

Behavioural Type-Based Static Verification Framework

for

GO



Julien Lange



Nicholas Ng



Bernardo
Toninho



Nobuko
Yoshida





ETAPS 2019 TEST-OF-TIME AWARD

[Language primitives and type discipline for structured communication-based programming](#) by Kohei Honda, Vasco T. Vasconcelos and Makoto Kubo

POPL 2008 MOST INFLUENTIAL PAPER AWARD



POPL 2008 Most Influential Paper Award

Kohei Honda, Nobuko Yoshida and Marco Carbone

Multiparty asynchronous session types





LICS 2018 TEST OF TIME AWARD

LICS'98

A fully abstract game semantics for general references

Samson Abramsky Kohei Honda

LFCS, University of Edinburgh

{samson, kohei}@dcs.ed.ac.uk

Guy McCusker

St John's College, Oxford

mccusker@comlab.ox.ac.uk

Abstract

A games model of a programming language with higher-order store in the style of ML-references is introduced. The category used for the model is obtained by relaxing certain behavioural conditions on a category of games previously used to provide fully abstract models of pure functional languages. The model is shown to be fully abstract by means of factorization arguments which reduce the question of definability for the language with higher-order store to that for its purely functional fragment.

Mobility Research Group



π -calculus, Session Types research at Imperial College

Home

People

Publications

Grants

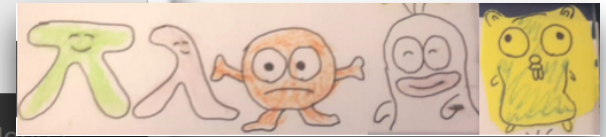
Talks

Tutorials

Tools

Awards

Kohei Honda



NEWS

The paper *Multiparty asynchronous session types* by Kohei Honda, Nobuko Yoshida, and Marco Carbone, published in POPL 2008 has been awarded the ACM SIGPLAN Most Influential POPL Paper Award today at POPL 2018.

» more

10 Jan 2018

Estafet has published a page on their usage of the Scribble language developed in our group with RedHat and other industry partners.

» more

25 Sep 2017

Nick spoke at Golang UK 2017 on applying behavioural types to verifv concurrent Go programs.

SELECTED PUBLICATIONS

2018

Julien Lange , Nicholas Ng , Bernardo Toninho , Nobuko Yoshida : [A Static Verification Framework for Message Passing in Go using Behavioural Types](#). *To appear in ICSE 2018* .

Bernardo Toninho , Nobuko Yoshida : [Depending On Session Typed Process](#). *To appear in FoSSaCS 2018* .

Bernardo Toninho , Nobuko Yoshida : [On Polymorphic Sessions And Functions: A Talk of Two \(Fully Abstract\) Encodings](#). *To appear in ESOP 2018* .

Rumyana Neykova , Raymond Hu , Nobuko Yoshida , Fahd Abdeljallal : [Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#](#). *To appear in CC 2018* .

Post-docs:

Simon CASTELLAN

David CASTRO

Francisco FERREIRA

Raymond HU

Rumyana NEYKOVA

Nicholas NG

Alceste SCALAS

PhD Students:

Assel ALTAYEVA

Juliana FRANCO

Eva GRAVERSEN



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.

POPL 2019 Research Papers

[Write a Blog >>](#)

15:21 - 16:27: [Research Papers - Capabilities and Session Types I at POPL Track 2](#)


15:21 - 15:43 *Talk* ☆ [StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities](#)
Lau Skorstengaard , Dominique Devriese Vrije Universiteit Brussel, Belgium, Lars Birkedal Aarhus University

15:43 - 16:05 *Talk*  ☆ [Two sides of the same coin: Session Types and Game Semantics](#)
Simon Castelan Imperial College London, UK, Nobuko Yoshida Imperial College London
[DOI](#) [Pre-print](#)

16:05 - 16:27 *Talk* ☆ [Exceptional Asynchronous Session Types: Session Types without Tiers](#)
Simon Fowler The University of Edinburgh, Sam Lindley University of Edinburgh, UK, J. Garrett Morris University of Kansas, USA, Sara Décova
[Pre-print](#)

16:37 - 17:43: [Research Papers - Session Types II at POPL Track 2](#)

