GoPi: Compiling Linear and Static Channels in Go

Marco Giunti

▶ **To cite this version:**

Marco Giunti. GoPi: Compiling Linear and Static Channels in Go. 22th International Conference on Coordination Languages and Models (COORDINATION), Jun 2020, Valletta, Malta. pp.137-152, 10.1007/978-3-030-50029-0_9 . hal-03273983

HAL Id: hal-03273983
https://inria.hal.science/hal-03273983

Submitted on 29 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

GoPi: Compiling linear and static channels in Go

Marco Giunti*

NOVA LINCS, New University of Lisbon, Portugal
marco.giunti@gmail.com

Abstract. We identify two important features to enhance the design of communication protocols specified in the pi-calculus, that are linear and static channels, and present a compiler, named *GoPi*, that maps high level specifications into executable *Go* programs. Channels declared as linear are deadlock-free, while the scope of static channels, which are bound by a *hide* declaration, does not enlarge at runtime; this is enforced statically by means of type inference, while specifications do not include annotations. Well-behaved processes are transformed into Go code that supports non-deterministic synchronizations and race-freedom. We sketch two main examples involving protection against message forwarding, and forward secrecy, and discuss the features of the tool, and the generated code. We argue that GoPi can support academic activities involving process algebras and formal models, which range from the analysis and testing of concurrent processes for research purposes to teaching formal languages and concurrent systems.

1 Introduction

Concurrent programming is nowadays pervasive to most software development processes. However, it poses hard challenges to the developers, which must envisage and try to solve without automatic support undesired behaviours like security breaches, deadlocks, races, often leading to bugs of substantial impact [11,22]. Automated techniques and tools are thus needed to analyse and ensure secure and correct concurrent code. Formal methods have been advocated as an effective tool to analyse and deploy secure communicating programs and protocols [10]. Process calculi, in particular, allow to study prototype analysis techniques that could be embedded into next generation compilers for distributed languages, and to investigate high-level security abstractions that can be effectively deployed into lower-level languages, thus providing for *APIs* for secure process interaction (e.g., [5,2]).

* This work is partially supported by the EU Horizon 2020 research and innovation programme under the MSCA RISE grant agreement N° 77823 (BehAPI), and by Fundacão para a Ciência e a Tecnologia, Ministério da Ciência, Tecnologia e Ensino Superior, via project PTDC/CCI-COM/32166/2017 (DeDuCe). Tool available at: `https://github.com/marcogiunti/gopi`. Demo video available at: `https://sites.fct.unl.pt/gopi`.

$$\begin{aligned}
&\textbf{let } \mathit{Alice} = \mathit{priv}\textbf{?}(c).c\textbf{!}\texttt{helloAlice} \textbf{ in}\\
&\textbf{let } \mathit{Bob} = \mathit{priv}\textbf{?}(c).c\textbf{!}\texttt{helloBob}.\mathit{pub}\textbf{!}\mathit{priv} \textbf{ in}\\
&\textbf{let } \mathit{Carl} = \mathit{pub}\textbf{?}(p).p\textbf{?}(c).c\textbf{!}\texttt{helloCarl} \textbf{ in}\\
&\textbf{let } \mathit{Board} = *\mathit{chat}\textbf{?}(\texttt{message}).\textbf{print}::\texttt{message} \textbf{ in}\\
&\textbf{let } \mathit{Setup} = *\mathit{priv}\textbf{!}\mathit{chat} \textbf{ in}\\
&\textbf{let } \mathit{Chat} = [\textbf{hide } \mathit{chat}][\mathit{Board} \mid (\textbf{new } \mathit{priv})(\mathit{Setup} \mid \mathit{Alice} \mid \mathit{Bob}) \mid \mathit{Carl}] \textbf{ in } \mathit{Chat}
\end{aligned}$$

Fig. 1: Suspicious specification of a secret chat in the *LSpi* language

This paper presents a contribution towards this direction by introducing a fully-automated tool, named GoPi [1], that allows to analyse and run communication protocols specified in a variant of the pi calculus featuring linear channels that must be used exactly once for input and once for output, and static channels that are never extruded. Well-behaved high-level processes are mapped into executable Go programs communicating through message-passing: rather than enforcing the channels' constraints at the target language level, GoPi performs a static analysis of the specification and only generates executable Go code that at runtime preserves the specified invariants. The analysis is based on type inference, while the specification language does not include type decorations. GoPi supports further non-trivial features, which include a contextual analysis of static channels, and deadlock detection on linear channels, at the source language level, and non-deterministic synchronizations, and race-freedom, at the target language level.

The aim is twofold:

- to provide for an automated static analysis of processes described in a variant of the linear pi-calculus without relying on annotations;
- to make available a message-passing runtime system for well-behaved pi-calculus processes featuring static channels that are never extruded.

