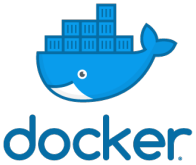
A Static Verification Framework for Message Passing in Go using Behavioural Types

Julien Lange[1], Nicholas Ng[2],
Bernardo Toninho[3], Nobuko Yoshida[2]

[1]University of Kent [2]Imperial College London
[3]Universidade Nova de Lisboa

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, e.g.



UBER

NETFLIX

Go and concurrency

Approach and philosophy

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

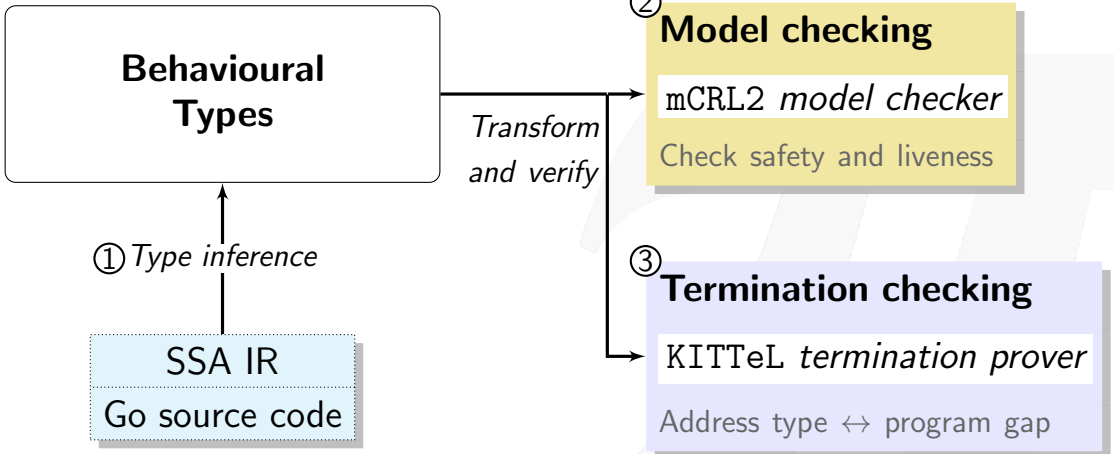
Encourages message passing over locking

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

Inspired by Hoare's CSP/**process calculi**

Static verification framework for Go

Overview



Concurrency in Go 

Goroutines

```
1  func main() {
2    ch := make(chan string)
3    go send(ch)
4    print(<-ch)
5    close(ch)
6  }
7
8  func send(ch chan string) {
9    ch <- "Hej ICSE!"
10 }
```

`go` keyword + function call

- Spawns function as goroutine
- Runs in parallel to parent

Concurrency in Go 

Channels

```
1  func main() {
2    ch := make(chan string)
3    go send(ch)
4    print(<-ch)
5    close(ch)
6  }
7
8  func send(ch chan string) {
9    ch <- "Hej ICSE!"
10 }
```

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

```
1  func main() {
2    ch := make(chan string)
3    go send(ch)
4    print(<-ch)
5    close(ch)
6  }
7
8  func send(ch chan string) {
9    ch <- "Hej ICSE!"
10 }
```

Also `select-case`:

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

Concurrency in Go 

Deadlock detection

```
1  func main() {
2    ch := make(chan string)
3    go send(ch)
4    print(<-ch)
5    close(ch)
6  }
7
8  func send(ch chan string) {
9    ch <- "Hej ICSE!"
10 }
```

- Send message thru channel
- Print message on screen

Output:

```
$ go run hello.go
Hej ICSE!
$
```

Concurrency in Go 

Deadlock detection

Missing 'go' keyword

```
1  // import _ "net"
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 <- "Hej ICSE"
11 }
```

- 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

```
1  // import _ "net"
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 <- "Hej ICSE"
11 }
```

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

```
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 <- "Hej ICSE"
11 }
```

Import unused, unrelated package

Concurrency in Go 

Deadlock detection

Missing 'go' keyword

```
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 <- "Hej ICSE"
11 }
```

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

Output:

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

Hangs: Deadlock **NOT** detected

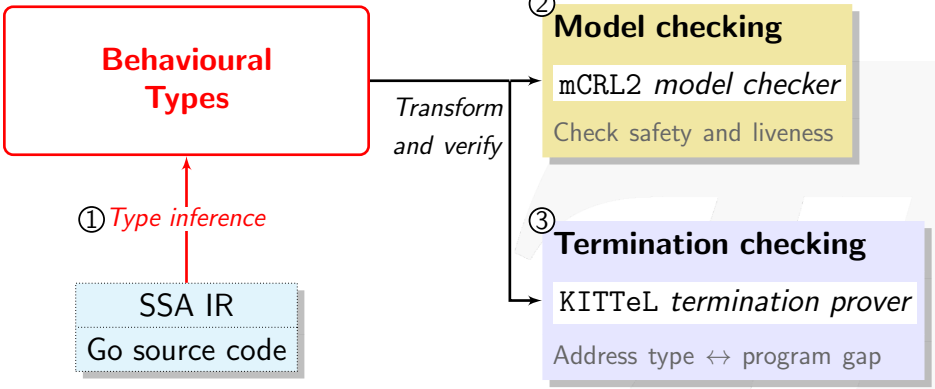
Our goal

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

```
1  func main() {
2    ch := make(chan int)
3    go send(ch)
4    print(<-ch)
5    close(ch)
6  }
7
8  func send(c chan int) {
9    c <- 1
10 }
```