16:37 - 16:59 *Talk*  ☆ [Interconnectability of Session-Based Logical Processes](#) **TOPLAS**
Bernardo Toninho NOVA-LINCS, FCT/UNL, Nobuko Yoshida Imperial College London
[DOI](#) [Pre-print](#)

16:59 - 17:21 *Talk*  ☆ [Distributed Programming using Role-Parametric Session Types in Go](#)
David Castro Imperial College London, Raymond Hu Imperial College London, Sung-Shik Jongmans Open University of the Netherlands, Nicholas Ng Imperial College London, Nobuko Yoshida Imperial College London
[DOI](#) [Pre-print](#)

17:21 - 17:43 *Talk*  ☆ [Less is More: Multiparty Session Types Revisited](#)
Alceste Scalas Imperial College London, Nobuko Yoshida Imperial College London
[DOI](#) [Pre-print](#)



Selected Publications [2018-2019]

- [PLDI19] [Alceste Scalas](#), NY, Elias Benussi: [Verifying message-passing programs with dependent behavioural types](#).
- [CAV19] [Julien Lange](#), NY: [Verifying Asynchronous Interactions via Communicating Session Automata](#).
- [ECOOP19] [Rupak Majumdar](#), [Marcus Pirron](#), NY, and [Damien Zufferey](#), [Motion Session Types for Robotic Interactions](#)
- [FoSSaCs19] [Simon Castellan](#), NY: [Causality in Linear Logic](#)
- [ESOP19] [Laura Bocchi](#), [Maurizio Murgia](#), [Vasco T. Vasconcelos](#), NY: [Asynchronous Timed Session Types](#)
- [POPL19] [Simon Castellan](#), NY: [Two Sides of the Same Coin: Session Types and Game Semantics](#)
- [POPL19] [David Castro](#), [Raymond Hu](#), [Sung-Shik Jongmans](#), [Nicholas Ng](#), NY: [Distributed Programming Using Role Parametric Session Types in Go](#)
- [POPL19] [Alceste Scalas](#), [Nobuko Yoshida](#): [Less Is More: Multiparty Session Types Revisited](#)
- [POPL19] [Bernardo Toninho](#), NY: [Interconnectability of Session Based Logical Processes](#)
- [ICSE18] [Julien Lange](#), [Nicholas Ng](#), [Bernardo Toninho](#), NY: [A Static Verification Framework for Message Passing in Go using Behavioural Types](#)
- [LICS18] [Romain Demangeon](#), NY: [Causal Computational Complexity of Distributed Processes](#)
- [FoSSaCs18] [Bernardo Toninho](#), [Nobuko Yoshida](#): [Depending On Session Typed Process](#)
- [ESOP18] [Malte Viering](#), [Tzu-Chun Chen](#), [Patrick Eugster](#), [Raymond Hu](#), [Lukasz Ziarek](#): [A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems](#)
- [ESOP18] [Bernardo Toninho](#), NY: [On Polymorphic Sessions And Functions: A Tale of Two \(Fully Abstract\) Encodings](#)
- [CC18] [Rumyana Neykova](#), [Raymond Hu](#), NY, [Fahd Abdeljallal](#): [A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#](#)

Selected Publications [2018-2019]

- [PLDI19] [Alceste Scalas](#), NY, [Elias Benussi](#): [Verifying message-passing programs with dependent behavioural types](#).
- [CAV19] [Julien Lange](#), NY: [Verifying Asynchronous Interactions via Communicating Session Automata](#).
- [ECOOP19] [Rupak Majumdar](#), [Marcus Pirron](#), NY, and [Damien Zufferey](#), [Motion Session Types for Robotic Interactions](#)
- [FoSSaCs19] [Simon Castellan](#), NY: [Causality in Linear Logic](#)
- [ESOP19] [Laura Bocchi](#), [Maurizio Murgia](#), [Vasco T. Vasconcelos](#), NY: [Asynchronous Timed Session Types](#)
- [POPL19] [Simon Castellan](#), NY: [Two Sides of the Same Coin: Session Types and Game Semantics](#)
- [POPL19] [David Castro](#), [Raymond Hu](#), [Sung-Shik Jongmans](#), [Nicholas Ng](#), NY: [Distributed Programming Using Role Parametric Session Types in Go](#)
- [POPL19] [Alceste Scalas](#), [Nobuko Yoshida](#): [Less Is More: Multiparty Session Types Revisited](#)
- [POPL19] [Bernardo Toninho](#), NY: [Interconnectability of Session Based Logical Processes](#)
- [ICSE18] [Julien Lange](#), [Nicholas Ng](#), [Bernardo Toninho](#), NY: [A Static Verification Framework for Message Passing in Go using Behavioural Types](#)
- [LICS18] [Romain Demangeon](#), NY: [Causal Computational Complexity of Distributed Processes](#)
- [FoSSaCs18] [Bernardo Toninho](#), [Nobuko Yoshida](#): [Depending On Session Typed Process](#)
- [ESOP18] [Malte Viering](#), [Tzu-Chun Chen](#), [Patrick Eugster](#), [Raymond Hu](#), [Lukasz Ziarek](#): [A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems](#)
- [ESOP18] [Bernardo Toninho](#), NY: [On Polymorphic Sessions And Functions: A Tale of Two \(Fully Abstract\) Encodings](#)
- [CC18] [Rumyana Neykova](#), [Raymond Hu](#), NY, [Fahd Abdeljallal](#): [A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#](#)