1.1 Message Forwarding Protection

To illustrate our approach, we consider the case when we want to study the design of a messaging application supporting secret chats[1] featuring *message forwarding protection*. To this aim, we analyse an instance of a secret chat that involves three users, and describe the protocol as follows: "*Alice, Bob*, and *Carl* share a *hidden chat* channel with *static* scope including the users, the board, and a setup process that distributes the channel to the users, where the scope of the channel should never be enlarged". The static scope invariant offers protection against message forwarding, and only processes that are included in the scope of the channel in the specification will be able to ever use the channel at runtime.

[1] https://www.viber.com/blog/2017-03-13/share-extra-confidently-secret-chats

Figure 1 presents a formal specification of the protocol in a variant of the pi-calculus featuring secret channels. The program is based on message-passing and builds around three main channels: the hidden channel *chat*, the distribution channel *priv*, and a public channel *pub*. Base channels are noted in typewriter. We use !, ?, . , $*$, and | to indicate output, input, sequence, loop and parallel execution constructors, respectively; channels are created with the new and hide constructors by indicating their scope with parentheses (new) and squares (hide). The print imperative construct allows to print channels. In order to be safe, the program in Figure 1 should preserve the static scope invariant, that is: *the scope of the hidden channel must not be enlarged at runtime*. The specification is suspicious since Carl, who is left out of the distribution process, is invited to the chat by receiving the private channel *priv* from the open channel *pub*, perhaps because of a bad design choice.

By running GoPi, we verify that, when considered in isolation, the program in Figure 1 is safe: intuitively, this holds since all processes receiving the hidden channel are included in its static scope (the squares). However, the protocol is flagged as *contextually unsafe*: the reason is that there exists a process that, once put in parallel with the *Chat* process, can break the static scope invariant by receiving the hidden channel. That is, because of non-determinism, the private channel *priv* can be received by a parallel process that is listening on the open channel *pub*, rather than by Carl, thus allowing a process outside the squares to receive the hidden channel *chat*. To fix to the program in Figure 1 we can resort to *linear channels* that must be used exactly once for input and once for output. By declaring *pub* as linear, written as $\langle pub \rangle$, the protocol $SafeChat \triangleq \langle pub \rangle Chat$ gains protection from parallel (typed) processes, which are assumed to do not break linearity, and in turn contextual safety, as established by GoPi.

The static analysis is relevant since, in general, detecting if a program may extrude a secret channel by code inspection can be hard, because of channel mobility, and of the arbitrary length of the attack sequence. To see that, take $P \triangleq (\mathbf{new}\, a_1, \ldots, a_n)([\mathsf{hide}\, c][a_n!c] \mid a_1!a_2 \mid \cdots \mid a_{n-1}!a_n \mid pub!a_1)$, for some $n > 1$: the secret channel c is sent over a restricted channel a_n, which in turn is sent over a restricted channel a_{n-1}, and so on, while the error is that the first channel in the chain, a_1, is sent over a public channel *pub*, allowing processes running in parallel with P to receive the hidden channel from a_n.

1.2 Related Work

We briefly discuss work related to the design of the specification language, and to runtime systems for process calculi and Go as a target language.

Language Design Secret channels have been studied by the author at the language [16] and type [15] level; this work integrates those results by presenting a compiler based on a novel type inference algorithm. The paper [16] presents a variant of the pi-calculus introducing a further operator, *hide*, that allows to declare channels that can be passed over channels, but cannot be extruded, and studies its behavioural properties. The static scope mechanism is embedded in

the operational semantics of the language, where a dynamic check ensures that the context cannot receive channels protected by *hide*. In subsequent work [15], the mechanism is shifted to the level of types by means of a declarative system that enforces the static scope invariant in a standard pi-calculus. These mechanisms, complemented with linear type qualifiers (cf., [18,14]) and deadlock detection (cf., [17]), are the core of the static analysis performed by the GoPi tool.

Static channels and boundaries in process calculi have been investigated since the origins of this research area [28], and more recently in, e.g., [6,26,7]. The work in [6] has similarities with our approach and introduces a pi-calculus featuring a group creation operator, and a typing system that disallows channels to be sent outside of the group. Programmers must declare which is the group type of the payload: the typing system rules out processes of the form $Q \triangleq (\mathsf{new}\, p\colon U)(P \mid (\mathsf{new}\, G)((\mathsf{new}\, x\colon G[])(p!x)))$ since the type U of channel p cannot mention the secret type G, which is local. In contrast, we do not rely on type decorations and accept process Q whenever x is hidden and P does not allow to extrude x, e.g., P does not input on p or distribute p. From the point of view of the language design, we share some similarity with the ideas behind the boxed pi-calculus [26]. A box in [26] acts as wrapper where we can confine untrusted processes; communication among the box and the context is subject to a fine-grained control that prevents the untrusted process to interfere with the protocol. Our *hide* construct is based on the symmetric principle: a process is trusted whenever contexts cannot interfere with the process' protocol, that is contexts cannot enlarge the scope of the hidden channels of the process.

Runtime System To the best of our knowledge, most interpreters for distributed calculi do not rely on channel-based mechanisms at the target language level; such implementations, pioneered by [29,25,27] for the pi-calculus, are commonly based on simulating non-determinism and concurrency by process interleaving. Previous attempts to develop calculi-inspired languages with native support for channel-over-channel passing include *JoCaml* [12], where mobility is now discontinued [23].

