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.
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
Overview

Behavioral Types

Type inference

SSA IR

Go source code

Transform and verify

Model checking

mCRL2 model checker

Check safety and liveness

Termination checking

KITTeL termination prover

Address type ↔ program gap

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types

mrg.doc.ic.ac.uk
Concurrency in Go 🌬

Goroutines

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

func send(ch chan string) {
    ch <- "Hej ICSE!"
}
```

- **go** keyword + function call
  - Spawns function as goroutine
  - Runs in parallel to parent
Concurrency in Go

Channels

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

func send(ch chan string) {
   ch <- "Hej ICSE!"
}
```

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 <- "Hej ICSE!"
}
```

Also `select-case`:
- Wait on multiple channel operations
- `switch-case` for communication
Concurrency in Go

Deadlock detection

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

func send(ch chan string) {
    ch <- "Hej ICSE!"
}
```

- Send message thru channel
- Print message on screen

Output:

```
$ go run hello.go
Hej ICSE!
$
```
Concurrency in Go 🧑‍💻

Deadlock detection

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

Output:

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

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

func send(ch chan string) {
    ch <- "Hej ICSE"
}
Concurrency in Go 🧟‍♂️

Deadlock detection

Go’s runtime deadlock detector
- Checks if all goroutines are blocked (‘global’ deadlock)
- Print message then crash
- Some packages disable it (e.g. `net`)

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

func send(ch chan string) {
  ch <- "Hej ICSE"
}
Concurrency in Go 🐸

Deadlock detection

Missing 'go' keyword

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

func send(ch chan string) {
    ch <- "Hej ICSE"
}
```

Import unused, unrelated package
Concurrent in Go 🦇
Deadlock detection

Missing 'go' keyword

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

func send(ch chan string) {
  ch <- "Hej ICSE"
}
```

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

Output:

```
$ go run deadlock2.go
Hangs: Deadlock **NOT** detected
```

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
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**

1. **Type inference**
   - Go source code
   - SSA IR

2. **Model checking**
   - mCRL2 model checker
   - Check safety and liveness

3. **Termination checking**
   - KITTeL termination prover
   - Address type ↔ program gap

Overview
Concurrency in Go
**Behavioural type inference**
Model checking behavioural types
Termination checking
Summary
Infer Behavioural Types from Go Program

Go source code

```go
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

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

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

Inferred Behavioural Types

```
main() = (new ch); (send(ch) | ch; close ch),

send(ch) = \overline{ch}
```
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

- `main()` creates a new channel `ch` and spawns a new channel `ch` using `send(ch)`. It then receives a value from `ch` and closes `ch`.

- `send(ch)` sends a value `ch`.

- `create channel` is associated with `main()`.
- `spawn` is associated with `send(ch)`.
- `receive` is associated with the print operation on `<-ch`.
- `close` is associated with `close(ch)`.
- `send` is associated with `c <- 1`.

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
mrg.doc.ic.ac.uk
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

func main.main()

entry

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

3

func main.sendFn(c)

entry

0

send c <- 42:int

return

func main.recvVal(c)

entry

0

t0 = <-c

return

for loop

Block of instructions

Function boundary

Package boundary

Analyse in

Static Single Assignment

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**

1. Type inference

   SSA IR
   Go source code

2. Model checking

   mCRL2 *model checker*
   Check safety and liveness

3. Termination checking

   KITTeL *termination prover*
   Address type $\leftrightarrow$ program gap
Model checking behavioural types

\[ M \models \phi \]

- **LTS model**: inferred type + type semantics
- **Safety/liveness properties**: $\mu$-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 \textbf{LTS model}

Standard CCS semantics, i.e.

\[ \bar{a}; T \xrightarrow{\bar{a}} T \]

Send on channel \( a \)

\[ T \xrightarrow{\tau_a} T' \quad S \xrightarrow{a} S' \quad T \mid S \xrightarrow{\tau_a} T' \mid S' \]

Synchronise on \( a \)

\[ a; T \xrightarrow{a} T \]

Receive on channel \( a \)
Behavioural Types as **LTS model**

Standard CCS semantics, i.e.

\[
\begin{align*}
\bar{a}; T & \xrightarrow{\bar{a}} T \\
T & \xrightarrow{a} T' \\
S & \xrightarrow{a} S' \\
\tau a; T & \xrightarrow{T'} \\
T | S & \xrightarrow{\tau a} T' | S'
\end{align*}
\]

Send on channel \(a\)  \hspace{2cm} \text{Synchronise on } \(a\)  \hspace{2cm} \text{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]
- \( \mu \)-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

\[ \overline{a}; T \downarrow a \]

\[ \begin{align*}
T \downarrow a & \quad \quad T' \downarrow a \\
\frac{}{T \parallel T' \downarrow \tau a} & \quad \quad a; T \downarrow a
\end{align*} \]

Ready to send  Ready to synchronise  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

\[ \overline{a}; T \downarrow \overline{a} \quad T \downarrow \overline{a} \quad T' \downarrow a \quad a; T \downarrow a \]

Ready to send  Ready to synchronise  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 $\mu$-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)

\[ \bigwedge_{a \in \mathcal{A}} (\downarrow a \lor \downarrow \bar{a} \implies \text{eventually } (\langle \tau_a \rangle \text{true})) \]

\[ \mathcal{A} = \text{set of initialised channels} \]

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

(i.e. there’s corresponding send for receiver/recv for sender)
Property: Liveness (partial deadlock freedom)

\[ \bigwedge_{a \in A} \left( \downarrow a \lor \downarrow a = \rightarrow \text{eventually} \left( \langle \tau_a \rangle \text{true} \right) \right) \]

where:

\[
\text{eventually} \left( \phi \right) \overset{\text{def}}{=} \mu y. \left( \phi \lor \langle A \rangle y \right)
\]

If a channel is ready to receive or send, then for some reachable state it can synchronise \((\tau_a)\).
Property: Liveness (partial deadlock freedom)

\[ \bigwedge_{a \in A} (\downarrow a \lor \downarrow \overline{a} \implies \text{eventually} (\langle \tau_a \rangle \text{true})) \]

