Hello Kent!"
}
```

go keyword + function call

- Spawns function as goroutine
- Runs in parallel to parent

Concurrency in Go

Channels

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

func send(ch chan string) {
  ch <- "Hello Kent!"
}
```

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 <- "Hello Kent!"
}
```

Also `select-case`:

- Wait on multiple channel operations
- `switch-case` for communication

Concurrency in Go

Deadlock detection

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

func send(ch chan string) {
  ch <- "Hello Kent!"
}
```

- Send message thru channel
- Print message on screen

Output:

```
$ go run hello.go
Hello Kent!
$
```

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 <- "Hello Kent!"
}
```

- Only one (main) goroutine
- Send without receive - blocks

Output:

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

Concurrency in Go

Deadlock detection

Missing 'go' keyword

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

func send(ch chan string) {
  ch <- "Hello Kent!"
}
```

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

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

func send(ch chan string) {
  ch <- "Hello Kent"
}
```

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 <- "Hello Kent"
}
```

- Only one (main) goroutine
- Send without receive - blocks

Output:

```
$ go run deadlock2.go
```

Hangs: Deadlock **NOT** detected

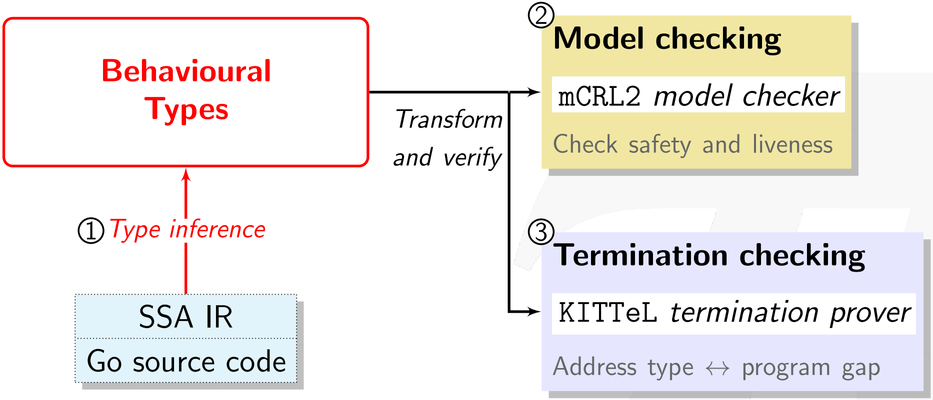
Our goal

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**



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

$$\rightarrow \left\{ \begin{array}{rcl} \mathsf{main}() & = & (\mathtt{new}\, ch); \\ & & (\mathsf{send}\langle \mathsf{ch} \rangle \mid \\ & & \mathsf{ch}; \\ & & \mathtt{close}\, \mathsf{ch}), \\ & & \\ \mathsf{send}(ch) & = & \overline{\mathsf{ch}} \end{array} \right\}$$

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

spawn

receive

close

send

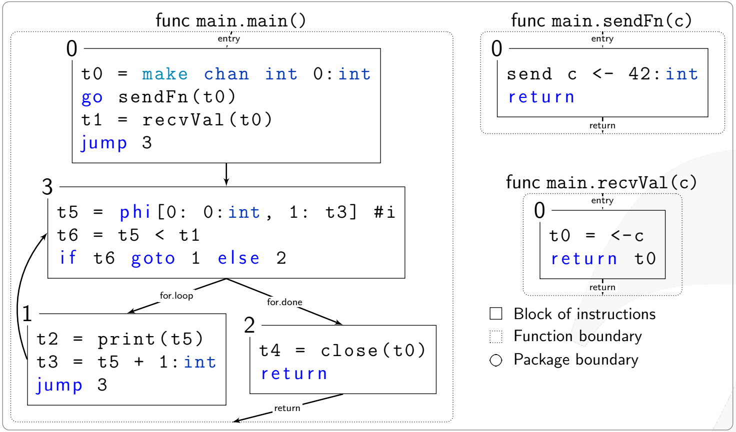
$$\left\{ \begin{array}{rcl} \mathrm{main}() & = & (\mathsf{new}\; ch); \\ & & (\mathrm{send}\langle \mathrm{ch}\rangle \mid \\ & & \;\mathrm{ch}; \\ & & \;\mathsf{close}\; \mathrm{ch}), \\ & & \\ \mathrm{send}(ch) & = & \overline{\mathrm{ch}} \end{array} \right\}$$