Recently, a behavioural static analysis of Go programs based on multiparty session types (*MPST*, [19]) has been presented in [20,21]. The approach followed in that line of work consists in analysing existing Go programs to ensure stronger properties at compile-time, e.g., deadlock-freedom. None of those works, however, support channel-over-channel passing. Castro et al. [8] introduced a framework to translate distributed MPST written in the *Scribble* protocol language into a Go API; safety in API's clients is enforced at runtime by generating linearity exceptions. Differently, we obtain safety of Go programs statically by means of type inference of pi-calculus channels.

Structure of the Paper

§ 2 presents the specification language and the notion of error, and sketches few examples. The next two sections introduce the two main parts of the GoPi com-

piler: the static analyser, presented in § 3, and the Go code generator, presented in § 4. We conclude in § 5 by envisioning possible usage scenarios of GoPi, and by discussing limitations and future work.

2 The LSpi Specification Language

This section introduces the syntax of the language processed by the GoPi compiler. We consider communication channels, or variables, $a, \ldots, z$, and processes generated by the grammar:

$$P, Q \ ::= x\mathbf{!}v.P \mid x\mathbf{?}(y).P \mid (P \mid Q) \mid \mathbf{0} \mid [\mathbf{hide}\, x][P] \mid (\mathbf{new}\, x)(P) \mid *P \mid \langle a, \ldots, x\rangle\, P \mid \mathbf{let}\ X = P\ \mathbf{in}\ Q \mid X \mid \mathbf{print} :: v$$

Most operators are standard for message passing languages, with some exceptions. We have primitives for sending and receiving channels and continuing as P, noted as $x\mathbf{!}v.P$ and $x\mathbf{?}(y).P$, respectively, for parallel composition, noted $P \mid Q$, for inert processes, noted $\mathbf{0}$, for channel creation, noted $(\mathbf{new}\, x)(P)$, for process variables, noted X, and for assigning processes to process variables, noted $\mathbf{let}\ X = P\ \mathbf{in}\ Q$. The hide operator is the main feature of the language and shall be interpreted as follows: $[\mathbf{hide}\, c][P]$ declares that the fresh channel c should be confined into the (fixed) square brackets *even* when process P interacts with other processes. In the pi-calculus jargon, this is better summarized by the sentence: "scope extrusion of channel c is disallowed ". The other crucial feature is the linear channel declaration $\langle a, \ldots, x\rangle P$, which declares that each of the channels $a, \ldots, x$ must be used exactly once for input and once for output. Loops are programmed with the construct $*P$, which executes P forever. The construct $\mathbf{print} :: v$ supplies an imperative command to observe the channel v.

We assume the usual notions of free and bound variables and process variables, which we deem pairwise distinct by following the Barendregt convention, and let x be bound in $[\mathbf{hide}\, x][P]$, $(\mathbf{new}\, x)(P)$, and $a\mathbf{?}(x).P$, and be free otherwise, and X be bound in $\mathbf{let}\ X = P\ \mathbf{in}\ Q$, and free otherwise. The process $\mathbf{let}\ X = P\ \mathbf{in}\ Q$ is acyclic whenever X is not free in P, and P, Q are acyclic; the remaining cases are homomorphic. We only consider acyclic processes not containing free process variables. We will often avoid training nils, use the _ variable wildcard, and refer to channels not used in input or output as to base values, and write them in typewriter style, when convenient.

2.1 Runtime and Errors

GoPi allows to run LSpi processes by mapping well-behaved processes into executable Go programs. At a more abstract level, the semantics of the language is provided by translating LSpi processes into standard (typed) pi-calculus processes: intuitively, the hide construct is mapped into a restriction and has standard semantics (cf., [15]), while linear annotations are separated from processes and used in the static analysis. For instance, the specification $[\mathsf{hide}\, c][a\mathbf{!}c] \mid$

$a?(x).P$ declares that c should be confined in the squares, while at runtime P can receive the restricted channel c: therefore this process is unsound and should be rejected at compile-time.

LSpi programs can contain three kind of errors, all detected by the GoPi compiler:

(A) channels declared as hidden that can be received by processes outside the static scope of the channels;
(B) channels declared as linear that are not used exactly once for input and once for output;
(C) channels declared as linear that at runtime give rise to deadlocks.

Examples Process *Chat* in Figure 1 does not contain errors. In contrast, process $Chat \mid P$, where $P \triangleq pub?(x_{priv}).x_{priv}?(x_{chat}).Q$, is an error of kind A: there is a sequence of reductions which leads to the instantiation of the variable x_{chat} in Q with the hidden channel *chat*, that is the channel *chat* can be received by a process outside its static scope. Because of that, GoPi flags *Chat* as contextually unsafe. Process $SafeChat \triangleq \langle pub \rangle Chat$ does not contain errors, and is contextually safe, as we will see in § 3: intuitively, this holds since process P above is no longer a valid (typed) opponent, because channel *pub* is linear and cannot be accessed by the context.