GOLANG UK CONFERENCE

16th*, 17th & 18th AUGUST 2017

📍 The Brewery, London



Imperial College
London

Home College and Campus Science **Engineering** Health Business Search here... Go ▶

Go concurrency verification research at DoC grabs headline

POPL'17

the morning paper

ICSE'18

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

Home About Info QR Editions Subscribe

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

JANUARY 25, 2018

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

currency
rates a

tured in the
'high
interesting
easily
le of the
L (Principles

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 - **goroutines**
 - Selective **send/receive**

CSP_{80'}

FUN

Dropbox, Netflix, Docker, CoreOS

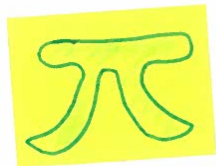
- ▶ **GO** has a runtime deadlock detector
- ▶ How can we detect *partial deadlock* and *channel errors* for realistic programs?
- ▶ Use *behavioural types* in process calculi
e.g. [ACM Survey, 2016] **185** citations, 6 pages
- ▶ Dynamic channel creations, unbounded thread creations, recursions, ..
- ▶ **Scalable** (synchronous/asynchronous) **Modular, Refinable**

▶  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, ..

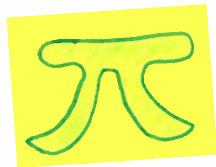
▶ **Scalable** (synchronous/asynchronous) **Modular, Refinable**



▶  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



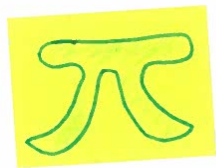
- ▶ Dyn.  ne , unbounded thread creations, recursions, ..
- ▶ **Scalable** (synchronous/asynchronous) **Modular, Refinable**

▶  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



186 ??

▶ channel creations, unbounded thread creations, recursions, ..

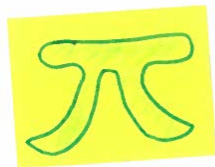
▶ **Scalable** (synchronous/asynchronous) · **Modular, Refinable**

▶  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 creation, ...



▶ **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)

