Compiling linear and static channels in Go

Marco Giunti

NOVA LINCS, Universidade NOVA de Lisboa

INForum, September 5 2019

Channels and programming languages

- Support for communication channels in programming languages is increasing (XC, **Go**, Crystal, Flix,...)
- `tour.golang.org`: sum of numbers in a slice by 2 goroutines

```
func sum(s []int, c chan int) {
    sum := 0
    for _, v := range s {
        sum += v
    }
    c ← sum //send sum to c
}

func calc() {
    s := generateRandomSlice(1000)
    c := make(chan int)
    go sum(s[:len(s)/2], c) //concurrent thread
    go sum(s[len(s)/2:], c) //concurrent thread
    x, y := ←c, ←c //receive from c
    fmt.Printf("The sum of the slice is %d", x + y)
}
```

Channel forwarding

- Function *sum* has full control on channel *c*, and can forward it to some public channel *pub* (cf. line 2)
- The sum of the slice can be intercepted and replaced with an arbitrary value (lines 13–14)

```
 1 func sum(s []int, c chan int) {
 2     go func(){pub ← c}() //c is forwarded
 3     sum := 0
 4     c ← sum
 5 }
 6 func calc() {
 7     s := generateRandomSlice(1000)
 8     c := make(chan int)
 9     go sum(s[:len(s)/2], c)
10 }
11 func inject() {
12     x := ←pub
13     _ =←x
14     x ← 0 //sum is set to 0
15 }
```

Designing protocols with no-forwarding

- Some apps as instant messengers already provide a no-forwarding feature to strengthen secrecy (e.g. Viber)
- In Go, we would need a **static make** that **disallows channel extrusion**

```
 1 func sum(s []int, c chan int) {
 2     go func() {pub ← c}()
 3     ...
 4 }
 5
 6 func calc() {
 7     s := generateRandomSlice(1000)
 8     c := static_make(chan int)
 9     go sum(s[:len(s)/2], c) //rejected
10 }
```

- Code would be rejected by compiler

```
sum.go:9:22: static channel may be extruded
```

Difficulties in tracking forwarding

- Because of channel-over-channel passing, detecting the extrusion of a static channel can be tricky

```
 1 //Protocol variant
 2 func sum(s []int, c chan int, p chan chan int) {
 3     go func(){pub ← p}() //forwarding p
 4     ...
 5 }
 6 func calc() {
 7     s := generateRandomSlice(1000)
 8     p := make(chan chan int)
 9     c := static_make(chan int)
10     go func(){ p ← c}() //passing c over p
11     go sum(s[:len(s)/2], c, p)
12 }
```

- Line 2: sum opens the scope of channel p
- Lines 10–11: c is sent over p: scope of c can be opened

Linearity and deadlock-avoidance

- Linearity or use channels exactly once *enhance* programs
- Benefits: resource-awareness, session-based protocols, predisposition towards deadlock-avoidance
- Analogously, in Go we would need a **linear make**

```
1 a,b := linear_make(chan string),linear_make(chan string)
2 go func(){
3       a ← "Hello"
4       b ← "world"
5 }()
6 _, _ =←b, ←a //Order of channels inverted
```

- Compilation would prevent deadlock (now catched at runtime)

```
hello.go:6:5: linear channel is deadlocked
```

An high level language with linear and static channels

- We study the problem of designing, analysing, and executing *message-passing protocols* featuring **channel-over-channel** passing, **linear** channels, and **static** channels
- We propose an high level language, named **LSpi**, that extends the pi calculus and offers support for all these features
- The language has few constructs, **no decorations**, and **fully-automatic compilation** in executable **Go** programs
- The compiler, named **GoPi**, is available through GitHub

Specification of the sum protocol in LSpi

- Channel c declared as static with **hide**, s_i is the slice, loop's result calculated by process listening on *for*, $|$ splits threads

$$P \overset{\text{def}}{=} [\text{hide}\, c][(\text{new}\, r_1)(Sum(s_1, c, r_1)) \mid (\text{new}\, r_2)(Sum(s_2, c, r_2)) \\ \mid c?(x).c?(y).\text{print} :: x + y]$$

$$Sum(slice, channel, result) \overset{\text{def}}{=} \\ for!\langle slice, result\rangle.result?(z).channel!z$$

- The **square brackets** indicate the static scope of the **hide** declaration, and *should not be enlarged* at runtime
- *Operational semantics*: input (**?**) and output (**!**) on same channel synchronize (noted: $\rightarrow$)

$$P \mid ForProc \rightarrow^{*} \text{print} :: n_1 + n_2 \mid ForProc$$

Disallowing channel forwarding

- Consider the unsafe version of Sum

$$P \overset{\text{def}}{=} (\text{new}\, p)(\, [\text{hide}\, c][\, p!c \mid (\text{new}\, r_1)(Sum(s_1, c, r_1, p)) \mid (\text{new}\, r_2)(Sum(s_2, c, r_2, p)) \mid c?(x).c?(y).\text{print} :: x + y\,]\,)$$

$$Sum(slice, channel, result, opt) \overset{\text{def}}{=} pub!opt \mid for!\langle slice, result\rangle.result?(z).channel!z$$

- Protocol P is **rejected** by GoPi compiler (with contextual option)
- Catalyser (parallel co-process) breaking the static scope invariant:

$$pub?(w).w?(u).u?(v)$$

How does it work?