To see an example of an error of kind B, take process $\langle priv \rangle Chat$, where channel *priv* is declared as linear. The linear invariant does not hold, because channel *priv* is used three times in input, by Alice, Bob and Carl (through delegation), respectively, and an unbound number of times in output, by process *Setup*.

Typical errors of kind C are processes containing self-deadlocks, which arise when a linear input (output) prefixes a continuation containing the matching output (input), and processes containing mutual deadlocks. The variant of process *Chat* below, where an *ack* is sent after sending channel *priv* over channel *pub*, and where channels *ack* and *pub* are linear, contains a mutual deadlock:

$$\begin{aligned} \cdots \quad & \textbf{let } Bob = priv?(c).c!\texttt{helloBob}.pub!priv.ack!\texttt{ok} \textbf{ in} \\ & \textbf{let } Carl = ack?(x).confirm!x.pub?(p).p?(c).c!\texttt{helloCarl} \textbf{ in } \cdots \textbf{ in} \\ & \textbf{let } ChatAck = \langle ack, pub \rangle Chat \textbf{ in } ChatAck \end{aligned} \tag{1}$$

At runtime the continuation of process *Bob* will be stuck on the output on the linear channel *pub*, which can be only unblocked by *Carl*, because *pub* is linear and must be used exactly once for input and once for input. Since *Carl*, in turn, is blocked on the linear channel *ack*, the process will deadlock.

An interesting example of security error is process *FSA* below, which abstracts a forward secrecy attack. Process *FSA* distributes a secret channel c on a private channel a, sends a password on c, and afterwards releases channel c on a public channel *pub*:

$$FSA \triangleq (\mathsf{new}\, a)([\mathsf{hide}\, c][a!c.c!\texttt{pwd} \mid a?(x).x?(\_).pub!x]) \mid pub?(z).Q \tag{2}$$

```
;; DATATYPES
(declare-datatypes () ((Scope static dynamic)))
(declare-datatypes () ((ChanType top
 (channel (scope Scope)(payload ChanType)(id Int)(i Int)(o Int)(ord Int)))))
```

Fig. 2: LSpi types in the SMT-LIB language

By considering that a hide is mapped into a new at runtime, process *FSA* might be interpreted as secure, because the context cannot observe the exchange over the restricted channel c, and in turn cannot retrieve the password. However, preserving the invisibility of restricted communications when pi-calculus processes are deployed in open, untrusted networks is problematic, exactly because of scope extrusion (cf., [3]), and eventually leads to complex solutions based on cryptographic protocols relying on trusted authorities (cf., [5]). For these reasons, we advocate that processes relying on dynamic scope restriction for security should be rejected (cf., [16,15]). In fact, process *FSA* contains an error of kind A, because at runtime the secret channel c can be received by a process outside the squares, that is c can be received from *pub*.

The forward secrecy attack hints on how to use secret channels to develop more secure programs: *whenever a secret is sent over an hidden channel of an error-free process, the secret will be unknown outside the static scope of the hide declaration.* Process *FSecret* is one of such secure programs, where we note that the distribution channel a can occur in processes outside the scope of the hide:

$$FSecret \triangleq (\mathsf{new}\, a)(\mathsf{new}\, b)(([\mathsf{hide}\, c][a!c.c!\mathtt{pwd} \mid a?(x).x?(\_)] \mid b!a \mid b?(\_)))$$

3 Static Analyser

The static analyser is based on the type inference of LSpi channels and is implemented as an automatically generated constraint system written in the *SMT-LIB* language [4], and decided through the *Z3* theorem prover [24]. Notably, the constraint system does not make use of quantifiers.

Figure 2 presents the syntax of the type of LSpi channels, named *ChanType*: base values are represented by the *top* constructor, while channels are built with the *channel* constructor receiving six arguments, where the last three (integer) constructors are for linearity. Type inference of a process P relies on a set of *allowed* identifiers (cf., *id*), which are the type identifiers that each input process is allowed to receive. Roughly, the static scope analysis is based on this technique.

To illustrate, consider the encoding[2] of the forward secrecy attack *FSA* in (2); the input on a is allowed to receive both (dynamic) channels tagged with 0 and the static channel identified by id_c, while the input on *pub* can only receive

[2] The main rationale is that a new is mapped into a new with a dynamic type tagged with 0, while a hide is mapped into a new with a static type tagged with a positive identifier.

channels tagged with 0:

$$(\mathsf{new}\, a\colon \mathsf{dyn@0})((\mathsf{new}\, c\colon \mathsf{stat@}id_c)a!c.c!\mathtt{pwd} \mid a?(x).x?(\_).pub!x)) \mid pub?(z).Q$$

The corresponding SMT-LIB assertions generated by GoPi enforce the invariants for a and pub through their payload, where the randomly generated identifier that instantiates id_c is 345:

```
(assert (! (= (id c) 345) :named A5))
(assert (! (= c (payload a)) :named A12))
(assert (! (and (= (payload a) x) (or (= (id x) 0) (= (id x) 345)):named A23))
(assert (! (= x (payload pub)) :named A46))
(assert (! (and (= (payload pub) z)  (= (id  z) 0)) :named A48))
```

These assertions make the model *UNSAT*, as expected, because by transitivity we obtain $345 = 0$: that is, the variable z bound by the input prefix on channel pub should have id equal to 0, while it has the id of the static (hidden) channel.

