Fencing Off GO

Liveness and Safety for Channel-based Concurrency

Julien Langé

Nicholas Ng

Bernardo Toninho

Nobuko Yoshida

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

Motivation

Deadlock Detection in Go

Fibonacci in Go

```
func fib(n int, ch chan int) {
    if (n <= 1) {
      ch <- n
    } else {
      newch := make(chan int)
      go fib(n-1,newch)
      go fib(n-2,newch)
      x := <-newch
      y := <-newch
      ch <- x+y
    }}
```

```
func main() {
   c := make(chan int)
   go fib(10,c)
   fmt.Println(<-c)
   close c
}
```

Motivation

Deadlock Detection in Go

Fibonacci in Go

```
func fib(n int, ch chan int) {
    if (n <= 1) {
        ch <- n
    } else {
        newch := make(chan int)
        go fib(n-1,newch)
        go fib(n-2,newch)
        x := <-newch
        y := <-newch
        ch <- x+y
    }}
```

```
func main() {
    c := make(chan int)
    go fib(10,c)
    fmt.Println(<-c)
    close c
}
```

Send n to ch

new channel

go rounine

receive x at newch

close channel

Fibonacci in Go

```
func fib(n int, ch chan int) {
    if (n <= 1) {
      ch <- n
    } else {
      newch := make(chan int)
      go fib(n-1,newch)
      go fib(n-2,newch)
      x := <-newch
      y := <-newch
      ch <- x+y
    }}
```

ch!⟨n⟩

Fib⟨n-1, newch⟩

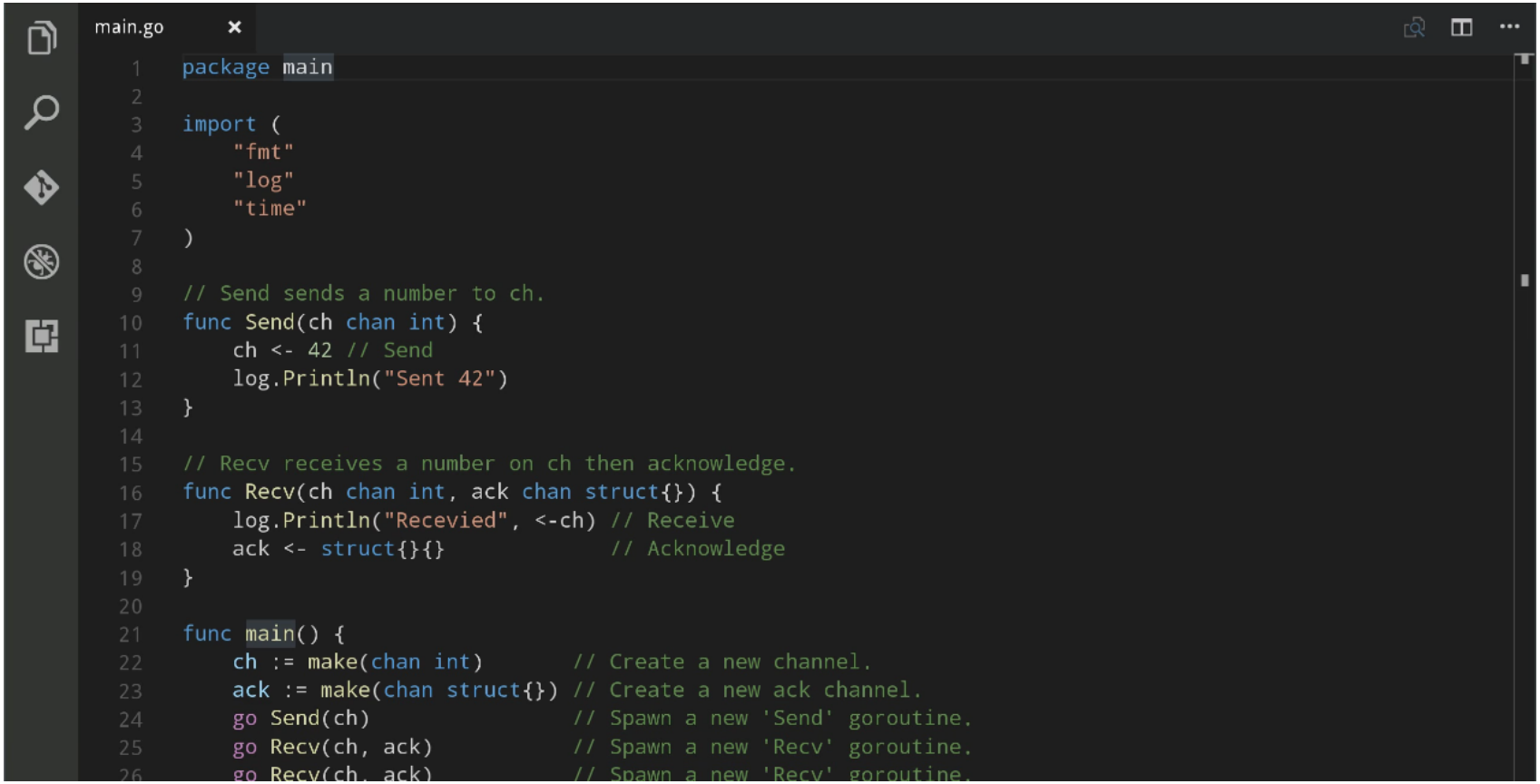
newch?(x)

```
func main() {
    c := make(chan int)
    go fib(10,c)
    fmt.Println(<-c)
    close c
```

