Fencing Off Go 🐒:
Liveness and Safety for Channel-based Programming

Julien Lange, Nicholas Ng,
Bernardo Toninho, Nobuko Yoshida

Imperial College London, UK

POPL, January 20, 2017
Introduction
Formal Models and Behavioural Typing

Channel-based Concurrency – Theoretical World

- Modeled using process algebras (CCS, $\pi$-calculus, etc.)
- Disciplined by sophisticated typing systems:
  - Linear types, Session types, Usage types, etc.
  - Comm. error freedom, Deadlock- and livelock-freedom, etc.
**Introduction**

**Formal Models and Behavioural Typing**

**Channel-based Concurrency – Theoretical World**

- Modeled using process algebras (CCS, $\pi$-calculus, etc.)
- Disciplined by sophisticated typing systems:
  - Linear types, Session types, Usage types, etc.
  - Comm. error freedom, Deadlock- and livelock-freedom, etc.

**Channel-based Concurrency – Programming World**

- Socket-based programming
- Actor-based programming
- Buffers-as-channels programming
- Little to no type-level support for communication!
Introduction
Go Programming Language

The Go Programming Language

- Systems PL for multicore programming, successor of C
- Developed by Google – Pike, Thompson, Griesemer (2009)
- Users outside Google: Dropbox, Netflix, Docker, CoreOS
Introduction
Go Programming Language

The Go Programming Language

► Systems PL for multicore programming, successor of C
► Developed by Google – Pike, Thompson, Griesemer (2009)
► Users outside Google: Dropbox, Netflix, Docker, CoreOS
► Message-passing concurrency primitives:
  ► Buffered I/O communication channels.
  ► Lightweight thread spawning (goroutines).
  ► Selective send/receive construct.
► Encourages concurrency via message-passing over shared-memory.
Motivation
Deadlock Detection in Go

Fibonacci in Go

```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
    }
}
```

Program eventually prints 55.
Motivation
Deadlock Detection in Go

Fibonacci in Go

```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)
}
```

Program eventually prints 55.
Fibonacci in Go

```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
    }
}
```

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

Program eventually prints 55.
Fibonacci in Go

```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 := <-ch // was newch
        ch <- x+y
    }
}
```

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

Program is deadlock.
Motivation
Deadlock Detection in Go

- Go has a **runtime** global deadlock detector.
- Many deadlocks/errors of real programs are not captured.
- Can we detect deadlocks and comm. errors **statically**?
Motivation
Deadlock Detection in Go

- Go has a **runtime** global deadlock detector.
- Many deadlocks/errors of real programs are not captured.
- Can we detect deadlocks and comm. errors **statically**?

Goals

- **Static** liveness and safety analysis tool for Go.
- Applicable to realistic programs.
Motivation
Deadlock Detection in Go

- Go has a runtime global deadlock detector.
- Many deadlocks/errors of real programs are not captured.
- Can we detect deadlocks and comm. errors statically?

Goals

- Static liveness and safety analysis tool for Go.
- Applicable to realistic programs.
- Leverage research on deadlock analysis from process calculi.
- Bridge gap between theoretical models and language.
Our Approach

- Develop a formal model of (the core) of Go – MiGo:
  - (Most) message passing features of Go.
  - Recursive procedures over imperative features.
  - No shared memory concurrency features (yet!).
  - MiGo programs can be extracted from Go code.
Our Approach

- Develop a formal model of (the core) of Go – MiGo:
  - (Most) message passing features of Go.
  - Recursive procedures over imperative features.
  - No shared memory concurrency features (yet!).
  - MiGo programs can be extracted from Go code.

- A behavioural typing system for MiGo
  - Types are CCS-like processes.
  - Abstract MiGo behaviours.
Our Approach

- Develop a formal model of (the core) of Go – MiGo:
  - (Most) message passing features of Go.
  - Recursive procedures over imperative features.
  - No shared memory concurrency features (yet!).
  - MiGo programs can be extracted from Go code.

- A behavioural typing system for MiGo
  - Types are CCS-like processes.
  - Abstract MiGo behaviours.