```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)

\[ \bigwedge_{a \in A} (\downarrow a \lor \downarrow \overline{a} \implies \text{eventually} (\langle \tau_a \rangle \text{true})) \]

What about this one?

- Type: Live
- Program: NOT live

Needs additional guarantees
Termination checking

Addressing the program-type abstraction gap

1. Type inference
   - SSA IR
   - Go source code

2. Model checking
   - mCRL2 model checker
   - Check safety and liveness

3. Termination checking
   - KITTeL termination prover
   - Address type ↔ program gap

Transform and verify
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

<table>
<thead>
<tr>
<th></th>
<th>Program terminates</th>
<th>Program does not terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type live</td>
<td>✓ Program live</td>
<td>?</td>
</tr>
<tr>
<td>Type not live</td>
<td>$\times$ Program not live</td>
<td>$\times$ Program not live</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Programs</th>
<th>LoC</th>
<th># states</th>
<th>$\psi_g$</th>
<th>$\psi_l$</th>
<th>$\psi_s$</th>
<th>$\psi_e$</th>
<th>Infer</th>
<th>Live</th>
<th>Live+CS</th>
<th>Term</th>
<th>dingo-hunter [36]</th>
<th>gopherlyzer [40]</th>
<th>Golnfer/Gong [30]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Live</td>
<td>Time</td>
<td>DF</td>
</tr>
<tr>
<td>mismatch</td>
<td>29</td>
<td>53</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>620.7</td>
<td>996.8</td>
<td>996.7</td>
<td>✓</td>
<td>×</td>
<td>639.4</td>
<td>×</td>
</tr>
<tr>
<td>fixed</td>
<td>27</td>
<td>16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>624.4</td>
<td>995.6</td>
<td>996.3</td>
<td>✓</td>
<td>✓</td>
<td>603.1</td>
<td>✓</td>
</tr>
<tr>
<td>fanin</td>
<td>41</td>
<td>39</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>631.1</td>
<td>996.2</td>
<td>996.2</td>
<td>✓</td>
<td>✓</td>
<td>608.9</td>
<td>✓</td>
</tr>
<tr>
<td>sieve</td>
<td>43</td>
<td>∞</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>-</td>
<td>n/a</td>
<td></td>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
</tr>
<tr>
<td>philo</td>
<td>41</td>
<td>65</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>6.1</td>
<td>996.5</td>
<td>996.6</td>
<td>✓</td>
<td>×</td>
<td>34.2</td>
<td>×</td>
</tr>
<tr>
<td>dinephil</td>
<td>55</td>
<td>3838</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>645.2</td>
<td>996.4</td>
<td>996.3</td>
<td>✓</td>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
</tr>
<tr>
<td>starvphi1</td>
<td>47</td>
<td>3151</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>628.2</td>
<td>996.5</td>
<td>996.5</td>
<td>✓</td>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
</tr>
<tr>
<td>sel</td>
<td>22</td>
<td>103</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>4.2</td>
<td>996.7</td>
<td>996.6</td>
<td>✓</td>
<td>×</td>
<td>15.3</td>
<td>×</td>
</tr>
<tr>
<td>selfixed</td>
<td>22</td>
<td>20</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>4.0</td>
<td>996.3</td>
<td>996.4</td>
<td>✓</td>
<td>✓</td>
<td>14.9</td>
<td>✓</td>
</tr>
<tr>
<td>jobsched</td>
<td>43</td>
<td>43</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>632.7</td>
<td>996.7</td>
<td>1996.1</td>
<td>✓</td>
<td>n/a</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>forselect</td>
<td>42</td>
<td>26</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>623.3</td>
<td>996.4</td>
<td>996.3</td>
<td>✓</td>
<td>n/a</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>cond-recu</td>
<td>37</td>
<td>12</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>4.0</td>
<td>996.2</td>
<td>996.2</td>
<td>✓</td>
<td>9.4</td>
<td>n/a</td>
<td>✓</td>
</tr>
<tr>
<td>concsys</td>
<td>118</td>
<td>15</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>549.7</td>
<td>996.5</td>
<td>996.4</td>
<td>✓</td>
<td>n/a</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>alt-bit</td>
<td>70</td>
<td>112</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>634.4</td>
<td>996.3</td>
<td>996.3</td>
<td>✓</td>
<td>n/a</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>prod-cons</td>
<td>28</td>
<td>106</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>4.1</td>
<td>996.4</td>
<td>1996.2</td>
<td>✓</td>
<td>×</td>
<td>10.1</td>
<td>×</td>
</tr>
<tr>
<td>nonlive</td>
<td>16</td>
<td>8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>630.1</td>
<td>996.6</td>
<td>996.5</td>
<td>timeout</td>
<td>613.6</td>
<td>n/a</td>
<td>-</td>
</tr>
<tr>
<td>double-close</td>
<td>15</td>
<td>17</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3.5</td>
<td>996.6</td>
<td>1996.6</td>
<td>✓</td>
<td>8.7</td>
<td>11.8</td>
<td>✓</td>
</tr>
<tr>
<td>stuckmsg</td>
<td>8</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3.5</td>
<td>996.6</td>
<td>996.6</td>
<td>✓</td>
<td>n/a</td>
<td>-</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