3.1 Contextual Safety

Contextual safety is analysed by resorting to auto-generated catalysers (cf., [9]) of order n, that are processes that can both inject and receive channels, on which they inject and receive channels, and so on, with depth n. Catalysers are put in parallel with the process in order to collect the process' global constraints, as if the process was immersed in an arbitrary (typed) context. The contexts under consideration are those that respect the linearity invariants of the process: that is, we generate catalysers from the unrestricted free variables of the process.

To see an example of catalyser, consider process *Chat* in Figure 1, where we note that the only unrestricted free variable of *Chat* is *pub*. The catalyser below is generated by following the structure of *Chat* and by matching each input (output) on *pub* with an output (input) on *pub* with depth three, which is the maximum order of *Chat*, where f is a randomly generated channel distinct from any channel in the free and bound variables of *Chat*:

$$Cat \triangleq pub?(x).(x?(y).y?(z) \mid \mathbf{0}) \mid pub!f.(f?(x).x?(y) \mid f?(x).(x?(y).y?(z) \mid \mathbf{0}))$$

Process *Chat* is contextually unsafe because *Chat* | *Cat* contains an error: the hidden channel *chat* at runtime can be received by process *Cat*, which is outside the static scope of the channel (cf., § 2). This is established by GoPi via the generation of the SMT-LIB assertions of *Chat* | *Cat*, and by discovering that the model is UNSAT; we omit the core assertions, which are similar to those of the forward secrecy attack.

As a further example, consider $SafeChat \triangleq \langle pub \rangle Chat$. Given that the set of the unrestricted free variables of *SafeChat* is empty, we generate an inert catalyser (cf., $\mathbf{0}$), and in turn obtain that *SafeChat* is contextually safe because the SMT-LIB model generated from *SafeChat* | $\mathbf{0}$ is *SAT*, that is the parallel composition is error-free.

3.2 Linearity Analysis

To enforce linearity, we use the input, output, and order integer constructors, noted *i*, *o*, and *ord*, respectively, of the type *ChanType* in Figure 2. Input (output) fields contain the number of times that the input (output) capability is used for a variable of the given type. Order fields are manipulated by the solver to find an ordering among linear channels.

The linearity analysis is performed by mapping the actual usage of channels into assertions of the constraint system. While analysing processes and generating the corresponding assertions for type reconstruction, we build a usage table that maps channels x to entries of the form (n_i, n_o, ls), where n_i, n_o are integers tracking the usage of x in input and output, respectively, and ls is a list containing the channels where x has been sent. At the end of the process analysis, the contents of the table are transformed into assertions and added to the constraint system.

The SMT-LIB assertions below are an excerpt of the model generated from process *ChatAck* in (1):

```
(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))
```

Assertions A111 and A112 come from the linear declaration $\langle ack, pub \rangle$ in (1). Assertions A113, A114, A137, and A138 are generated from the usage table, where, for each conjunction, the first entry is the expected value, and the second entry is the actual value. The assertions are satisfiable: that is, each i/o capability of channel *ack*, and of channel *pub*, respectively, is used exactly once in (1). The model is UNSAT because the conclusions in the assertions A67 and A96 state that the order of *pub* is smaller than the order of *ack*, and vice-versa. We note that the unsatisfiability of the model prevents the mutual deadlock inside *ChatAck* (cf., § 2).

4 Go Code Generation

Given a well-behaved LSPI process, and the type of its channels, GoPi generates executable Go code that is based on the channels' types. Channel types in Go have the following syntax[3], where ElementType is any type:

```
ChannelType = ( "chan" | "chan" "<-" | "<-" "chan" ) ElementType .
```

[3] https://golang.org/ref/spec

```
1  var pub chan chan chan base
2  //Chat process
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); ... }(); //Carl
10 }()
11 //Parallel process
12 go func(){ a := make(chan chan base) ; pub ← a}()
```

Fig. 3: Naive implementation of the *Chat* protocol in Go

We map types in Figure 2 to types of the form above by ignoring all fields but the *payload*, and by mapping the *top* type to **string**.

The generation of code implementing LSpi processes is not straightforward: while the target language features concurrent goroutines (cf., **go** f(a ,..., z)) that are a natural candidate to represent high-level parallel processes, the whole application's design must be carefully pondered.