- Verification framework for MiGo types (Tool):
  - A simple syntactic check on types – *fencing*.
  - Check bounded comm. safety and liveness.
  - Relate type properties with MiGo properties.
Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} 
  c!\langle n \rangle & \text{if } (n \leq 1) \\
  \text{newchan}(c':\text{int}); \\
  (Fib\langle n-1, c' \rangle \mid Fib\langle n-2, c' \rangle \mid c'? (x); c'? (y); c!\langle x+y \rangle) & \text{else}
\end{cases} \]
Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} \text{if } (n \leq 1) \text{ then } c!\langle n \rangle \text{ else newchan}(c':\text{int}); \\ (Fib\langle n-1, c' \rangle \mid Fib\langle n-2, c' \rangle \mid c'?(x); c'? (y); c!\langle x+y \rangle) \end{cases} \]
A Model of Go – MiGo
By Example

Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} c!\langle n \rangle & \text{if } (n \leq 1) \\ \text{newchan}(c':\text{int}); \\
(Fib\langle n-1, c' \rangle \mid Fib\langle n-2, c' \rangle \mid c'? (x); c'? (y); c!\langle x+y \rangle) & \text{else} \end{cases} \]
A Model of Go – MiGo

By Example

Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} 
  c!\langle n \rangle & \text{if } (n \leq 1) \\
  \text{newchan}(c':\text{int});
  (Fib\langle n-1, c' \rangle | Fib\langle n-2, c' \rangle | c'? (x); c'? (y); c!\langle x+y \rangle) & \text{else}
\end{cases} \]
A Model of Go – MiGo

By Example

Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} \text{if } (n \leq 1) \text{ then } c!(n) \text{ else newchan}(c':\text{int}); \\ (Fib(n-1, c') \mid Fib(n-2, c') \mid c'(x); c'(y); c!(x+y)) \end{cases} \]
A Model of Go – MiGo

By Example

Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} 
\text{if } (n \leq 1) \text{ then } c!\langle n \rangle 
\text{ else newchan}(c':\text{int}); \\
(Fib\langle n-1, c' \rangle \mid Fib\langle n-2, c' \rangle \mid c'?(x); c'?(y); c!\langle x+y \rangle) 
\end{cases} \]
A Model of Go – MiGo

By Example

Fibonacci in MiGo

\[ \text{Fib}(n, c) \triangleq \begin{cases} \text{if } (n \leq 1) \text{ then } c!\langle n \rangle \text{ else } \text{newchan}(c':\text{int}); \\ \text{\{} \text{Fib}(n-1, c') \mid \text{Fib}(n-2, c') \mid c'?x; c'?y; c!\langle x+y \rangle \text{\}} \end{cases} \]

\{ \text{Fib}(n, c) \} \text{ in } \text{newchan}(c:\text{int}); (\text{Fib}(10, c) \mid c?u; 0)
A Model of Go – MiGo

By Example

Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} c!\langle n \rangle & \text{if } (n \leq 1) \\ \text{newchan}(c':\text{int}); \\
(Fib\langle n-1, c' \rangle | Fib\langle n-2, c' \rangle | c'?\langle x \rangle; c'?\langle y \rangle; c!\langle x+y \rangle) & \text{else} \end{cases} \]

\{ Fib(n, c) \} \text{ in newchan}(c:\text{int}); (Fib\langle 10, c \rangle | c?(u); 0)
A Model of Go – MiGo
By Example

Fibonacci in MiGo

\[ Fib(n, c) \triangleq \begin{cases} c!\langle n \rangle & \text{if } (n \leq 1) \\ \text{newchan}(c':\text{int}); \\ (Fib\langle n-1, c' \rangle \mid Fib\langle n-2, c' \rangle \mid c'?(x); c'?(y); c!\langle x+y \rangle) & \end{cases} \]

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

- A core process calculus (value-passing CCS with recursion).
- No name-passing.
- Still very expressive (i.e. most properties are undecidable).
A Model of Go – MiGo
By Example

Prime Sieve

\[ \text{Gen}(n, c) \triangleq c!\langle n \rangle; \text{Gen}\langle n+1, c \rangle \]
A Model of Go – MiGo

By Example

Prime Sieve

\[
\begin{align*}
Gen(n, c) & \triangleq c!\langle n \rangle; Gen\langle n+1, c \rangle \\
F(n, i, o) & \triangleq i?(x); \text{if } (x \% n \neq 0) \text{ then } o!\langle x \rangle; F\langle n, i, o \rangle \text{ else } F\langle n, i, o \rangle
\end{align*}
\]
A Model of Go – MiGo