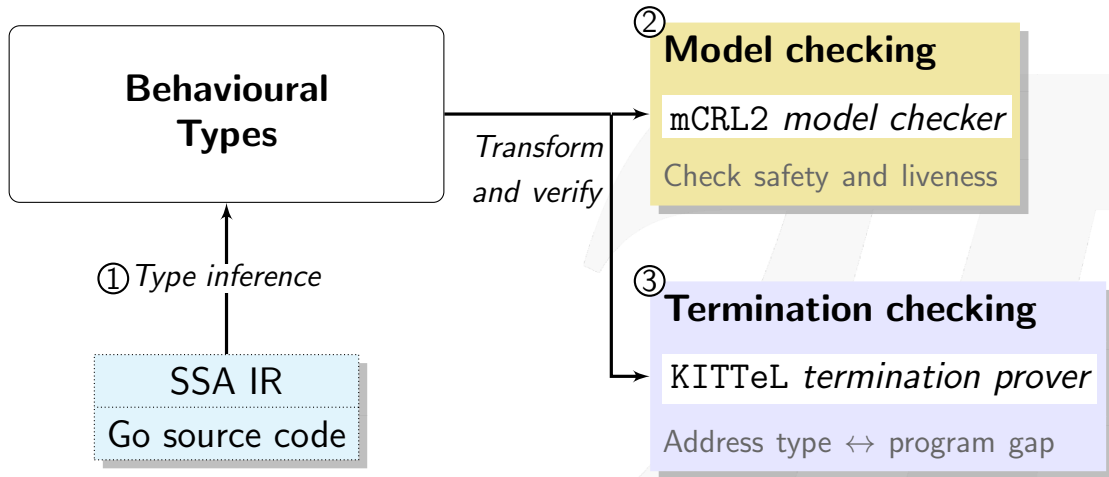
STEP 3

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



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 <- "Hello Kent!"
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 <- "Hello Kent!"
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 <- "Hello Kent!"
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 <- "Hello Kent!"
10 }

```

- Send message thru channel
- Print message on screen

Output:

```

$ go run hello.go
Hello Kent!
$

```

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 <- "Hello Kent!"
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 <- "Hello Kent!"
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 <- "Hello Kent"
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 <- "Hello Kent"
11 }

```

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

Output:

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

Hangs: Deadlock **NOT** detected

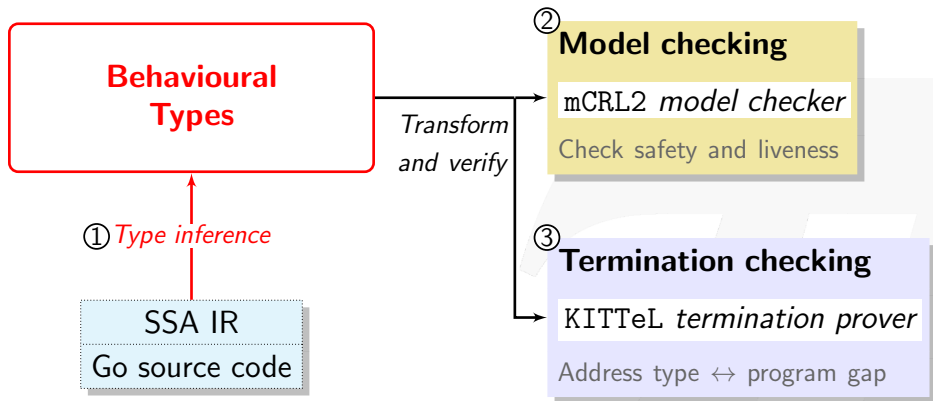
Our goal

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

→

$$\left\{ \begin{array}{l} \text{main()} = (\text{new } ch); \\ \quad (\text{send}\langle ch \rangle \mid \\ \quad \quad ch; \\ \quad \quad \text{close } ch), \\ \text{send}(ch) = \overline{ch} \end{array} \right\}$$

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

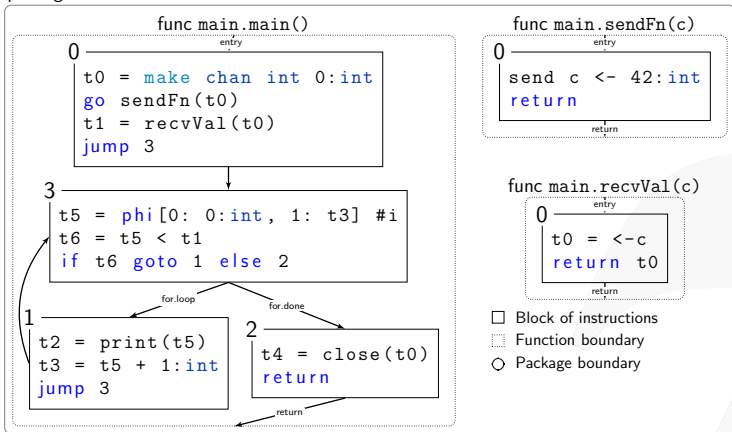
create channel → $\text{main}() = (\text{new } ch);$
 spawn → $(\text{send}\langle ch \rangle \mid$
 receive → $ch;$
 close → $\text{close } ch),$
 $\text{send}(ch) = \overline{ch}$
 send →

