GoPi: Compiling linear and static channels in Go

Marco Giunti

NOVA LINCS, NOVA School of Science and Technology, Portugal

Coordination, June 17 2020

Channels and programming languages

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

```
1  func sum(s []int, c chan int) {
2      sum := 0
3      for _, v := range s { sum += v }
4      c ← sum //send sum to c
5  }
6
7  func calc(c chan int) {
8      s := generateRandomSlice(1000)
9      go sum(s[:len(s)/2], c) //concurrent goroutine
10     go sum(s[len(s)/2:], c) //concurrent goroutine
11     x, y := ←c, ←c //receive from c
12     fmt.Printf("The sum of the slice is %d", x + y)
13 }
```

Channel forwarding

- Channels can be sent over channels, allowing to dynamically extend their scope
- In some situations this is too liberal, e.g., function *sum* should not need to distribute the communication channel *c*

```
1  func sum(s []int, c chan int) {
2      go func(){ pub ← c }() //c is forwarded -- backdoor?
3      ...
4      c ← sum //send sum to c
5  }
6
7  func () {
8      c := make(chan int)
9      calc(c)
10 }()  //scope of c
11
12 go func () { x := ←pub; x ← 0 }() //attacker opening scope
```

Designing protocols with no-forwarding

- Some apps as instant messengers already provide protection against message forwarding in order to strengthen secrecy
- To offer such protection, channel-based languages should feature a command to create **static channels** having a scope that **cannot be extruded**

```
1  func sum(s []int, c chan int) {
2      go func(){ pub ← c }() //c is forwarded -- backdoor?
3      ...
4  }
5
6  func () {
7      c := static_make(chan int) //proposal: Go 2
8      calc(c)
9  }()  //scope of c
10
11 go func () { x := ←pub; x ← 0 }() //attacker opening scope
```

Compile-time detection of scope extrusion

- Programs that at runtime can extrude the scope of a static channel should be rejected at compile-time

```
1  func sum(s []int, c chan int) {
2      go func(){ pub ← c }() //c is forwarded -- backdoor?
3  }
4
5  func () {
6      c := static_make(chan int) //proposal: Go 2
7      calc(c)
8  }()  //scope of c
9
10 go func () { x := ←pub; x ← 0 }() //attacker opening scope
```

- Code rejected by the compiler

```
sum.go:10:22: static channel may cross its boundary
```

GoPi

- In **this talk**, we present the GoPi compiler
- GoPi compiles high-level programs featuring linear and static channels into executable Go programs

INPUT $\textbf{let } Sum_i = \cdots \textbf{ in let } Calc = (\text{new } s_1, s_2)(Slice \mid Sum_1 \mid Sum_2) \mid Print \textbf{ in } [\text{hide } c][Calc] \mid For$

OUTPUT

```
************************** GOPI ****************************
TYPE-CHECKED -- MAX ORDER: 2
GENERATING GO FILE gopiProcess.go
RUNNING THE PROCESS (go run gopiProcess.go)
**********Init*********
Waiting for value on for3  Waiting for value on c ...
Retrieved s1 from for1  Retrieved s2 from for3 ...
Waiting for value on r1  Retrieved r1 from for2 ...
Retrieved slice2 from s2  Retrieved slice1 from s1 ...
Waiting for value on c  Retrieved slice1 from r1 ...
Retrieved res1 from c  Retrieved res2 from c ...
Print res1 + res2
```

Protocol specification and execution

- Aims:
  1. design
  2. analysis
  3. execution in channel-based runtime system

  of *message-passing protocols* featuring
  - **channel-over-channel** passing
  - **linear** channels
  - **static** channels
- Guidelines:
  - **avoid annotations**
  - **fully-automatic compilation** of well-behaved source specifications into executable target programs

Main features

- Language
  - compile-time detection of extrusion of the scope of channels declared as static with the $[\textbf{hide}\, x][P]$ construct
  - compile-time detection of deadlocks on channels declared as linear with the $\langle a, \ldots, z \rangle P$ construct
- Runtime system
  - realistic non-deterministic synchronizations
  - race-freedom

Example: Secret Chat protocol

- **Aim**: design an app (instance) offering **protection against message forwarding**
- Alice, Bob and Carl share an hidden chat channel with static scope including
  1. the users
  2. the board
  3. a setup process that distributes the channel to the users
- the scope of the channel should never be enlarged

Secret Chat in LSpi

- !, ?, . , $*$, |, indicate output, input, sequence, loop and parallel execution, respectively

let $Alice = priv?(c).c!\texttt{helloAlice}$ **in**

let $Bob = priv?(c).c!\texttt{helloBob}.pub!priv$ **in**

let $Carl = pub?(p).p?(c).c!\texttt{helloCarl}$ **in**

let $Board = *chat?(\texttt{message}).\textbf{print}::\texttt{message}$ **in**

let $Setup = *priv!chat$ **in**

let $Chat =$

$[\textbf{hide}\ chat][Board \mid (\textbf{new}\ priv)(Setup \mid Alice \mid Bob) \mid Carl]$ **in** $Chat$

- Specification is suspicious since the distribution channel *priv* is sent on public channel *pub*

Secret Chat: semantics

- Concrete:
  - compile *Chat* into an executable Go program and run it
- Abstract:
  - translate *Chat* into a (typed) pi calculus process
  - **hide** is mapped into **new** and has standard semantics
  - linear declarations separated from processes and used in the static analysis

$$[\![\, \mathit{Chat} \,]\!] \rightarrow^* (\mathbf{new}\, \mathit{chat})(\mathit{Board} \mid (\mathbf{new}\, \mathit{priv})(\mathit{pub}\mathbf{!}\mathit{priv} \mid \mathit{Setup}) \mid \mathit{chat}\mathbf{!}\texttt{helloAlice}) \mid \mathbf{print} :: \texttt{helloBob} \mid \mathit{Carl}$$