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

```
1  func main() {
2    ch := make(chan int)
3    go send(ch)
4    print(<-ch)
5    close(ch)
6  }
7
8  func send(c chan int) {
9    c <- 1
10 }
```

→

Inferred Behavioural Types

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

Infer Behavioural Types from Go Program

Go source code

```
1  func main() {
2    ch := make(chan int)
3    go send(ch)
4    print(<-ch)
5    close(ch)
6  }
7
8  func send(c chan int) {
9    c <- 1
10 }
```

Inferred Behavioural Types

create channel → $\mathsf{main}() = (\mathtt{new}\ ch);$

spawn → $(\mathsf{send}\langle \mathsf{ch} \rangle \mid$

receive → $\mathsf{ch};$

close → $\mathtt{close}\ \mathsf{ch}),$

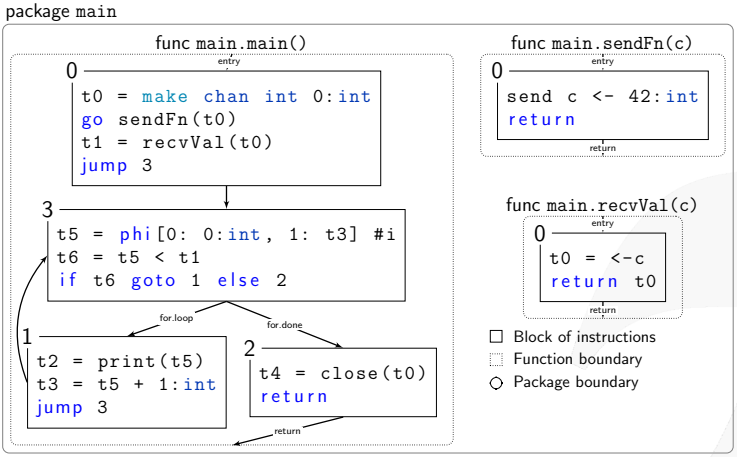
$\mathsf{send}(ch) = \overline{\mathsf{ch}}$

send

Infer Behavioural Types from Go Program

```
1  func main() {
2      ch := make(chan int) // Create channel
3      go sendFn(ch)        // Run as goroutine
4      x := recvVal(ch)     // Function call
5      for i := 0; i < x; i++ {
6          print(i)
7      }
8      close(ch) // Close channel
9  }
10 func sendFn(c chan int)   { c <- 3 }    // Send to c
11 func recvVal(c chan int) int { return <-c } // Recv from c
```

Infer Behavioural Types from Go Program



Analyse in

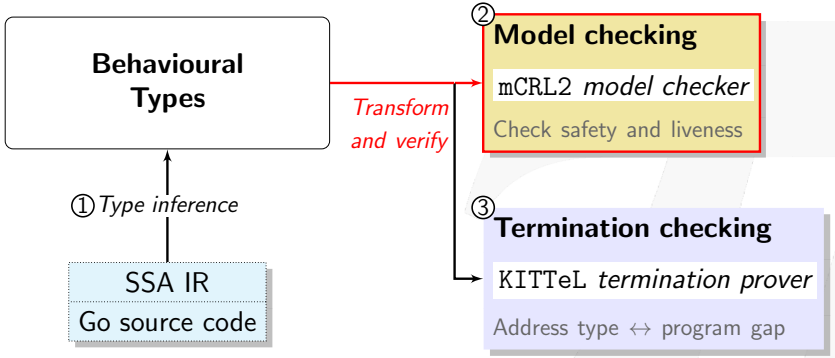
Static **S**ingle **A**ssignment

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 CCS semantics, i.e.

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

Send on channel a

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

Synchronise on a

$a; T \xrightarrow{a} T$

Receive on channel a

Behavioural Types as **LTS model**

Standard CCS semantics, i.e.