As a first attempt, we could map input and output constructs of LSpi directly into receive and send primitives of Go, respectively. To illustrate, take the parallel execution of process *Chat* in Figure 1 with a process sending a fresh channel a over the public channel *pub*, that is process $Chat_{\mathsf{ND}} \triangleq Chat \mid (\mathsf{new}\, a)(pub!a)$, where the subscript stands for non-deterministic, since *Carl* can receive *priv* from *Bob*, or a from the parallel process, non-deterministically. Process $Chat_{\mathsf{ND}}$ would be mapped into Go code of the form outlined in Figure 3, where we list the parts that are related to the communication over channel *pub*. The scope of channel *chat* is grouped by the function call in lines 3–10, while the scope of channel *priv* is grouped by the function call in lines 5–8. The listed processes that are executed concurrently are *Bob* (line 7), *Carl* (line 9), and the parallel process (line 12).

While appealingly simple, the implementation in Figure 3 has at least two main drawbacks:

- in the vast majority of cases, i.e., $\sim 90\%$, p is bound to *priv*, while the probability should be 50%, being receiving *priv* from *pub* equally probable to receiving a from *pub*;
- channels have no name associated, making difficult the interpretation of the output of the program, e.g., "*Retrieved: 0xc000022060*".

4.1 Channel Servers

The envisioned solution consists in using channel servers that take care of input and output requests of clients, while internally managing both non-deterministic synchronizations, and the naming of channels. The access to channel servers is regulated by an API for communication, implemented as methods of a *type*

```
1  type base string
2  type basePair struct{
3      ch base
4      replych chan bool
5  }
6  type queueBase []basePair;
7  type chan0 chan base
8  type chan0Pair struct{
9      ch chan0
10     replych chan bool
11 }
12 type queueChan0 []chan0Pair;
13 type chan1 chan chan0 ; ...
14 type typeEnv struct{
15       ord struct{ ... }
16       ord0 struct{
17         toStr map[chan0]string //marshalling
18         fromStr map[string]chan0 //unmarshalling
19         queue map[chan0]queueBase
20         dequeue map[chan0]func() //instantiated at registration
21         mux sync.Mutex
22       }
23       ord1 struct{
24         toStr map[chan1]string //marshalling
25         fromStr map[string]chan1 //unmarshalling
26         queue map[chan1]queueChan0
27         dequeue map[chan1]func() //instantiated at registration
28         mux sync.Mutex
29       } ; ...
```

Fig. 4: Type of channel servers

environment infrastructure; the structure, represented by the *typeEnv* typed collection in Figure 4, aggregates channel servers by their order.

Servers are equipped with dynamic arrays, referred as *queues*, that collect the values concurrently sent on the channel by output clients, and act as a bridge between input and output clients: input clients send requests to the server and receive values sent by output clients and stored in the queue. Non-determinism is simulated through a randomization of queues, and can be pushed forward by tuning the timeouts in retrieving messages[4]. A *mutex* regulating the access to queue and dequeue operations prevents data races; this is verified with Go's race detector.

Server Registration A channel server of order $n \geq 0$ is registered by instantiating the entries of ord_n in the (unique) variable Γ of type *typeEnv* (cf., Figure 4). The procedure to register a channel server of order zero for the name "a", where, by convention, zero is the order of channels conveying base values, consists of five major steps:

1. create a fresh channel c of type *chan0*;
2. acquire the lock (cf., line 21)
3. **defer** the unlock
4. insert the mappings between "a" and c (cf., lines 17, 18)

[4] Non-zero dequeue timeouts are optional, and discouraged for non-academic purposes.

5. insert the mapping from c to a function (cf., line 20) that retrieves values from *Gamma.ord0.queue[c]* (cf., line 19).

4.2 Clients' Access to Servers

The channels servers are accessed by clients by means of methods of the variable Γ of type *typeEnv*. The signatures below list the most relevant operations.

```
1 //Methods of typeEnv accessed by clients
2 func (t *typeEnv) register(name string, nameType string) error
3 func (t *typeEnv) dequeue(input value) error
4 func (t *typeEnv) queue(output value, payload value,
5                         replyCh chan bool) error
6 func (t *typeEnv) nameOf(c value) string
```

The *register* method is invoked by clients in correspondence of a new or of a hide declaration, where the second parameter is the order of the declared channel. The *dequeue* method is called by input clients, where *value* is an **interface** implemented by channels and base values. The *queue* method is invoked by output clients, where the third parameter will be instantiated by a (fresh) ack channel, to enforce synchronous communications. The *nameOf* method is called by print clients in order to print the string associated to a channel reference.

4.3 Working Example

Figure 5 contains the code generated by GoPi for the *Chat* process (cf., Figure 1), where we only list the code of clients, being the code of servers invariant. The outer function call generates channel *chat* and closes its scope. In the body of the call, we have the parallel execution of *Board* (lines 5–20), of (new *priv*)(*Setup* | *Alice* | *Bob*) (lines 21–51), and of *Carl* (lines 52–62). Generation of fresh channels is implemented by a mechanism that uses randomly generated keys, and a counter protected by a mutex, for loops (cf., lines 7, 9, 27, 29).