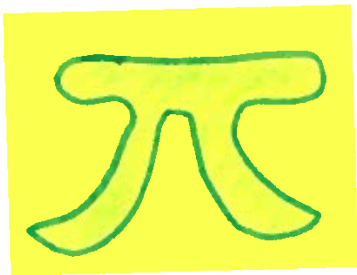
- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi
  e.g. [ACM Survey, 2016] 185 citations, 6 pages

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

Go runtime deadlock detector

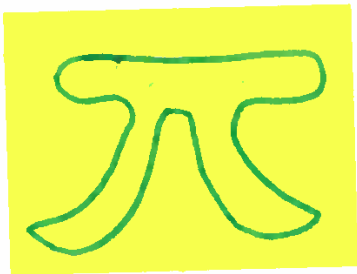


- Go has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi
  e.g. [ACM Survey, 2016] 185 citations, 6 pages



- 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?
- Use behavioural types in process calculi



e.g. [ACM Survey, 2016] 185 citations, 6 pages

186 ??

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

Understandable

Approach

STEP 1 MiGo (value-passing CCS)

- (Most) Message passing features of GO
- Tricky primitives : selection, channel closing
- Liveness / Safety

STEP 2 MiGo TYPES (CCS)

- Restrict to **Fenced** types and define symbolic semantics
- Check Liveness / Safety

STEP 3

- Relate Liveness / Safety in Migo with Liveness / Safety in Migo Types

Implement verification tools

Approach

STEP 1 MiGo (value-passing CCS)

- (Most) Message passing features of GO
- Tricky primitives : selection, channel closing
- Liveness / Safety

STEP 2 MiGo TYPES (CCS)

- Restrict to **Fenced** types and define symbolic semantics
- Check Liveness / Safety

STEP 3

- Relate Liveness / Safety in Migo with Liveness / Safety in Migo Types

Implement verification tools

Fibonacci in MiGo

Definition

$$Fib(n, c) \triangleq \text{if } (n \leq 1) \text{ then } c!\langle n\rangle \text{ else}$$

$$\text{newchan}(c'\text{:int}); (Fib\langle n-1, c'\rangle \mid Fib\langle n-2, c'\rangle \mid c'?(x); c'?(y); c!\langle x+y\rangle)$$

Program

$$\{Fib(n, c)\} \text{ in } \quad \text{newchan}(c\text{:int}); (Fib\langle 10, c\rangle \mid c?(u); \mathbf{0})$$

Prime Sieve

$$\begin{aligned} Gen(n, c) &\triangleq c!\langle n\rangle; Gen\langle n+1, c\rangle \\ F(n, i, o) &\triangleq i?(x); \text{if } (x\%n \neq 0) \text{ then } o!\langle x\rangle; F\langle n, i, o\rangle \text{ else } F\langle n, i, o\rangle \\ Rec(c) &\triangleq c?(x); \text{newchan}(c'\text{:int}); (F\langle x, c, c'\rangle \mid Rec\langle c'\rangle) \end{aligned}$$

$$\{Gen(n, c), F(n, i, o), Rec(c)\} \text{ in } \quad \text{newchan}(c\text{:int}); (Gen(2, c) \mid Rec(c))$$