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

package main



Analyse in

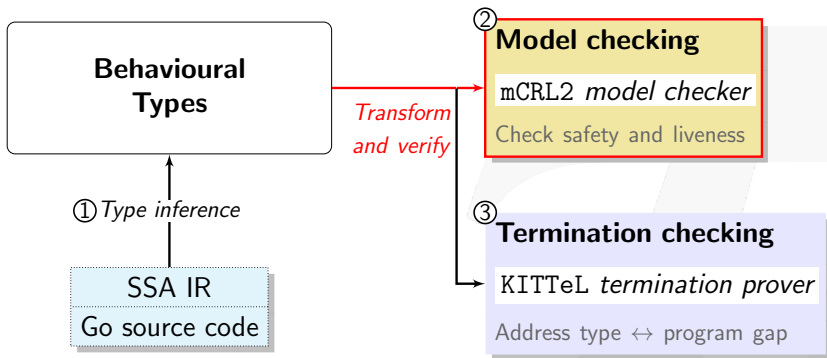
Static **S**ingle
Assignment

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**



Model checking behavioural types

$$M \models \phi$$

- **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.

$$\begin{array}{ccc}
 \bar{a}; T \xrightarrow{\bar{a}} T & \frac{T \xrightarrow{\bar{a}} T' \quad S \xrightarrow{a} S'}{T \mid S \xrightarrow{\tau_a} T' \mid S'} & a; T \xrightarrow{a} T \\
 \text{Send on channel } a & \text{Synchronise on } a & \text{Receive on channel } a
 \end{array}$$

Behavioural Types as LTS model

Standard CS semantics, i.e.

$$\begin{array}{ccc}
 \bar{a}; T \xrightarrow{\bar{a}} T & \frac{T \xrightarrow{\bar{a}} T' \quad S \xrightarrow{a} S'}{T \mid S \xrightarrow{\tau_a} T' \mid S'} & a; T \xrightarrow{a} T \\
 \text{Send on channel } a & \text{Synchronise on } a & \text{Receive on channel } a
 \end{array}$$

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

$$\begin{array}{ccc}
 \bar{a}; T \downarrow_{\bar{a}} & \frac{T \downarrow_{\bar{a}} \quad T' \downarrow_a}{T \mid T' \downarrow_{\tau_a}} & a; T \downarrow_a \\
 \text{Ready to send} & \text{Ready to synchronise} & \text{Ready to receive}
 \end{array}$$

Barbs ($T \downarrow_o$)

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

Specifying properties of model

$$\begin{array}{ccc}
 \bar{a}; T \downarrow_{\bar{a}} & \frac{T \downarrow_{\bar{a}} \quad T' \downarrow_a}{T \mid T' \downarrow_{\tau_a}} & a; T \downarrow_a \\
 \text{Ready to send} & \text{Ready to synchronise} & \text{Ready to receive}
 \end{array}$$

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 8
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 8
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 $P \xrightarrow{*} (\nu \bar{c}) Q$ and $Q \Downarrow \text{close}(a)$

$$\neg(Q \Downarrow \text{end}(a)) \wedge \neg(Q \Downarrow \bar{a})$$

never closing

never send

a closed

Migo Liveness / Safety

► Liveness

All reachable actions are eventually performed

P is live if $P \xrightarrow{*(vc)} Q$

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

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

Reduction
(τ)
at a

Select



Time
Out

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

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

$R_1 = a?$

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 not
live

$P_2 \mid R_2$ is

Select



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

Time
out

if P is live
 P_1 is live

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

$R_1 = a?$

P_2 is not
live
 $P_2 | R_2$ is

Barb $\downarrow \tilde{a}$

$\frac{\pi_i \downarrow a_i}{\text{select } \{\pi_i. P_i\} \downarrow \tilde{a}}$

$\frac{P \downarrow \tilde{a} \quad Q \downarrow \bar{a}_i}{P | Q \downarrow [a_i]}$

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

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

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

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 \mathbb{A} = set of all labels
 \implies Ready **receive/send** = not end of program.