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



Analyse in

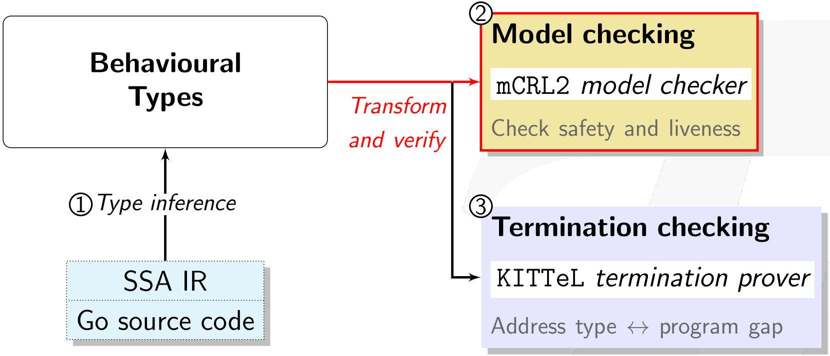
Static **S**ingle **A**ssignment

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**



Model checking behavioural types

$$\mathbf{M} \models \phi$$

- **LTS model** : inferred type + type semantics
- **Safety/liveness properties** : μ-calculus formulae for LTS
- Check with mCRL2 model checker
  - mCRL2 constraint: *Finite control* (no spawning in loops)

- Global deadlock freedom
- Channel safety (no send/`close` on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception

Behavioural Types as **LTS model**

Standard CS semantics, i.e.

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

$$\frac{T \xrightarrow{\overline{a}} T' \quad S \xrightarrow{a} S'}{T \mid S \xrightarrow{\tau_a} T' \mid S'}$$

$a; T \xrightarrow{a} T$

Send on channel a Synchronise on a Receive on channel a

Behavioural Types as **LTS model**

Standard CS semantics, i.e.

$$\overline{a}; T \xrightarrow{\overline{a}} T \qquad \frac{T \xrightarrow{\overline{a}} T' \quad S \xrightarrow{a} S'}{T \mid S \xrightarrow{\tau_a} T' \mid S'} \qquad a; T \xrightarrow{a} T$$

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]
- μ-calculus properties specified in terms of barbs

Barbs ($T \downarrow_o$)

- Predicates of state/type T
- Holds when T is ready to fire action o

Specifying **properties** of model

$\overline{a}; T \downarrow_{\overline{a}}$

Ready to send

$$\frac{T \downarrow_{\overline{a}} \quad T' \downarrow_{a}}{T \mid T' \downarrow_{\tau_a}}$$

Ready to synchronise

$a; T \downarrow_{a}$

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

$\overline{a}; T \downarrow_{\overline{a}}$

Ready to send

$$\frac{T \downarrow_{\overline{a}} \qquad T' \downarrow_{a}}{T \mid T' \downarrow_{\tau_a}}$$

Ready to synchronise

$a; T \downarrow_{a}$

Ready to receive

Barbs ($T \downarrow_o$)

- Predicates of state/type T
- Holds when T is ready to fire action o

Mi Go Liveness / Safety

$P \Downarrow a$

Barb [Milner & Sangiorgi 92]

Channel Safety

- Channel is closed at most once
- Can only input from a closed channel (default value)
- Others raise an error and crash

MiGo Liveness / Safety

$P \Downarrow a$

Barb [Milner & Sangiorgi 92]

Channel Safety

- Channel is closed at most once
- Can only input from a closed channel (default value)
- Others raise an error and crash