The code implementing *Board* invokes the *dequeue* method of Γ (line 10), which triggers the selection of a message m from the queue of channel *chat* and the dispatch of m over *chat*. Subsequently, the message is retrieved from *chat* and printed, where the code in lines 13–17 implements the polymorphic **print** construct of LSpi. The sending on channel *done* (line 19) is discussed below.

The code for *Setup* continously uses the *queue* method of Γ to send *chat* over *priv* (cf., lines 28–33); to enforce synchrony, the write request includes a reply boolean channel that will be unblocked by the server once *priv* is retrieved in the queue (cf., lines 30, 32).

The code for *Bob* sends three requests to Γ: one *dequeue*, to retrieve a channel from *priv* (line 39), one *queue*, to send the string *helloBob* over the channel retrieved from *priv* (line 42), and one *queue*, to send *priv* over *pub* (line 45). Before the exit, a boolean ack is sent over channel *done* (line 47), to signal that the thread ended. The ack is received by the loop in line 63, which allows the program to wait for the termination of all threads until a given timeout,

```
var Gamma typeEnv ; ...
func(){
  Gamma.register("chat"+ string(counter.Value(key)), "0")
  chat := Gamma.chanOf("chat"+ string(counter.Value(key))).(chan0)
  //Board
  go func() {
    key := RandStringRunes(32)
    for{
      counter.Inc(key)
      Gamma.dequeue(chat)
      message := ← chat
      var v value = message
      switch v.(type){
        case base: fmt.Printf("Print %v\n",message)
        default: fmt.Printf("Print %v with address %v\n",
                            Gamma.nameOf(message), message)
      }
    }
    done ← true
  }()
  go func() {
    func() {
      Gamma.register("priv"+ string(counter.Value(key)), "1")
      priv := Gamma.chanOf("priv"+ string(counter.Value(key))).(chan1)
      //Setup
      go func() {
        key := RandStringRunes(32)
        for{
          counter.Inc(key)
          privReply0 := make(chan bool)
          Gamma.queue(priv, chat, privReply0)
          _ =← privReply0
        }
        done ← true
      }()
      //Alice ...
      //Bob
      go func() {
        Gamma.dequeue(priv)
        ch2 := ← priv
        ch2Reply2 := make(chan bool)
        Gamma.queue(ch2, helloBob, ch2Reply2)
        _ =← ch2Reply2
        pubReply3 := make(chan bool)
        Gamma.queue(pub, priv, pubReply3)
        _ =← pubReply3
        done ← true
      }()
    }()
    done ← true
  }()
  //Carl
  go func() {
    Gamma.dequeue(pub)
    ch3 := ← pub
    Gamma.dequeue(ch3)
    c := ← ch3
    cReply4 := make(chan bool)
    Gamma.queue(c, helloCarl, cReply4)
    _ =← cReply4
    done ← true
  }()
  for { ←done }
}()
```

Fig. 5: GoPi's implementation of the *Chat* process

to increase the chances to retrieve messages from queues. This mechanism is followed by all threads, regardless of loops.

Finally, the code for *Carl* sends three requests to Γ: one *dequeue*, to retrieve a channel from *pub* (line 54), one *dequeue*, to retrieve a channel c from the channel retrieved from *pub* (line 56), and one *queue*, to send the string *helloCarl* over c (line 59).

5 Discussion

GoPi's main aim is to support academic activities involving process algebras and formal models, which range from the analysis and testing of concurrent processes for research purposes to teaching formal languages and concurrent systems.

In this context, we have done some tests[5] with encouraging results, e.g, GoPi decided the safety of a complex variant of the secret chat protocol of § 1 involving a communication of order seven and more than thirty programming constructs in 0.2 seconds, producing 600 constraints and a Go file of 1Kloc (cf.,[1]). On the Go's side, we ran the code generated from a LSpi process continuously creating, sending and printing fresh channels for one day, without encountering exceptions. With António Ravara, we plan to use GoPi in the course Modelling and Validating Concurrent Systems of the Integrated Master in Computer Engineering, New University of Lisbon, 2020/21.

5.1 Limitations

The current architecture of GoPi does not allow to separate the static analysis from the generation of the Go code, and in turn to generate code based on type annotations provided by different tools. Another limitaton is that modifications of the Go code made by the programmer are lost when the specification is changed, since GoPi does not support annotations of the specification with Go snippets. We also note that the static analysis is not compositional, since to determine whether a process is safe, we perform a contextual analysis.

The information reported in case of failure of the analysis is not parsed into an human-readable format; this limits the usability of the tool.

At the language level, one current limitation is that delegation of partial capabilities of linear channels is rejected, because of issues related to the detection of deadlocks (cf., [17]). Another limitation, which is common in the context of behavioural type systems (cf., [13]), is that deadlocks are detected on linear channels, while unrestricted channels, interpreted as open ports, can give rise to runtime locks caused by decoupled input and output communications.

5.2 Future Work

GoPi aims at being an open and live project developing and maintaining a compiler for a language with built-in support for mobility, security, resource-

[5] Testing machine: MacBook 2GHz i5 8GB 1867 MHz LPDDR3.

awareness, and deadlock-resolution. In that direction, most limitations outlined above need to be overcome.