Property: Global deadlock freedom

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

```

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

Property: Channel safety

$$\left(\bigwedge_{a \in \mathcal{A}} \downarrow a^* \right) \implies \neg(\downarrow \bar{a} \vee \downarrow \text{clo } a)$$

Once a channel a is closed (a^*),
it will not be **sent to**, nor closed again (**clo** a)

Property: Channel safety

$$\left(\bigwedge_{a \in \mathcal{A}} \downarrow a^* \right) \implies \neg(\downarrow \bar{a} \vee \downarrow \text{close } a)$$

```

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 }

```

- $\downarrow \text{close } ch$ when `close(ch)`
- $\downarrow ch^*$ fires after closed
- Send ($\downarrow \bar{ch}$: line 4)

Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

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

If a channel is ready to **receive** or **send**,

then **eventually**

it can synchronise (τ_a)

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

Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

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

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)

Liveness for Select

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

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

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

$$P_1 = \text{select}\{\bar{a}, b, \tau.P\}$$

$$P_2 = \text{select}\{\bar{a}, b\}$$

$$R_1 = a$$

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

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

$$P_1 = \text{select}\{\bar{a}, b, \tau.P\}$$

$$P_2 = \text{select}\{\bar{a}, b\}$$

$$R_1 = a$$

P_1 is live if P is ✓

P_2 is not live ✗

$(P_2 \mid R_1)$ is live ✓

Property: Liveness (partial deadlock freedom)

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

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

```

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)