By Example

Prime Sieve

\[
\begin{align*}
    \text{Gen}(n, c) & \triangleq c!\langle n \rangle; \text{Gen}\langle n+1, c \rangle \\
    F(n, i, o) & \triangleq i?(x); \text{if } (x \% n \neq 0) \text{ then } o!\langle x \rangle; F\langle n, i, o \rangle \text{ else } F\langle n, i, o \rangle \\
    \text{Rec}(c) & \triangleq c?(x); \text{newchan}(c':\text{int}); (F\langle x, c, c' \rangle \mid \text{Rec}\langle c' \rangle)
\end{align*}
\]
A Model of Go – MiGo
By Example

Prime Sieve

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

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

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

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

Constructs all prime numbers.
MiGo

Channel Safety and Liveness – Informally

- In Go (and MiGo), a channel may be closed at most **once**.
- Can only input from a closed channel (default value).
- Other actions raise an error (and crash the program).
MiGo

Channel Safety and Liveness – Informally

- In Go (and MiGo), a channel may be closed at most once.
- Can only input from a closed channel (default value).
- Other actions raise an error (and crash the program).

Channel Safety

A program $P$ is channel safe iff:

- Each channel is closed at most once.
- Once a channel is closed, no outputs to it are performed.
MiGo
Channel Safety and Liveness – Informally

▶ In Go (and MiGo), a channel may be closed at most once.
▶ Can only input from a closed channel (default value).
▶ Other actions raise an error (and crash the program).

Channel Safety
A program $P$ is channel safe iff:
▶ Each channel is closed at most once.
▶ Once a channel is closed, no outputs to it are performed.

Liveness
A program $P$ is live iff:
▶ All reachable communication actions are eventually performed.
▶ i.e. no “stuck” communication.
MiGo Types
Behavioural Typing for MiGo

▶ Approximate the behaviour of processes
▶ *Almost* the same as processes (only diff. is conditionals and lack of data)!

Fibonacci Types

\[
fib(x) \triangleq x \oplus (\text{new } b)(fib(b) | fib(b) | b; b; x)
\]

\[
t_0() \triangleq (\text{new } a)(fib(a) | a)
\]

Prime Sieve Types

\[
gen(x) \triangleq x; gen(x)
\]

\[
filter(x, y) \triangleq x; (y; filter(x, y) \oplus filter(x, y))
\]

\[
rec(x) \triangleq x; (\text{new } b)(filter(x, b) | rec(b))
\]

\[
t_0() \triangleq (\text{new } a)(gen(a) | rec(a))
\]
MiGo Types

Behavioural Typing for MiGo

▶ Approximate the behaviour of processes
▶ *Almost* the same as processes (only diff. is conditionals and lack of data)!

Fibonacci Types

\[
\begin{align*}
\text{fib}(x) &\triangleq \bar{x} \oplus (\text{new } b)(\text{fib}\langle b \rangle | \text{fib}\langle b \rangle | b; b; \bar{x}) \\
n_0() &\triangleq (\text{new } a)(\text{fib}\langle a \rangle | a)
\end{align*}
\]
MiGo Types
Behavioural Typing for MiGo

- Approximate the behaviour of processes
- *Almost* the same as processes (only diff. is conditionals and lack of data)!

**Fibonacci Types**

\[
\begin{align*}
\text{fib}(x) & \triangleq \overline{x} \oplus (\text{new } b)(\text{fib}\langle b \rangle \mid \text{fib}\langle b \rangle \mid b; b; \overline{x}) \\
t_0() & \triangleq (\text{new } a)(\text{fib}\langle a \rangle \mid a)
\end{align*}
\]

**Prime Sieve Types**

\[
\begin{align*}
\text{gen}(x) & \triangleq \overline{x}; \text{gen}\langle x \rangle \\
\text{filter}(x, y) & \triangleq x; (\overline{y}; \text{filter}\langle x, y \rangle \oplus \text{filter}\langle x, y \rangle) \\
\text{rec}(x) & \triangleq x; (\text{new } b)(\text{filter}\langle x, b \rangle \mid \text{rec}\langle b \rangle) \\
t_0() & \triangleq (\text{new } a)(\text{gen}\langle a \rangle \mid \text{rec}\langle a \rangle)
\end{align*}
\]
MiGo Types
Behavioural Typing for MiGo

▶ Approximate the behaviour of processes
▶ *Almost* the same as processes (only diff. is conditionals and lack of data)!