The separation of the static analysis and of the generation of Go code, and the readability of the output of the static analysis, appear as the most urgent issues. We believe that both features could be supported in the next release of GoPi, while the presentation of the results of the static anaysis could (at least) state a list of channels, and the kind or error encountered (cf., § 2).

Supporting partial delegation of linear capabilities is another feature that we are keen to support in future releases, while the static analysis may be more involved, because of deadlock detection.

Acknowledgements

The author would like to warmly thank the anonymous reviewers for their competent comments and constructive criticism on a previous draft of the paper, and for providing insightful suggestions in the preparation of this paper.

References

1. The GoPi compiler, `https://github.com/marcogiunti/gopi`, `https://sites.fct.unl.pt/gopi`
2. 4th Workshop on Principles of Secure Compilation (2020), POPL, `https://popl20.sigplan.org/home/prisc-2020`
3. Abadi, M.: Protection in programming-language translations. In: ICALP. LNCS, vol. 1443, pp. 868–883. Springer (1998)
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017)
5. Bugliesi, M., Giunti, M.: Secure implementations of typed channel abstractions. In: POPL. pp. 251–262. ACM (2007)
6. Cardelli, L., Ghelli, G., Gordon, A.D.: Secrecy and group creation. Inf. Comput. **196**(2), 127–155 (2005). https://doi.org/10.1016/j.ic.2004.08.003
7. Castagna, G., Vitek, J., Nardelli, F.Z.: The seal calculus. Inf. Comput. **201**(1), 1–54 (2005). https://doi.org/10.1016/j.ic.2004.11.005
8. Castro, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in Go: statically-typed endpoint APIs for dynamically-instantiated communication structures. PACMPL **3**(POPL), 29:1–29:30 (2019). https://doi.org/10.1145/3290342
9. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Mathematical Structures in Computer Science **26**(2), 238–302 (2016)
10. Cortier, V., Kremer, S. (eds.): Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security, vol. 5. IOS Press (2011)
11. Fonseca, P., Li, C., Singhal, V., Rodrigues, R.: A study of the internal and external effects of concurrency bugs. In: DSN. pp. 221–230. IEEE Computer Society (2010). https://doi.org/10.1109/DSN.2010.5544315
12. Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: JoCaml: A language for concurrent distributed and mobile programming. In: International School on Advanced Functional Programming. pp. 129–158. Springer (2002)

13. Gay, S., Ravara, A. (eds.): Behavioural Types: from Theory to Tools. River Publishers (2017). https://doi.org/0.13052/rp-9788793519817
14. Giunti, M.: Algorithmic type checking for a pi-calculus with name matching and session types. The Journal of Logic and Algebraic Programming **82**(8), 263–281 (2013). https://doi.org/10.1016/j.jlap.2013.05.003
15. Giunti, M.: Static semantics of secret channel abstractions. In: NORDSEC. LNCS, vol. 8788, pp. 165–180. Springer (2014)
16. Giunti, M., Palamidessi, C., Valencia, F.D.: Hide and New in the Pi-Calculus. In: EXPRESS/SOS. EPTCS, vol. 89, pp. 65–79 (2012)
17. Giunti, M., Ravara, A.: Towards static deadlock resolution in the π-calculus. In: TGC 2013. LNCS, vol. 8358, pp. 136–158. Springer (2014)
18. Giunti, M., Vasconcelos, V.T.: Linearity, session types and the pi calculus. Mathematical Structures in Computer Science **26**(2), 206–237 (2016). https://doi.org/10.1017/S0960129514000176
19. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016)
20. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off Go: liveness and safety for channel-based programming. In: POPL. pp. 748–761. ACM (2017)
21. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in Go using behavioural types. In: ICSE. pp. 1137–1148. ACM (2018). https://doi.org/10.1145/3180155.3180157
22. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS. pp. 329–339. ACM (2008). https://doi.org/10.1145/1346281.1346323
23. Mandel, L., Maranget, L.: The JoCaml language, `http://jocaml.inria.fr/doc`, Release 4.01, March 14, 2014
24. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
25. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Proof, Language, and Interaction, Essays in Honour of Robin Milner. pp. 455–494. The MIT Press (2000)
26. Sewell, P., Vitek, J.: Secure composition of untrusted code: Box pi, wrappers, and causality. J. Comp. Sec. **11**(2), 135–188 (2003)
27. Sewell, P., Wojciechowski, P.T., Unyapoth, A.: Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. ACM Trans. Program. Lang. Syst. **32**(4), 12:1–12:63 (2010). https://doi.org/10.1145/1734206.1734209
28. Thomsen, B.: Plain CHOCS: A second generation calculus for higher order processes. Acta Inf. **30**(1), 1–59 (1993). https://doi.org/10.1007/BF01200262
29. Turner, D.N.: The Polymorphic Pi-calculus:Theory and Implementation. Ph.D. thesis, University of Edinburgh (1995)