P is **channel safe** if $P \rightarrow^{*} (\nu \tilde{c}) Q$ and $Q \downarrow close(a)$

a closed

$\neg(Q \Downarrow end(a)) \wedge \neg(Q \Downarrow \overline{a} \quad)$

never closing

never send

Migo Liveness / Safety

- **Liveness**

All reachable actions are eventually performed

P is live if $P \rightarrow^{*} (\nu \tilde{c}) Q$

$$Q \downarrow a \Rightarrow Q \Downarrow \tau \text{ at } a$$

$$Q \downarrow \bar{a} \Rightarrow Q \Downarrow \tau \text{ at } a$$

Reduction (τau) at a

Select

$P_1 = \text{select}\ \{a!,\ b?,\ z.P\}$

Time Out

$P_2 = \text{select}\ \{a!,\ b?\}$

$R_1 = a?$

Select

$P_1 = \text{select}\ \{a!,\ b?,\ z.P\}$

Time Out

if P is live P_1 is live

$P_2 = \text{select}\ \{a!,\ b?\}$

$R_1 = a?$

Select

$P_1 = \text{select } \{a!,\ b?,\ z.P\}$

Time Out

if P is live P_1 is live

$P_2 = \text{select } \{a!,\ b?\}$

$R_1 = a?$

P_2 is <u>not</u> live

$P_2 | R_2$ is

Select

$P_1 = \text{select}\ \{a!,\ b?,\ z.P\}$

Time Out

if P is live P_1 is live

$P_2 = \text{select}\ \{a!,\ b?\}$ $\qquad R_1 = a?$

P_2 is <u>not</u> live

$P_2 | R_2$ is

Barb $\downarrow \tilde{a}$

$$\frac{\pi_i \downarrow a_i}{\text{select}\ \{\pi_i.\ P_i\} \downarrow \tilde{a}} \qquad \frac{P \downarrow \tilde{a} \qquad Q \downarrow \overline{a_i}}{P \mid Q \downarrow [a_i]}$$

Liveness $\quad Q \downarrow \tilde{a} \Rightarrow Q \Downarrow z$ at a_i

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/`close` on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception

Property: Global deadlock freedom

$$\left(\bigwedge_{a \in \mathcal{A}} \downarrow_a \vee \downarrow_{\overline{a}}\right) \implies \langle \mathbb{A} \rangle \mathtt{true}$$

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 $\mathbb{A}$ = set of all labels

$\Rightarrow$ Ready receive/send = not end of program.

Property: Global deadlock freedom

$$\left(\bigwedge_{a\in\mathcal{A}} \downarrow_{a} \vee \downarrow_{\overline{a}}\right) \implies \langle\mathbb{A}\rangle\mathtt{true}$$

```
1  import _ "net" // unused
2  func main() {
3      ch := make(chan string)
4      send(ch) // Oops
5      print(<-ch)
6      close(ch)
7  }
8
9  func send(ch chan string) {
10     ch <- "Hello Kent"
11 }
```

- Send ($\downarrow_{\overline{ch}}$: line 10)
- No synchronisation
- No more reduction

Property: Channel safety

$$\Big(\bigwedge_{a \in \mathcal{A}} \downarrow_{a^*}\Big) \implies \neg\big(\downarrow_{\overline{a}} \vee \downarrow_{\mathtt{clo}\ a}\big)$$

Once a channel a is closed (a^*),
it will not be sent to, nor closed again ($\mathtt{clo}\ a$)

Property: Channel safety

$$\left(\bigwedge_{a\in\mathcal{A}} \downarrow_{a^*}\right) \implies \neg\left(\downarrow_{\overline{a}} \vee \downarrow_{\texttt{clo}\ a}\right)$$

```
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
8 }
```

- $\downarrow_{\texttt{clo ch}}$ when `close(ch)`
- $\downarrow_{\texttt{ch}^*}$ fires after closed
- Send ($\downarrow_{\overline{\texttt{ch}}}$: line 4)

Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

$$\left(\bigwedge_{a \in \mathcal{A}} \downarrow_{a} \vee \downarrow_{\overline{a}}\right) \implies \text{eventually}\,(\langle \tau_a \rangle \texttt{true})$$

If a channel is ready to receive or send,
then **eventually**
it can synchronise (τ_a)

(i.e. there's corresponding send for receiver/recv for sender)

Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

$$\left(\bigwedge_{a \in \mathcal{A}} \downarrow_a \vee \downarrow_{\overline{a}}\right) \implies \text{eventually}\left(\langle \tau_a \rangle \texttt{true}\right)$$

where:

$$\text{eventually}(\phi) \stackrel{\text{def}}{=} \mu \mathbf{y}.\left(\phi \vee \langle \mathbb{A} \rangle \mathbf{y}\right)$$

If a channel is ready to receive or send,

then **for some reachable state**
it can synchronise (τ_a)