**Fibonacci Types**

\[
\begin{align*}
\text{fib}(x) & \triangleq x \oplus (\text{new } b)(\text{fib}<b> | \text{fib}<b> | b; b; \overline{x}) \\
\text{t}_0() & \triangleq (\text{new } a)(\text{fib}<a> | a)
\end{align*}
\]

**Prime Sieve Types**

\[
\begin{align*}
\text{gen}(x) & \triangleq x; \text{gen}<x> \\
\text{filter}(x, y) & \triangleq x; (\overline{y}; \text{filter}<x, y> \oplus \text{filter}<x, y>) \\
\text{rec}(x) & \triangleq x; (\text{new } b)(\text{filter}<x, b> | \text{rec}<b>) \\
\text{t}_0() & \triangleq (\text{new } a)(\text{gen}<a> | \text{rec}<a>)
\end{align*}
\]
Analysing Behavioural Types

Overview

- Types have a notion of channel safety and liveness.
- Undecidable in general...
Analysing Behavioural Types

Overview

▶ Types have a notion of channel safety and liveness.
▶ Undecidable in general. . .

Fencing

▶ A restriction on recursive types with parallel composition.
▶ Types that spawn infinite threads:
  ▶ Consist of finitely many comm. patterns, using finitely many channels.
▶ Goal: Finite control of symbolic LTS ($\rightarrow_k$) for fenced types.
Fibonacci Types – Fenced

\[
\begin{align*}
\text{fib}(x) & \triangleq \overline{x} \oplus (\text{new } b)(\text{fib}\langle b \rangle \mid b; b; \overline{x} \mid \text{fib}\langle b \rangle) \\
\text{t}_0() & \triangleq (\text{new } a)(\text{fib}\langle a \rangle \mid a)
\end{align*}
\]
Fibonacci Types – Fenced

\[ fib(x) \triangleq \overline{x} \oplus (\text{new } b)(fib\langle b \rangle | b; b; \overline{x} | fib\langle b \rangle) \]
\[ t_0() \triangleq (\text{new } a)(fib\langle a \rangle | a) \]

Observation: Recursive calls to \( fib\langle b \rangle \) cannot access parameter \( x \).
Analysing Behavioural Types

Fencing

Fibonacci Types – Fenced

\[
\begin{align*}
\text{fib}(x) & \triangleq \overline{x} \oplus (\text{new } b)(\text{fib}(b) \mid b; b; \overline{x} \mid \text{fib}(b)) \\
t_0() & \triangleq (\text{new } a)(\text{fib}(a) \mid a)
\end{align*}
\]

Observation: Recursive calls to \( \text{fib}(b) \) cannot access parameter \( x \).
Fibonacci Types – Fenced

\[ \text{fib}(x) \triangleq \overline{x} \oplus (\text{new } b)(\text{fib}(b) \mid b; b; \overline{x} \mid \text{fib}(b)) \]

\[ \text{t}_0() \triangleq (\text{new } a)(\text{fib}(a) \mid a) \]

Observation: Recursive calls to fib(b) cannot access parameter x.
Analysing Behavioural Types

Fencing

Fibonacci Types – Fenced

\[
\begin{align*}
\text{fib}(x) & \triangleq \overline{x} \oplus (\text{new } b)(\text{fib}(b) \mid b; b; \overline{x} \mid \text{fib}(b)) \\
\text{t}_0() & \triangleq (\text{new } a)(\text{fib}(a) \mid a)
\end{align*}
\]

Observation: Recursive calls to \text{fib}(b) cannot access parameter \( x \).
**Analysing Behavioural Types**

**Fencing**

**Fibonacci Types – Fenced**

\[
\text{fib}(x) \triangleq \overline{x} \oplus (\text{new } b)(\text{fib}(b) \mid b; b; \overline{x} \mid \text{fib}(b))
\]

\[
t_0() \triangleq (\text{new } a)(\text{fib}(a) \mid a)
\]

Observation: Recursive calls to \(\text{fib}(b)\) cannot access parameter \(x\).
Fibonacci Types – Fenced

\[
\begin{align*}
\text{fib}(x) & \triangleq \overline{x} \oplus (\text{new } b)(\text{fib}(b) \mid b; b; \overline{x} \mid \text{fib}(b)) \\
t_0() & \triangleq (\text{new } a)(\text{fib}(a) \mid a)
\end{align*}
\]