MiGo Liveness / Safety

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 $P \xrightarrow{*} (\nu \tilde{c}) Q$ and $Q \downarrow close(a)$

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

a closed

never closing

never send

Migo Liveness / Safety

- **Liveness**

  All reachable actions are eventually performed

P is live if $P \xrightarrow{*} (\nu \vec{c})Q$

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

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

Reduction (tau) at a

TYPES

Fibonacci Types

$$\begin{aligned} \textbf{\textit{fib}}(x) &\triangleq \overline{x} \oplus (\text{new}\, b)(\textbf{\textit{fib}}\langle b\rangle \mid b; b; \overline{x} \mid \textbf{\textit{fib}}\langle b\rangle) \\ \textbf{\textit{t}}_0() &\triangleq (\text{new}\, a)(\textbf{\textit{fib}}\langle a\rangle \mid a) \end{aligned}$$

Prime Sieve Types

$$\begin{aligned} \textbf{\textit{gen}}(x) &\triangleq \overline{x}; \textbf{\textit{gen}}\langle x\rangle \\ \textbf{\textit{filter}}(x,y) &\triangleq x; (\overline{y}; \textbf{\textit{filter}}\langle x,y\rangle \oplus \textbf{\textit{filter}}\langle x,y\rangle) \\ \textbf{\textit{rec}}(x) &\triangleq x; (\text{new}\, b)(\textbf{\textit{filter}}\langle x,b\rangle \mid \textbf{\textit{rec}}\langle b\rangle) \\ \textbf{\textit{t}}_0() &\triangleq (\text{new}\, a)(\textbf{\textit{gen}}\langle a\rangle \mid \textbf{\textit{rec}}\langle a\rangle) \end{aligned}$$

Subject Reduction Theorem

$$\Gamma \vdash P \triangleright T \wedge P \to P' \Rightarrow T \to T' \wedge \Gamma \vdash P' \triangleright T'$$

TYPES

Fibonacci Types

$$\begin{aligned} \mathbf{fib}(x) &\triangleq \overline{x} \oplus (\text{new } b)(\mathbf{fib}\langle b\rangle \mid b; b; \overline{x} \mid \mathbf{fib}\langle b\rangle) \\ \mathbf{t_0}() &\triangleq (\text{new } a)(\mathbf{fib}\langle a\rangle \mid a) \end{aligned}$$

Prime Sieve Types

$$\begin{aligned} \mathbf{gen}(x) &\triangleq \overline{x}; \mathbf{gen}\langle x\rangle \\ \mathbf{filter}(x,y) &\triangleq x; (\overline{y}; \mathbf{filter}\langle x,y\rangle \oplus \mathbf{filter}\langle x,y\rangle) \\ \mathbf{rec}(x) &\triangleq x; (\text{new } b)(\mathbf{filter}\langle x,b\rangle \mid \mathbf{rec}\langle b\rangle) \\ \mathbf{t_0}() &\triangleq (\text{new } a)(\mathbf{gen}\langle a\rangle \mid \mathbf{rec}\langle a\rangle) \end{aligned}$$

Subject Reduction Theorem

$$\Gamma \vdash P \triangleright T \wedge P \to P' \Rightarrow T \to T' \wedge \Gamma \vdash P' \triangleright T'$$

Typing MiGo Types T

$G ; \tilde{x} ; \tilde{y} \vdash_t T$ judgement for **FENCE**

single thread recursion

multi thread for each recursion, at least one of parameters must be forgotten

Ensure Finite Control of Labelled Transition $\xrightarrow{\ell}$ over **Fenced Types**