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

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

```

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

Property: Eventual reception

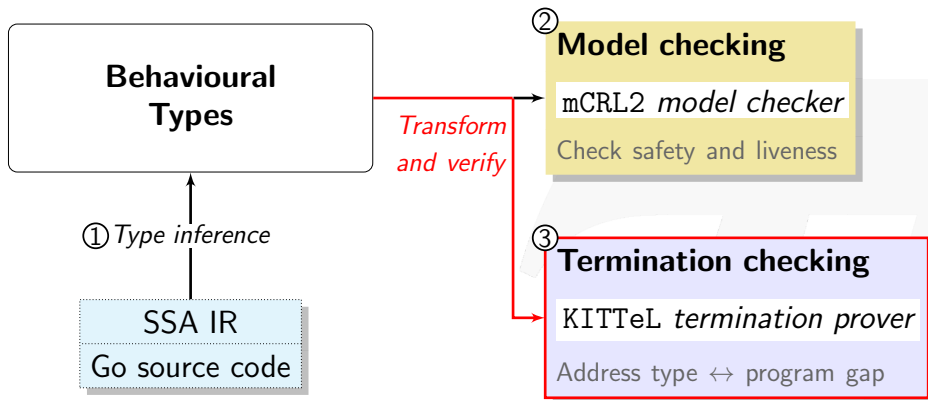
$$\left(\bigwedge_{a \in \mathcal{A}} \downarrow_{a^\bullet} \right) \implies \text{eventually} (\langle \tau_a \rangle \text{true})$$

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*



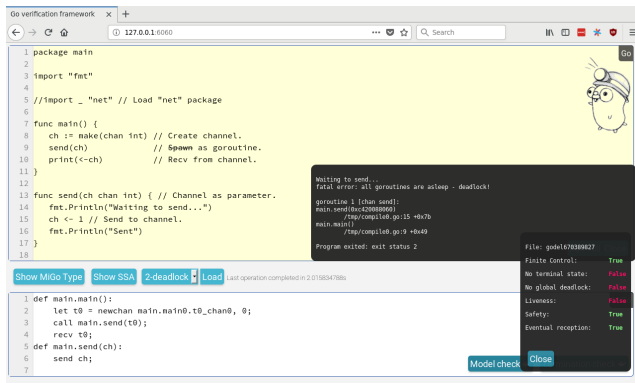
Termination checking with KITTeL

Type inference does not consider *program data*

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



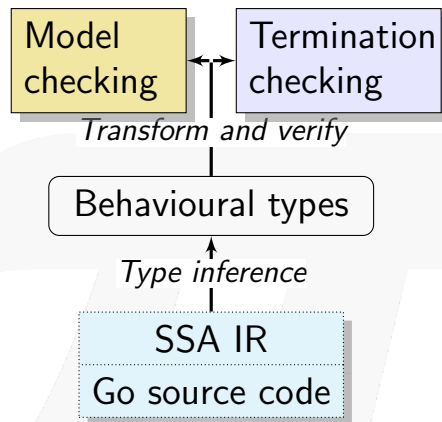
 Understanding Concurrency with Behavioural Types

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	Godel Checker						dingo-hunter [36]		gopherlyzer [40]		GolInfer/Gong [30]				
			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 stuckmsg	8	4	✓	✓	✓	×	3.5	996.6	996.6	✓	n/a	-	n/a	-	✓	✓	7.6

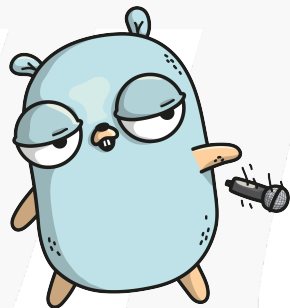
Future and related work

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)



Distributed Programming using Role-Parametric Session Types in Go

David Castro, Raymond Hu, Sung-Shik Jongmans,
Nicholas Ng, Nobuko Yoshida

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.

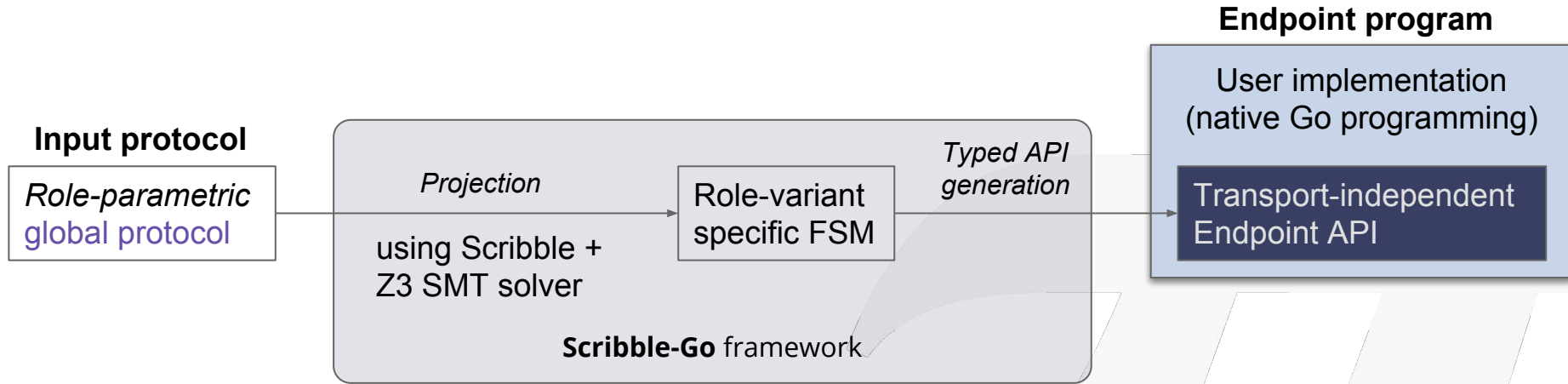
Implement

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

$\{M, F[1..n], \text{Server}\}$

If $F[1]$ sends an extra request

HTTP HEAD to Server to get total size

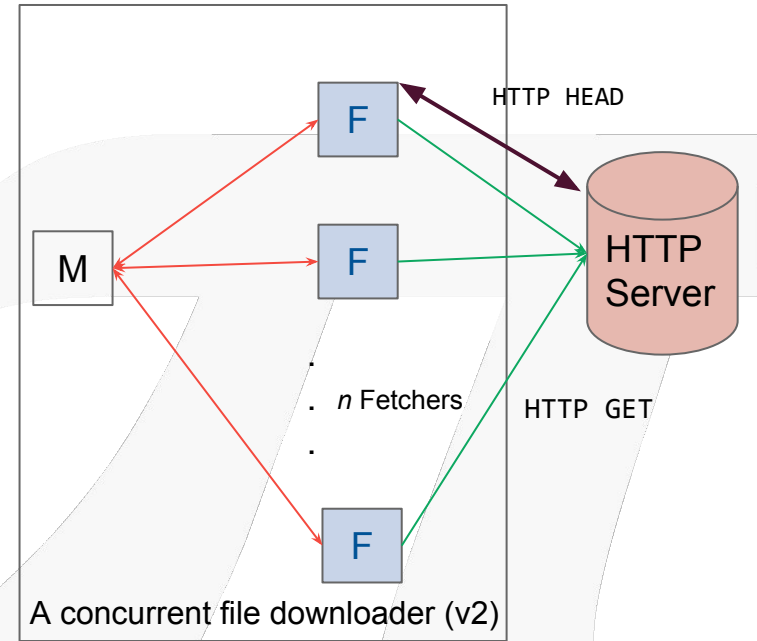
Then acts as a normal F

The role variants are:

$\{M, F[1], F[2..n], \text{Server}\}$

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

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



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