Property: Liveness (partial deadlock freedom)

Liveness for Select

$$\left(\bigwedge_{\tilde{a} \in \mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}} \right) \implies \text{eventually}\,(\langle \{ \tau_a \mid a \in \tilde{a} \} \rangle \texttt{true})$$

If one of the channels in `select` is ready to receive or send,
Then **eventually** it will synchronise (τ_a)

Property: Liveness (partial deadlock freedom)

Liveness for Select

$$\left(\bigwedge_{\tilde{a} \in \mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}} \right) \implies \text{eventually}\left(\langle \{ \tau_a \mid a \in \tilde{a} \} \rangle \texttt{true} \right)$$

$$P_1 = \texttt{select}\{\overline{a}, b, \tau.P\}$$

$$P_2 = \texttt{select}\{\overline{a}, b\}$$

$$R_1 = a$$

P_1 is live if P is ✓

P_2 is not live ×

$(P_2 \mid R_1)$ is live ✓

Property: Liveness (partial deadlock freedom)

Liveness for Select

$$\left(\bigwedge_{\tilde{a} \in \mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}} \right) \implies \text{eventually}\left(\langle \{ \tau_a \mid a \in \tilde{a} \} \rangle \texttt{true}\right)$$

$$P_1 = \texttt{select}\{\overline{a}, b, \tau.P\}$$

$$P_2 = \texttt{select}\{\overline{a}, b\}$$

$$R_1 = a$$

P_1 is live if P is ✓

P_2 is not live ×

$(P_2 \mid R_1)$ is live ✓

Property: Liveness (partial deadlock freedom)

$$\Big(\bigwedge_{a\in\mathcal{A}} \downarrow_a \vee \downarrow_{\bar{a}}\Big) \implies \text{eventually}\,(\langle\tau_a\rangle\texttt{true})$$