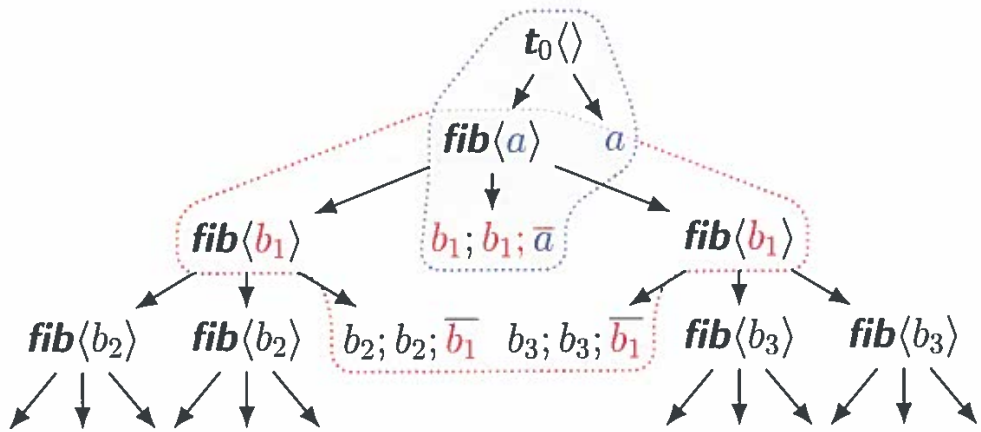
$T \xrightarrow[k]{\ell} T'$ LTS

created new names are less than k

Fibonacci Types – Fenced

$$\begin{aligned} \textbf{\textit{fib}}(x) &\triangleq \overline{x} \oplus (\text{new}\, b)(\textbf{\textit{fib}}\langle b\rangle \mid b; b; \overline{x} \mid \textbf{\textit{fib}}\langle b\rangle) \\ \textbf{\textit{t}}_0() &\triangleq (\text{new}\, a)(\textbf{\textit{fib}}\langle a\rangle \mid a) \end{aligned}$$

Observation: Recursive calls to $\textbf{\textit{fib}}\langle b\rangle$ cannot access parameter x.

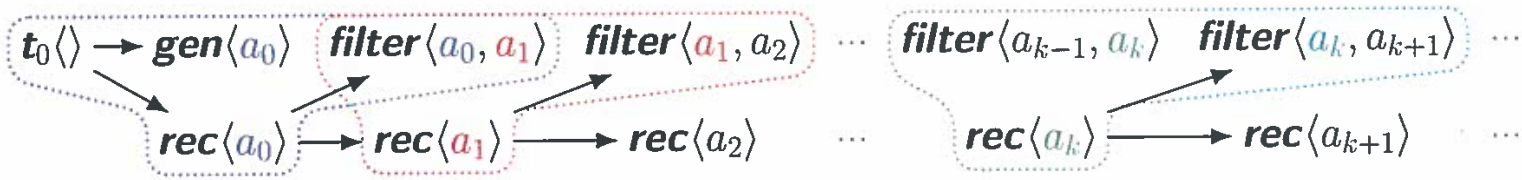


Analysing Behavioural Types

Fencing

Prime Sieve Types – Fenced

$$
\begin{aligned}
\textbf{\textit{gen}}(x) &\triangleq \overline{x}; \textbf{\textit{gen}}\langle x\rangle \\
\textbf{\textit{filter}}(x,y) &\triangleq x; (\overline{y}; \textbf{\textit{filter}}\langle x,y\rangle \oplus \textbf{\textit{filter}}\langle x,y\rangle) \\
\textbf{\textit{rec}}(x) &\triangleq x; (\mathsf{new}\, b)(\textbf{\textit{filter}}\langle x,b\rangle \mid \textbf{\textit{rec}}\langle b\rangle) \\
\textbf{\textit{t}}_0() &\triangleq (\mathsf{new}\, a)(\textbf{\textit{gen}}\langle a\rangle \mid \textbf{\textit{rec}}\langle a\rangle)
\end{aligned}
$$