Observation: Recursive calls to \( \text{fib}(b) \) cannot access parameter \( x \).
Analysing Behavioural Types

Fencing

- Fenced($T$) checks usage of recursive parameters:
  1. **Single-threaded** Recursion: “don’t care”; or,
  2. **Multi-threaded** Recursion: for each recursion, at least one of the parameters must be forgotten.
Analysing Behavioural Types

Fencing

▶ Fenced(\(T\)) checks usage of recursive parameters:
  1. **Single-threaded** Recursion: “don’t care”; or,
  2. **Multi-threaded** Recursion: for each recursion, *at least one* of the parameters must be forgotten.

▶ Generally, \(t(x, y, z) = (T \mid \ldots; t(x, y, z))\) is **not** fenced.

▶ Neither is \(t(x, y, z) = (T \mid (\text{new } a)\ldots; t(x, y, a))\).
Fenced($T$) checks usage of recursive parameters:

1. **Single-threaded** Recursion: “don’t care”; or,
2. **Multi-threaded** Recursion: for each recursion, *at least one* of the parameters must be forgotten.

Generally, $t(x, y, z) = (T | \ldots; t\langle x, y, z\rangle)$ is not fenced.

Neither is $t(x, y, z) = (T | (\text{new } a)\ldots; t\langle x, y, a\rangle)$.

But $t(x, y, z) = (T | (\text{new } a)\ldots; t\langle y, z, a\rangle)$ is fenced!
For all $T$ s.t. Fenced($T$):

**Finite Control**

$\{[T] \equiv | t_0 \rightarrow_k^* T\}$ finite, for finite $k$. 

**Symbolic Liveness and Safety**

It is decidable if $T$ is $k$-live and $k$-safe.

**Safety and Liveness**

We cannot compute $k$ s.t. $k$-liveness ($k$-safety) $\Rightarrow$ liveness ($k$-safety).
For all $T$ s.t. $\text{Fenced}(T)$:

**Finite Control**

$\{[T] \equiv | t_0 \rightarrow^*_k T\}$ finite, for finite $k$.

**Symbolic Liveness and Safety**

It is decidable if $T$ is $k$-live and $k$-safe.
For all $T$ s.t. $\text{Fenced}(T)$:

**Finite Control**

$\{[T] \equiv | t_0 \rightarrow^*_k T\}$ finite, for finite $k$.

**Symbolic Liveness and Safety**

It is decidable if $T$ is $k$-live and $k$-safe.

**Safety and Liveness**

We cannot compute $k$ s.t. $k$-liveness (safety) $\Rightarrow$ liveness (safety).
Prime Sieve – Excerpt

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

\[ \text{filter}(i, o) \triangleq i; (\overline{o}; t_F\langle i, o \rangle \oplus t_F\langle i, o \rangle) \]

- Liveness of type relies on ability to eventually take the “correct” branch.
- Conditionals in programs may disallow that action from actually happening!
Identify 3 classes of programs for which type liveness implies program liveness:

▶ Programs that “may” terminate.
Identify 3 classes of programs for which type liveness implies program liveness:

- Programs that "may" terminate.
- Programs without infinitely running conditionals (e.g. Fibonacci).
Identify 3 classes of programs for which type liveness implies program liveness:

- Programs that “may” terminate.
- Programs **without** infinitely running conditionals (e.g. Fibonacci).
- Programs with “**non-deterministic**” conditionals (e.g. Sieve).
Relating Programs and Types

Challenge

Identify 3 classes of programs for which type liveness implies program liveness:

▶ Programs that “may” terminate.
▶ Programs without infinitely running conditionals (e.g. Fibonacci).
▶ Programs with “non-deterministic” conditionals (e.g. Sieve).

In general undecidable, but can be approximated:

▶ Termination checkers, productivity checking, etc.
Summary

- A formal model of message-passing concurrency in Go.
- Behavioural typing and a bounded verification framework.
- Relate type behaviour with program behaviour.
- Not in the talk: Asynchrony, Tool.

Future Work

- Channel-passing.
- Locks / Shared-memory concurrency.
- Model-checking encoding.
Thank You!
Questions?