$$\Big(\bigwedge_{\tilde{a}\in\mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}}\Big) \implies \text{eventually}\,(\langle\{\tau_a \mid a \in \tilde{a}\}\rangle\texttt{true})$$

```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)

$$\Big(\bigwedge_{a\in\mathcal{A}} \downarrow_a \vee \downarrow_{\overline{a}}\Big) \implies \text{eventually}\,(\langle\tau_a\rangle\mathtt{true})$$

$$\Big(\bigwedge_{\tilde{a}\in\mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}}\Big) \implies \text{eventually}\,(\langle\{\tau_a \mid a \in \tilde{a}\}\rangle\mathtt{true})$$

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

What about this one?

- Type: Live
- Program: NOT live

Needs additional guarantees

Property: Eventual reception

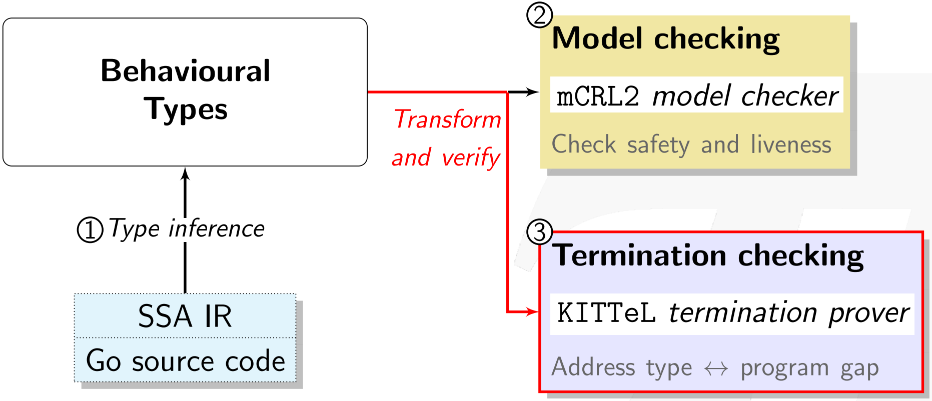
$$\Big(\bigwedge_{a \in \mathcal{A}} \downarrow_{a^\bullet}\Big) \implies \text{eventually}\,(\langle \tau_a \rangle \mathtt{true})$$

If an item is sent to a buffered channel ($a^\bullet$),
Then **eventually** it can be consumed/synchronised (τ_a)

(i.e. no orphan messages)

Termination checking

Addressing the program-type *abstraction gap*



Termination checking with KITTeL

Type inference does not consider *program data*

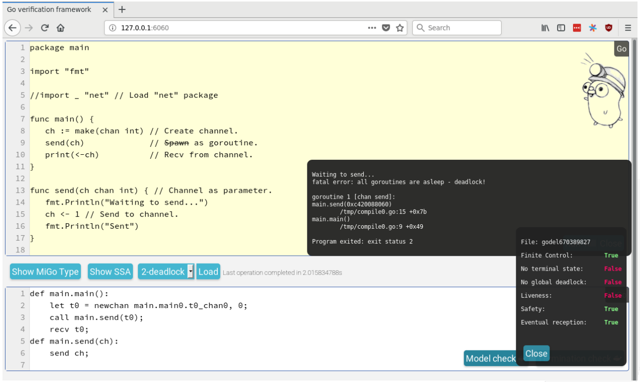
- Type liveness $\neq$ Program liveness if program non-terminating
- Especially when involving iteration

$\Rightarrow$ Check for loop termination

- If terminates, type liveness $=$ program liveness

	Program terminates	Program does not terminate
Type live	✓ Program live	?
Type not live	× Program not live	× Program not live

Tool: Godel-Checker



https://github.com/nickng/gospal

https://bitbucket.org/MobilityReadingGroup/godel-checker

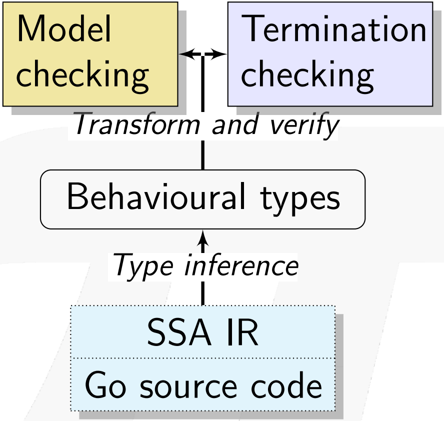
Understanding Concurrency with Behavioural Types

GolangUK Conference 2017

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.

	Programs	LoC	# states	ψ_g	ψ_l	ψ_s	ψ_e	Godel Checker Infer	Live	Live+CS	Term	dingo-hunter [36] Live	Time	gopherlyzer [40] DF	Time	GoInfer/Gong [30] Live	CS	Time
1	mismatch [36]	29	53	×	×	✓	✓	620.7	996.8	996.7	✓	×	639.4	×	3956.4	×	✓	616.8
2	fixed [36]	27	16	✓	✓	✓	✓	624.4	996.5	996.3	✓	✓	603.1	✓	3166.3	✓	✓	601.0
3	fanin [36, 39]	41	39	✓	✓	✓	✓	631.1	996.2	996.2	✓	✓	608.9	✓	19.8	✓	✓	696.7