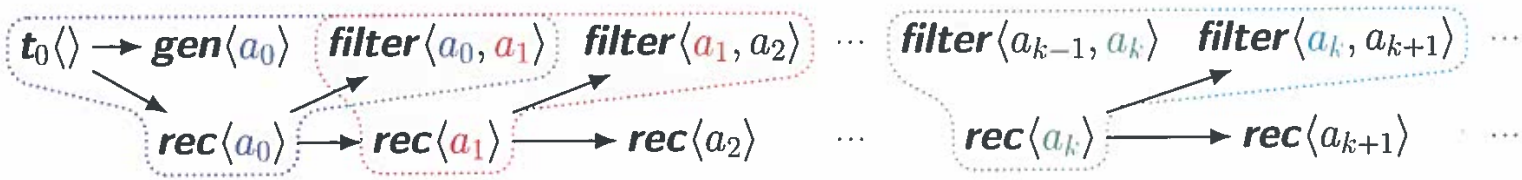


Analysing Behavioural Types

Fencing

Prime Sieve Types – Fenced

$$
\begin{aligned}
\textbf{\textit{gen}}(x) &\triangleq \overline{x}; \textbf{\textit{gen}}\langle x\rangle \\
\textbf{\textit{filter}}(x,y) &\triangleq x; (\overline{y}; \textbf{\textit{filter}}\langle x,y\rangle \oplus \textbf{\textit{filter}}\langle x,y\rangle) \\
\textbf{\textit{rec}}(x) &\triangleq x; (\mathsf{new}\, b)(\textbf{\textit{filter}}\langle x,b\rangle \mid \textbf{\textit{rec}}\langle b\rangle) \\
\textbf{\textit{t}}_0() &\triangleq (\mathsf{new}\, a)(\textbf{\textit{gen}}\langle a\rangle \mid \textbf{\textit{rec}}\langle a\rangle)
\end{aligned}
$$



- Finitely Many "Kinds" of Multi threads
- All names are shared by Finitely Many threads

NO FENCE

✗ $Rec(x,y) = (T \mid \cdot\cdot\ Rec(x,y))$

✗ $Rec(x,y,z) = (T \mid \cdot\cdot\ (new\ a) \cdot\cdot\ Rec(x,y,a))$

✓ $Rec(x,y,z) = (T \mid (new\ a) \cdot\cdot\ Rec(y,z,a))$

↑ forgotten

← shift

FENCED TYPES

Assume types are fenced then

Lemma (Finite Control)

$\{ [T]_{\equiv} \mid to\langle\rangle \xrightarrow{*}_k T \}$ finite for finite k

k-Safety · k-Liveness

It is decidable whether **T** is k-safe and k-live

FENCED TYPES

Assume types are fenced then

Lemma (Finite Control)

$\{ [T]_{\equiv} \mid to\langle\rangle \xrightarrow{*}_k T \}$ finite for finite k

k-Safety · k-Liveness

It is decidable whether T is k-safe and k-live

Safety · Liveness

There exists k s.t. if T is k-safe (k-live) then T is safe (live)

FENCED TYPES

Assume types are fenced then

Lemma (Finite Control)

$\{ [T]_{\equiv} \mid to\langle\rangle \xrightarrow{*}_k T \}$ finite for finite k

K-Safety · K-Liveness

It is decidable whether T is k-safe and k-live

Safety · Liveness

There exists k

then T is safe

NOT TRUE...

Thanks!

Naoki Kobayashi

Relating Programs and Types

Program

$$F(n,i,o) \triangleq i?(x); \text{if } (x\%n \neq 0) \text{ then } o!\langle x\rangle; F\langle n,i,o\rangle \text{ else } F\langle n,i,o\rangle$$

Type

$$\textbf{filter}(i,o) \triangleq i;(\overline{o};\mathbf{t}_F\langle i,o\rangle \oplus \mathbf{t}_F\langle i,o\rangle)$$

- Identify 3 classes (Liveness)
  1. May Terminate
  2. Without infinitely running conditionals
  3. Non-deterministic conditional
- Channel Safety Programs = Types

Relating Programs and Types

Program

$$F(n,i,o) \triangleq i?(x); \text{if } (x\%n \quad \text{o!}\langle x\rangle; F\langle n,i,o\rangle \text{ else } F\langle n,i,o\rangle$$

Type

$$\textbf{filter}(i,o) \triangleq i;(\overline{o}; \mathbf{t}_F\langle i,o\rangle \oplus \mathbf{t}_F\langle i,o\rangle)$$

- Identify 3 classes (Liveness)

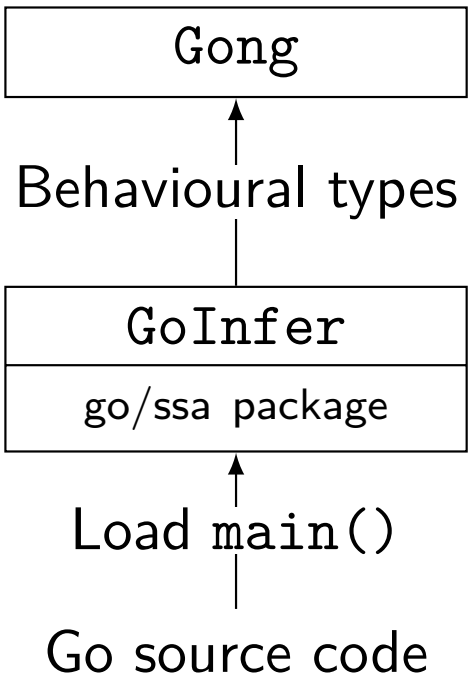
  1. May Terminate
  2. Without infinitely running conditionals
  3. Non-deterministic conditional

Undecidable but use Termination, Fair check...