- Soundness: processes that extrude static channels must be rejected

$$\Gamma \not\vdash [\![\, [\mathbf{hide}\, c][a\mathbf{!}c] \mid a\mathbf{?}(x).P \,]\!]$$

Stand-alone and contextual analysis

- GoPi offers two levels of analysis

$$[\textbf{hide}\ \mathit{chat}][\mathit{Board} \mid (\textbf{new}\ \mathit{priv})(\mathit{Setup} \mid \mathit{Alice} \mid \mathit{Bob}) \mid \mathit{Carl}]$$

1. **Stand-alone**. *Chat* will not be composed with other processes
   - **safe**: all processes are included in the static scope of *chat*
2. **Contextual**. *Chat* will be composed with other processes
   - **unsafe**: there exists a "well-behaved" process that can open the scope of the static channel when ran in parallel with *Chat*
   - Process $\mathit{pub}\textbf{?}(x_{priv}).x_{priv}\textbf{?}(x_{chat}).Q$ is one of such processes

Linear channels

- To recover the protocol, we resort to **linear** channels that are used once in input and once in output (noted $\langle\cdot\rangle$)

  $\langle pub \rangle [\textbf{hide}\ chat][Board \mid (\textbf{new}\ priv)(Setup \mid Alice \mid Bob) \mid Carl]$

- the process above is **contextually safe**
  - we assume that composed processes running in parallel respect the linearity assumptions
- safety established by resolving **SMT-LIB** constraint system automatically generated from process and *catalyser*

```
;; 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)))))
```

Deadlock detection

- Deadlocks that may arise on linear channels are detected (some limitation on delegation)

let $Bob = priv?(c).c!\texttt{helloBob}.pub!priv.ack!\texttt{ok}$ **in**

let $Carl = ack?(x).confirm!x.pub?(p).p?(c).c!\texttt{helloCarl}$ **in** $\cdots$ **in**

let $ChatAck = \langle ack, pub \rangle Chat$ **in** $ChatAck$

Assertions for linear channels *ack* and *pub*

```
(assert (! (=> (isLinear ack) (< (ord pub) (ord ack))) :named A67))
(assert (! (=> (isLinear pub) (< (ord ack) (ord pub))) :named A96))
(assert (! (isLinear ack) :named A111))
(assert (! (isLinear pub) :named A112))
(assert (! (=> (isLinear ack) (and (= (o ack) 1) (= (o ack) (+ 1 0 ))))
 :named A113))
(assert (! (=> (isLinear ack) (and (= (i ack) 1) (= (i ack) (+ 1 0 ))))
 :named A114))
(assert (! (=> (isLinear pub) (and (= (o pub) 1) (= (o pub) (+ 1 0 ))))
 :named A137))
(assert (! (=> (isLinear pub) (and (= (i pub) 1) (= (i pub) (+ 1 0 ))))
 :named A138))
```

Generating typed Go code

- SMT-LIB channel types mapped into Go types by ignoring all fields but the *payload*
- Implementation of processes not straigthforward
  - Mapping send/receive processes **directly** into send/receive primitives **breaks** semantics of processes
  - In practice, **non-determinism** is almost eliminated
- Solution relies on structured communication protocol based on randomized message queues

Naive implementation in Go ($Carl = pub?(p).p?(c).c!\texttt{helloCarl}$)

- Rationale: LSpi processes are mapped directly in Go primitives
- Problems:

1. 90% of executions bind p to *priv* (**line 9**): should be 50%
2. channels have no name associated: "Retrieved:0xc000022060"

```
1  var pub chan chan chan base
2  //Chat process - non-linear version
3  func(){
4    chat := make(chan base) ; ...
5    func(){ ...
6      priv := make(chan chan base); ...
7      go func(){ ... ; pub ← priv}() //Bob
8    }()
9    go func(){ p := ←pub; fmt.Print("Retrieved:", p)
10                c := ←p; fmt.Print... ; c ← "HelloCarl"}() //Carl
11 }()
12 //Parallel process
13 go func(){ a := make(chan chan base) ; pub ← a}()
```

GoPi's approach

- Channel servers
  1. take care of input and output requests of client processes
  2. internally manage non-deterministic synchronizations and the naming of channels
- Access to channel servers regulated by an *API* for communication
- API implemented as methods of *type environment* infrastructure

Code generated by GoPi ($Carl = pub?(p).p?(c).c!\texttt{helloCarl}$)

```
 1 type typeEnv struct{
 2         ord0 struct{ ... }
 3         ord1 struct{
 4           toStr map[chan1]string //marshalling
 5           fromStr map[string]chan1 //unmarshalling
 6           queue map[chan1]queueChan0
 7           dequeue map[chan1]func() //instantiated at registration
 8           mux sync.Mutex } ...
 9 }; var Gamma typeEnv  ...
10 func(){ ...
11      Gamma.register("chat" + counter, "0")
12      chat := Gamma.chanOf("chat" + counter).(chan0)
13      go func() { ... }() //Board ... //Setup, Alice, Bob
14      go func() {
15          Gamma.dequeue(pub); p := ←pub
16          Gamma.dequeue(p); c := ←p
17          cReply4 := make(chan bool)
18          Gamma.queue(c, helloCarl, cReply4)
19          _ = ←cReply4; done ← true }() //Carl ... }()
```

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