4	sieve [30, 36]	43	∞		n/a			-	-	-	n/a	n/a	-	n/a	-	✓	✓	778.3
5	philo [40]	41	65	×	×	✓	✓	6.1	996.5	996.6	✓	×	34.2	×	27.0	×	✓	16.8
6	dinephil3 [13, 33]	55	3838	✓	✓	✓	✓	645.2	996.4	996.3	✓	n/a	-	n/a	-	✓	✓	13.2 min
7	starvephil3	47	3151	×	×	✓	✓	628.2	996.5	996.5	✓	n/a	-	n/a	-	×	✓	3.5 min
8	sel [40]	22	103	×	×	✓	✓	4.2	996.7	996.6	✓	×	15.3	×	13.0	×	✓	50.5
9	selFixed [40]	22	20	✓	✓	✓	✓	4.0	996.3	996.4	✓	✓	14.9	✓	3168.3	✓	✓	13.1
10	jobsched [30]	43	43	✓	✓	✓	✓	632.7	996.7	1996.1	✓	n/a	-	✓	4753.6	✓	✓	635.2
11	forselect [30]	42	26	✓	✓	✓	✓	623.3	996.4	996.3	✓	✓	611.8	n/a	-	✓	✓	618.6
12	cond-recur [30]	37	12	✓	✓	✓	✓	4.0	996.2	996.2	✓	✓	9.4	n/a	-	✓	✓	14.7
13	concsys [42]	118	15	×	×	✓	✓	549.7	996.5	996.4	✓	n/a	-	×	5278.6	×	✓	521.3
14	alt-bit [30, 35]	70	112	✓	✓	✓	✓	634.4	996.3	996.3	✓	n/a	-	n/a	-	✓	✓	916.8
15	prod-cons	28	106	✓	×	✓	✓	4.1	996.4	1996.2	✓	×	10.1	×	30.1	×	✓	21.8
16	nonlive	16	8	✓	✓	✓	✓	630.1	996.6	996.5	timeout	⊗	613.6	n/a	-	⊗	✓	613.8
17	double-close	15	17	✓	✓	×	✓	3.5	996.6	1996.6	✓	⊠	8.7	⊠	11.8	✓	×	9.1
18	stuckmsg	8	4	✓	✓	✓	×	3.5	996.6	996.6	✓	n/a	-	n/a	-	✓	✓	7.6

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)

www.scribble.org

Home | Getting Started | Downloads | Documentation | Community

Scribble: Describing Multi Party Protocols

Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data. However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

Describe

Scribble is a language for describing multiparty protocols from a global, or endpoint neutral, perspective.

Verify

Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that protocols described using the language are sound, and do not suffer from deadlocks or livelocks.

Project

Endpoint projection is the term used for identifying the responsibility of a particular role (or endpoint) within a protocol.

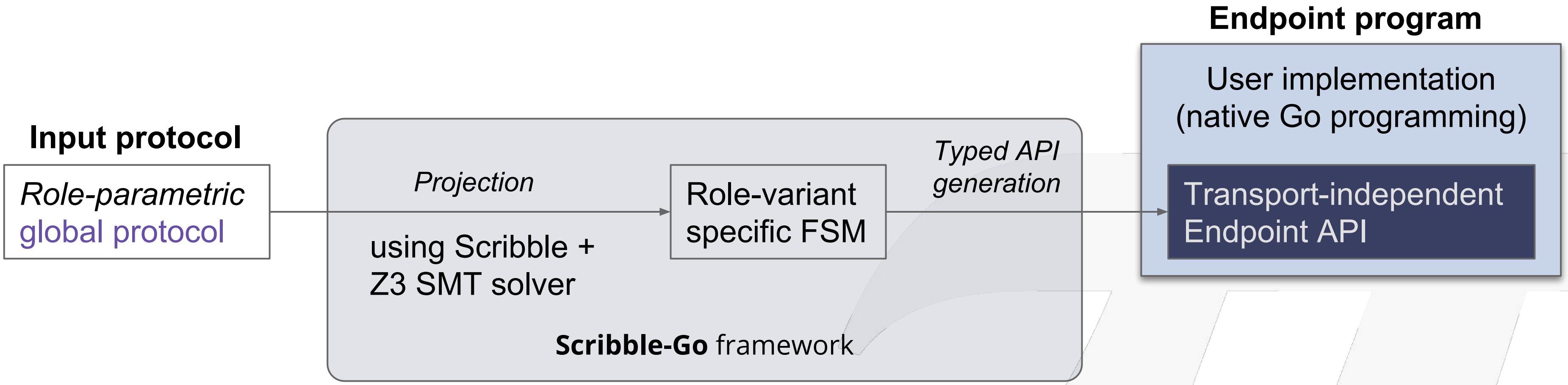
Implement

Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b) using session type APIs to clearly describe the behaviour, and (c) statically verify the code against the projection.

Monitor

Use the endpoint projection for roles defined within a Scribble protocol, to monitor the activity of a particular endpoint, to ensure it correctly implements the expected behaviour.

Scribble-Go workflow



1. Write a *role-parametric* global protocol