- The procedure relies on a **type inference** algorithm implemented as a constraint system in **SMT-LIB**
- Types are qualified as *static* or *dynamic* and have *integer id*
- Hidden channels are qualified as *static* and are *identified*
- Processes forced to receive identifiers “in their scope”, or dynamic channels ($id = 0$)
- Contextual analysis always available through catalysers generated from process

Demo: Sum - type-checks and runs

```
marco@gopi$ cat examples/sum.pi ; gopi examples/sum.pi
#Sum protocol
let Sum = pub!p | f!s.r?j.c!j in
let P = new p { hide c [ p!c | Sum | c?x.print x] } in
let For = f?w.r!n in
P | For
****************************** GOPI ******************************
TYPE-CHECKED -- MAX ORDER: 3
GENERATING GO FILE gopiProcess.go
RUNNING THE PROCESS (go run gopiProcess.go)
**********Init*********
*****Running process proc1******
Waiting for value on f
Waiting for value on f
Waiting for value on c
Retrieved s from f
Waiting for value on c
Waiting for value on c
Waiting for value on r
Waiting for value on c
Retrieved n from r
Waiting for value on c
Retrieved n from c
Print n
fatal error: all goroutines are asleep - deadlock!
```

Demo: Sum (contextual option) – rejected

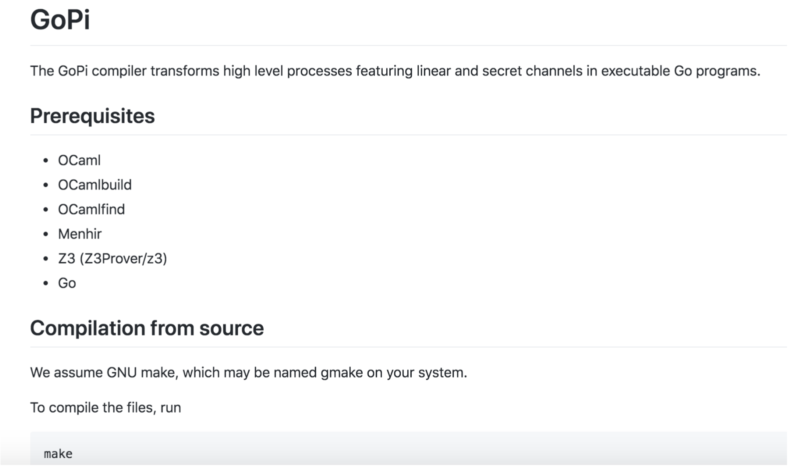
```
marco@gopi$ gopi -cat 3 -debug examples/sum.pi
******************************* GOPI *******************************
********************************************************************
Process does not type check
********************************************************************
PROCESS: new p { hide c [ p!c | pub!p | f!s.r?j.c!j
           | c?x.print x] } | f?w.r!n
CATALYSER: pub?(y).y?(z).z?(u).u?(v) | ...
********************************************************************
UNSAT CORE: (DebugMode is On)
(A5 A12 A20 A72 A79)
************************* SMT-LIB Header **************************
;; DATATYPES
(declare-datatypes () ((Scope static dynamic)))
(declare-datatypes () (
  (Chantype top
    (channel (scope Scope) (payload Chantype) (id Int)))))
;; FUNCTIONS
(define-fun equal ((c Chantype) (d Chantype)) Bool
  (= c d))
********************** SMT-LIB Constraints *************************
 (assert (! (= (id c) 101) :named A5))
 (assert (! (equal c (payload p)) :named A12))
 (assert (! (equal p (payload pub)) :named A20))
 (assert (! (and (equal (payload pub) y) (= (id  y) 0)) :named A72))
 (assert (! (and (equal (payload y) z) (= (id  z) 0)) :named A79))
```

Demo: Linearity – rejected

```
marco@gopi$ cat examples/mutual_simple.pi ; gopi -debug examples/mutual_simple.pi
#Mutual deadlock on linear channels a,b
<a,b> a!hello.b!world | b!x.a?y
****************************** GOPI ******************************
Symbolic linear channels: a b
Deadlock detection on a b is on
******************************************************************
Process does not type check
******************************************************************
UNSAT CORE: (DebugMode is On)
(A3 A4 A15 A16 A17 A39 A42 A43)
************************ SMT-LIB Header *************************
;; DATATYPES
(declare-datatypes () ((Scope static dynamic)))
;; i/o capabilities: 2 is used, 1 is used once, 0 is unused
(declare-datatypes () (
  (Chantype top
    (channel (scope Scope) (payload Chantype) (id Int) (i Int) (o Int) (ord Int)))))
********************* SMT-LIB Constraints ************************
 (assert (! (isChannel a) :named A3))
 (assert (! (and (>= (i a) 0) (<= (i a) 2) (>= (o a) 0) (<= (o a) 2)) :named A4))
 (assert (! (isChannel b) :named A15))
 (assert (! (and (>= (i b) 0) (<= (i b) 2) (>= (o b) 0) (<= (o b) 2)) :named A16))
 (assert (! (=> (isLinear b) (< (ord a) (ord b))) :named A17))
 (assert (! (=> (isLinear a) (< (ord b) (ord a))) :named A39))
 (assert (! (isLinear a) :named A42))
 (assert (! (isLinear b) :named A43))
```

Thanks!

https://github.com/marcogiunti/gopi



GoPi

The GoPi compiler transforms high level processes featuring linear and secret channels in executable Go programs.

Prerequisites

- OCaml
- OCamlbuild
- OCamlfind
- Menhir
- Z3 (Z3Prover/z3)
- Go

Compilation from source

We assume GNU make, which may be named gmake on your system.

To compile the files, run

```
make
```