- Channel Safety Programs = Types

Implementation

Toolchain



`Gong` *written in* ***Haskell***

The tool checks input behavioural types for liveness and channel safety.

`GoInfer` *written in* ***Go***

The tool loads source code, type-checks and builds SSA IR using `go/ssa` package, then extracts communication from the SSA IR as behavioural types.

github.com/nickng/gong

Conclusion

- MiGo — a formal model for Go — Liveness / Safety
- MiGo TYES — fenced types — k-liveness / k-safety
- Relate MiGo and MiGo Types' — liveness / safety

Asynchrony • Tools

Future Work

channel passing · locks

behaviour semantics • behavioural typing system

Model Checking ⟺ Fencing

Conclusion

- MiGo — a formal model for Go — Liveness / Safety
- MiGo TYES — fenced types — k-liveness / k-safety
- Relate MiGo and MiGo Types' — liveness / safety

Asynchrony • Tools

Future Work

channel passing · locks

[IK01] [K98, 05, 06] [S08] [YBH01] [HY94] [P13] [GKL14]

Session Types Contracts

behaviour semantics • behavioural typing system

Model Checking ⟺ Fencing

Conclusion

- MiGo a formal model for Go — Liveness / Safety
- MiGo TYES fenced types — k-liveness / k-safety
- Relate MiGo and MiGo Types' — liveness / safety

Asynchrony • Tools

Future Work

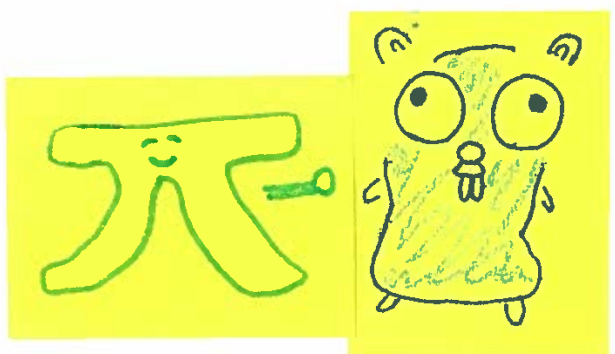
channel passing · locks

[IK01] [K98, 05, 06] [S08] [YBH01] [HY94] [P13] [GKL14]

Session Types Contracts

behaviour semantics • behavioural typing system

Model Checking ⇔ Fencing



Evaluation

Examples	LoC	# chans unbuf.	# chans buf.	Live	Safe	Analysis Time (ms)	[4] Static	[4] Safe
`sieve`	19	2	0	✓	✓	209.55	×	
`fib`	23	2	0	✓	✓	14638.4	×	
`fib-async`	23	1	1	✓	✓	32173.8	×	
`fact`	19	2	0	✓	✓	206.63	×	
`dinephil` [2]	56	3	0	✓	✓	646921.76	×	
`jobsched`	41	0	1	✓	✓	48.12	×	
`concsys` [1]	112	2	0	×	✓	323.75	×	
`fanin` [5, 4]	36	3	0	✓	✓	89.14	✓	✓
`fanin-alt` [4]	37	3	0	×[1]	✓	209.02	✓	✓
`mismatch` [4]	26	2	0	×	✓	26.59	✓	×
`fixed` [4]	25	2	0	✓	✓	24.58	✓	✓
`alt-bit` [3]	74	0	2	✓	✓	405.78	✓	✓
`forselect`	40	3	0	✓	✓	31.01	✓	✓
`cond-recur`	32	2	0	✓	✓	34.08	✓	✓

[1]: testing for channel close state is not supported in this version


Collection of Golang concurrency patterns.
`https://github.com/stillwater-sc/concurrency`.


E. Giachino, N. Kobayashi, and C. Laneve.
Deadlock analysis of unbounded process networks.
In *CONCUR*, volume 8704 of *LNCS*, pages 63–77. Springer, 2014.


R. Milner.
Communication and Concurrency.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.


N. Ng and N. Yoshida.
Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
In *CC 2016*, pages 174–184. ACM, 2016.


Sameer Ajmani.
Go Concurrency Patterns: Pipelines and cancellation, 2014.
`https://blog.golang.org/pipelines`.