```
func doM2(m2 *M_2) M_End {  
    if m3 := m2.  
        Err: error  
        F_1_to_k: t1  
  
    func doM2(m2 *M_2) M_End {  
        if m3 := m2.F_1_to_k.  
            Scatter: t2  
  
        func doM2(m2 *M_2) M_End {  
            if m3 := m2.F_1_to_k.Scatter.  
                Job(a []Job) *M_3  
  
                :  
  
        func doM2(m2 *M_2, meta *Meta) M_End {  
            if m3 := m2.F_1_to_k.Scatter.Job(split(meta)); m3.Err != nil {  
            } else {  
                return m3.F_1_to_k.Gather.  
            }  
        }  
        Data(a []Data) M_End
```



Behavioural Types for Go

Type syntax

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

- 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

$$\boxed{\text{SND}} \quad \bar{a}; T \xrightarrow{\bar{a}} T \quad \boxed{\text{RCV}} \quad a; T \xrightarrow{a} T \quad \boxed{\text{TAU}} \quad \tau; T \xrightarrow{\tau} T$$

$$\boxed{\text{END}} \quad \text{close } a; T \xrightarrow{\text{clo } a} T \quad \boxed{\text{BUF}} \quad [a]_k^n \xrightarrow{\overline{\text{clo } a}} \text{buf}[a]_{\text{closed}} \quad \boxed{\text{CLD}} \quad \text{buf}[a]_{\text{closed}} \xrightarrow{a^*} \text{buf}[a]_{\text{closed}}$$

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

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

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

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

$$\boxed{\text{CLOSE}} \quad \frac{T \xrightarrow{\text{clo } a} T' \quad S \xrightarrow{\overline{\text{clo } a}} S'}{T \mid S \xrightarrow{\tau} T' \mid S'} \quad \boxed{\text{IN}} \quad \frac{k < n}{[a]_k^n \xrightarrow{a^\bullet} [a]_{k+1}^n} \quad \boxed{\text{OUT}} \quad \frac{k \geq 1}{[a]_k^n \xrightarrow{a^\bullet} [a]_{k-1}^n}$$

Barb predicates for types

$$\begin{array}{c}
 a; T \downarrow_a \quad \text{close } a; T \downarrow_{\text{clo } a} \quad \frac{\forall i \in \{1, \dots, n\} : \alpha_i \downarrow_{o_i}}{\&\{\alpha_i; T\}_{i \in \{1, \dots, n\}} \downarrow_{\{o_1 \dots o_n\}}} \\
 \bar{a}; T \downarrow_{\bar{a}} \quad \text{buf}[a]_{\text{closed}} \downarrow_{a^*} \\
 \\
 \frac{T \downarrow_o}{T; T' \downarrow_o} \quad \frac{T \downarrow_a \quad T' \downarrow_{\bar{a}} \text{ or } T' \downarrow_{a^*}}{T \mid T' \downarrow_{\tau_a}} \quad \frac{T \{\bar{a}/\bar{x}\} \downarrow_o \quad \mathbf{t}(\bar{x}) = T}{\mathbf{t}(\bar{a}) \downarrow_o} \\
 \\
 \frac{T \downarrow_a \quad \alpha_i \downarrow_{\bar{a}}}{T \mid \&\{\alpha_i; S_i\}_{i \in I} \downarrow_{\tau_a}} \quad \frac{T \downarrow_{\bar{a}} \text{ or } T \downarrow_{a^*} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i; S_i\}_{i \in I} \downarrow_{\tau_a}} \\
 \\
 \frac{k < n}{[a]_k^n \downarrow_{\bullet a}} \quad \frac{k \geq 1}{[a]_k^n \downarrow_{a^\bullet}} \quad \frac{T \downarrow_{\bar{a}} \quad T' \downarrow_{\bullet a}}{T \mid T' \downarrow_{\tau_a}} \quad \frac{T \downarrow_{a^\bullet} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i; S_i\}_{i \in I} \downarrow_{\tau_a}} \\
 \\
 \frac{T \downarrow_o}{T \mid T' \downarrow_o} \quad \frac{T \downarrow_o \quad a \notin \text{fn}(o)}{(\text{new}^n a); T \downarrow_o} \quad \frac{T \downarrow_o \quad T \equiv T'}{T \downarrow_o}
 \end{array}$$

Figure: Barb predicates for types.