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

\[ \bigwedge_{a \in A} (\downarrow a \lor \downarrow \overline{a} \implies \langle A \rangle \text{true}) \]

```go
import _ "net" // unused

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

func send(ch chan string) {
    ch <- "Hej ICSE"
}
```

- Send (\(\downarrow \overline{ch}\): line 10)
- No synchronisation
- No more reduction
Property: Global deadlock freedom

\[ \bigwedge_{a \in \mathcal{A}} (\downarrow a \lor \downarrow \overline{a} \implies \langle A \rangle \text{true}) \]

If a channel \( a \) is ready to receive or send, then there must be a **next state** (i.e. not stuck)

\( \mathcal{A} = \text{set of all initialised channels} \) \hspace{1cm} \mathcal{A} = \text{set of all labels}
Property: Global deadlock freedom

\[ \bigwedge_{a \in \mathcal{A}} (\downarrow a \lor \downarrow \overline{a} \implies \langle \mathcal{A} \rangle \text{true}) \]

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

\[ \bigwedge_{a \in A} (\downarrow a^* \implies \neg (\downarrow \overline{a} \lor \downarrow \text{clo} a)) \]

```go
func main() {
    ch := make(chan int)
    go func(ch chan int) {
        ch <- 1 // is ch closed?
    }(ch)
    close(ch)
    <-ch
}
```
Property: Channel safety

\[ \bigwedge_{a \in \mathcal{A}} (\downarrow a^* \implies \neg (\downarrow \overline{a} \lor \downarrow \text{clo} a)) \]

```go
func main() {
    ch := make(chan int)
    go func(ch chan int) {
        ch <- 1 // is ch closed?
    }(ch)
    close(ch)
    <-ch
}
```

- \(\downarrow \text{clo} \text{ch}\) when \(\text{close(ch)}\)
- \(\downarrow \text{ch}*\) fires after closed
- Send (\(\downarrow \overline{\text{ch}}\): line 4)
Property: Channel safety

\[ \bigwedge_{a \in A} (\downarrow a^* \implies \neg (\downarrow a \lor \downarrow \text{clo } a)) \]

Once a channel \( a \) is closed (\( a^* \)), it will not be sent to, nor closed again (\text{clo } a)
Property: Liveness (select)

\[ \bigwedge_{\tilde{a} \in \mathcal{P}(A)} (\downarrow \tilde{a} \iff \text{eventually} (\langle \{\tau_a \mid a \in \tilde{a}\} \rangle \text{true})) \]

"If one of the channels in select is ready to receive or send, then eventually it will synchronise (\(\tau_a\)) (i.e. there's corresponding send for receiver/recv for sender)"
Property: Eventual reception

\[ \bigwedge_{a \in A} (\downarrow a \Rightarrow \text{eventually } (\langle \tau_a \rangle \text{true})) \]

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

(i.e. no orphan messages)
Behavioural Types for Go

Type syntax

\[ \alpha \ := \ \overline{u} \mid u \mid \tau \]

\[ T, S \ := \ \alpha; T \mid T \oplus S \mid \&\{\alpha_i; T_i\}_{i \in I} \mid (T \mid S) \mid 0 \]

\[ \| (\text{new } a)T \mid \text{close } u; T \mid t\langle \tilde{u} \rangle \mid \lfloor u \rfloor^k \mid \text{buf}[u]_{\text{closed}} \]

\[ T \ := \ \{t(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S \]

- 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)