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

Send on channel a

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

Synchronise on a

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

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

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

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

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: Liveness (partial deadlock freedom)

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

$\mathcal{A}$ = set of initialised channels

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)

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

where:

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

If a channel is ready to receive or send,

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

Property: Liveness (partial deadlock freedom)

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

```
1  func main() {
2      ch := make(chan int)
3      go looper() // !!!
4      <-ch        // No matching send
5  }
6  func looper() {
7      for {
8      }
9  }
```

× Runtime detector: Hangs

✓ Our tool: NOT live

Property: Liveness (partial deadlock freedom)

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

```
1  func main() {
2      ch := make(chan int)
3      go loopSend(ch)
4      <-ch
5  }
6  func loopSend(ch chan int) {
7      for i := 0; i < 10; i-- {
8          // Does not terminate
9      }
10     ch <- 1
11 }
```

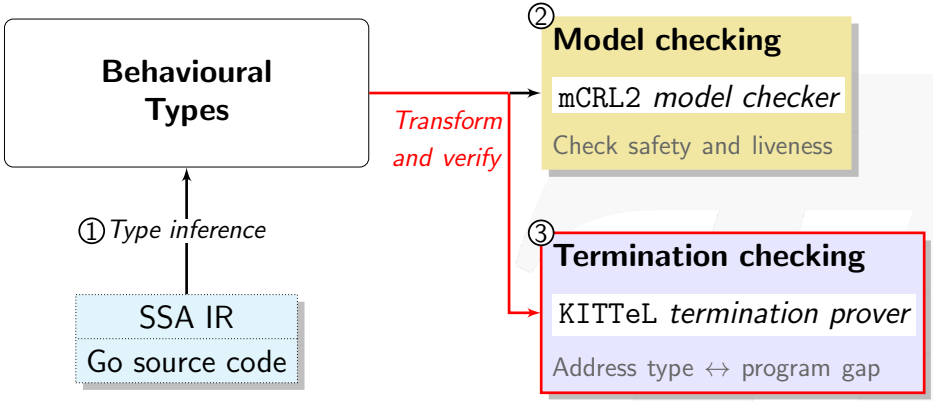
What about this one?

- Type: Live
- Program: NOT live

Needs additional guarantees

Termination checking

Addressing the program-type *abstraction gap*



Termination checking with KITTeL

Type inference does not consider *program data*

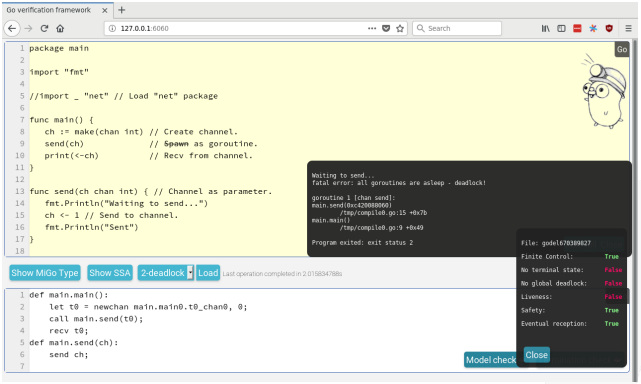
- Type liveness $\neq$ Program liveness if program non-terminating
- Especially when involving iteration

$\Rightarrow$ 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

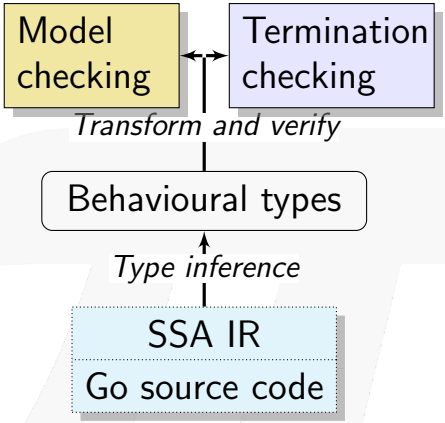
Understanding Concurrency with Behavioural Types

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.

							Godel Checker				dingo-hunter [36]		gopherlyzer [40]		GoInfer/Gong [30]		
	Programs	LoC	# states	ψ_g	ψ_l	ψ_s	ψ_e	Infer	Live	Live+CS	Term	Live	Time	DF	Time	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	stuckmse	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)

Property: Global deadlock freedom

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

```
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 <- "Hej ICSE"
11 }
```

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

Property: Global deadlock freedom

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

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

Property: Global deadlock freedom

$$\bigwedge_{a \in \mathcal{A}} (\downarrow_a \vee \downarrow_{\bar{a}} \implies \langle \mathbb{A} \rangle \texttt{true})$$

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 $\quad$ $\mathbb{A}$ = set of all labels
$\Rightarrow$ Ready receive/send = not end of program.

Property: Channel safety

$$\bigwedge_{a \in \mathcal{A}} (\downarrow_{a^*} \implies \neg(\downarrow_{\bar{a}} \vee \downarrow_{\mathtt{clo}\, a}))$$

```
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  }
```

Property: Channel safety

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

```
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  }
```

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

Property: Channel safety

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

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

Property: Liveness (select)

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

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

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

Property: Eventual reception

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

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

(i.e. no orphan messages)

Behavioural Types for Go

Type syntax

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

- 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)