2. Select endpoint *role variant* to implement (e.g. `Fetcher`)
3. Use **Scribble-Go** to project and generate **Endpoint API**
4. Implement endpoint (e.g. `Fetcher[3]`) using the **Endpoint API**

Role variant

Role variant are *unique kinds* of endpoints

`{M, F[1..`***`n`***`], Server}`

If `F[1]` sends an extra request

HTTP HEAD to Server to get total size

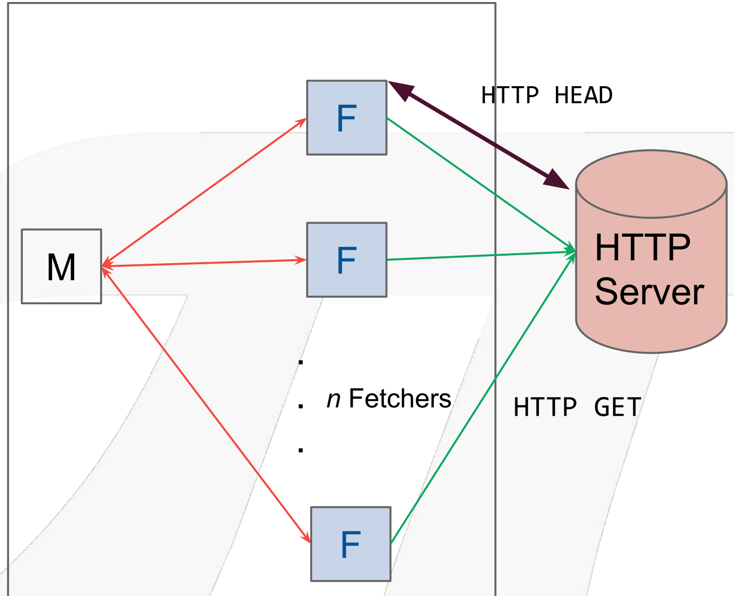
Then acts as a normal `F`

The role variants are:

`{M, F[1], F[2..`***`n`***`], Server}`

→ `F[1]` and `F[2..`***`n`***`]` are different endpoints

Inference of role variants (indices): formulated as SMT constraints for Z3

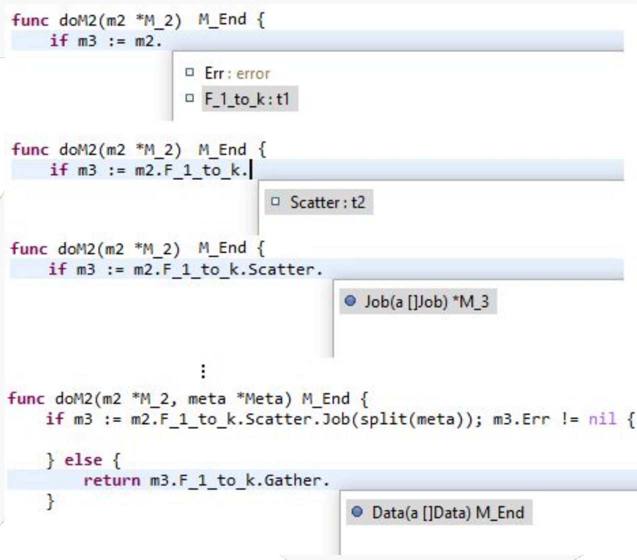


A concurrent file downloader (v2)

Endpoint API generation and usage

FSMs from local protocols → Message passing API

- Fluent-style
  - Every state is a unique type (struct)
  - Method calls (communication) returns next state
- Type information can be leveraged by IDEs
  - “dot-driven” content assist & auto complete



Behavioural Types for Go

Type syntax

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

- 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)

Semantics of types

$$\boxed{\text{SND}}\ \overline{a}; T \xrightarrow{\overline{a}} T \qquad \boxed{\text{RCV}}\ a; T \xrightarrow{a} T \qquad \boxed{\text{TAU}}\ \tau; T \xrightarrow{\tau} T$$

$$\boxed{\text{END}}\ \texttt{close}\, a; T \xrightarrow{\texttt{clo}\, a} T \qquad \boxed{\text{BUF}}\ \lfloor a \rceil_k^n \xrightarrow{\overline{\texttt{clo}}\, a} buf[a]_{closed} \qquad \boxed{\text{CLD}}\ buf[a]_{closed} \xrightarrow{a^*} buf[a]_{closed}$$

$$\boxed{\text{SEL}}\ \frac{i \in \{1,2\}}{T_1 \oplus T_2 \xrightarrow{\tau} T_i} \qquad \boxed{\text{BRA}}\ \frac{\alpha_j; T_j \xrightarrow{\alpha_j} T_j \quad j \in I}{\&\{\alpha_i; T_i\}_{i \in I} \xrightarrow{\alpha_j} T_j}$$

$$\boxed{\text{PAR}}\ \frac{T \xrightarrow{\alpha} T'}{T \mid S \xrightarrow{\alpha} T' \mid S} \qquad \boxed{\text{SEQ}}\ \frac{T \xrightarrow{\alpha} T'}{T; S \xrightarrow{\alpha} T'; S} \qquad \boxed{\text{TERM}}\ \mathbf{0}; S \xrightarrow{\tau} S$$

$$\boxed{\text{COM}}\ \frac{\alpha \in \{\overline{a}, a^*, a^{\bullet}\} \quad T \xrightarrow{\alpha} T' \quad S \xrightarrow{\beta} S' \quad \beta \in \{{}^{\bullet}a, a\}}{T \mid S \xrightarrow{\tau_a} T' \mid S'}$$

$$\boxed{\text{EQ}}\ \frac{T \equiv_\alpha T' \quad T \xrightarrow{\alpha} T''}{T' \xrightarrow{\alpha} T''} \qquad \boxed{\text{DEF}}\ \frac{T\{\tilde{a}/\tilde{x}\} \xrightarrow{\alpha} T' \quad \mathbf{t}(\tilde{x}) = T}{\mathbf{t}\langle \tilde{a} \rangle \xrightarrow{\alpha} T'}$$

$$\boxed{\text{CLOSE}}\ \frac{T \xrightarrow{\texttt{clo}\, a} T' \quad S \xrightarrow{\overline{\texttt{clo}}\, a} S'}{T \mid S \xrightarrow{\tau} T' \mid S'} \qquad \boxed{\text{IN}}\ \frac{k < n}{\lfloor a \rceil_k^n \xrightarrow{{}^{\bullet}a} \lfloor a \rceil_{k+1}^n} \qquad \boxed{\text{OUT}}\ \frac{k \geq 1}{\lfloor a \rceil_k^n \xrightarrow{a^{\bullet}} \lfloor a \rceil_{k-1}^n}$$

Barb predicates for types

$$a; T \downarrow_a \qquad \texttt{close}\ a; T \downarrow_{\texttt{clo}\ a} \qquad \frac{\forall i \in \{1,\ldots,n\} : \alpha_i \downarrow_{o_i}}{\&\{\alpha_i; T\}_{i\in\{1,\ldots,n\}} \downarrow_{\{o_1\ldots o_n\}}}$$

$$\overline{a}; T \downarrow_{\overline{a}} \qquad buf[a]_{closed} \downarrow_{a^*}$$

$$\frac{T \downarrow_o}{T; T' \downarrow_o} \qquad \frac{T \downarrow_a \quad T' \downarrow_{\overline{a}} \text{ or } T' \downarrow_{a^*}}{T \mid T' \downarrow_{\tau_a}} \qquad \frac{T\{\tilde{a}/\tilde{x}\} \downarrow_o \quad \mathbf{t}(\tilde{x}) = T}{\mathbf{t}\langle\tilde{a}\rangle \downarrow_o}$$

$$\frac{T \downarrow_a \quad \alpha_i \downarrow_{\overline{a}}}{T \mid \&\{\alpha_i; S_i\}_{i\in I} \downarrow_{\tau_a}} \qquad \frac{T \downarrow_{\overline{a}} \text{ or } T \downarrow_{a^*} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i; S_i\}_{i\in I} \downarrow_{\tau_a}}$$

$$\frac{k < n}{\lfloor a \rfloor_k^n \downarrow_{\bullet a}} \qquad \frac{k \geq 1}{\lfloor a \rfloor_k^n \downarrow_{a\bullet}} \qquad \frac{T \downarrow_{\overline{a}} \quad T' \downarrow_{\bullet a}}{T \mid T' \downarrow_{\tau_a}} \qquad \frac{T \downarrow_{a\bullet} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i; S_i\}_{i\in I} \downarrow_{\tau_a}}$$

$$\frac{T \downarrow_o}{T \mid T' \downarrow_o} \qquad \frac{T \downarrow_o \quad a \notin \mathrm{fn}(o)}{(\mathrm{new}^n\ a); T \downarrow_o} \qquad \frac{T \downarrow_o \quad T \equiv T'}{T \downarrow_o}$$

Figure: Barb predicates for types.