Compiling linear and static channels in Go
Extended Abstract

Marco Giunti*

NOVA LINCS, New University of Lisbon, Portugal

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 with their own forces undesired behaviours like security breaches, protocol incompatibilities, deadlocks, races, livelocks; this is a very difficult and error-prone task, requiring much more than just the programmer's skills: concurrency bugs appear frequently and have a substantial impact [11,21]. Automated techniques and tools are thus needed to analyse and ensure correct concurrent code. Crucially, in order to be effective, the techniques must tackle the struggle to balance the effort requested to the programmer, and the level of sophistication of the properties ensured by the development process.

This abstract presents a significant contribution towards this direction by introducing an high-level specification language with important features as channel-over-channel passing, secret channels, and deadlock-freedom, that aims at providing for correctness-by-construction, and is supported by a fully automated tool [2] that infers the types of well-behaved specifications, and generates executable Go code. Our approach is unique, and has no burden for the developer: the only task of the programmer is to describe the concurrent protocol in a specification language built upon few intuitive constructs, and with no decorations, everything else is automated. Nevertheless, we accept important challenges: statically checking that the scope of a channel specified as secret is not enlarged at runtime, even when the program runs in parallel with well-behaved interacting contexts; statically checking deadlock-freedom on linear channels that can be used exactly once in input and once in output; fully automated generation of executable Go code that mimics non-deterministic synchronizations à la pi calculus while enforcing race-freedom.

In the specification language, we consider a construct to declare *secret* channels, where secrecy is interpreted as the preservation of the following invariant of the runtime system: the scope of a secret channel cannot be enlarged, even when the program is executed in parallel with well-behaved interacting *contexts*; that is, secret means *static*. This leads to a powerful but yet simple approach, where the secrecy control does not rely on advanced features as, for instance,

* This work is partially supported by the Tezos fondation through the project FACTOR and, by national funds, through FCT-Fundacão para a Ciência e a Tecnologia, I.P, in the context of NOVA LINCS through the project UID/CEC/04516/2019.

non-interference [17,26], role-based- access control [27,9], or cryptographic analysis [5,3,4,32], but on a static analysis of the high-level language that is based on types that are inferred in a fully automated fashion.

Related Work We did not find work similar to our construction; in particular, the ideas behind the implementation in Go are original. We refer to [15,14] for previous work of the author on secret channels.

Static channels and boundaries in process calculi have been investigate since the origins of this research area [30], and more recently in [6,28,7], among the others. The work in [6] is the closest to 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. Decisively, programmers must declare which is the group type of the payload. In contrast, our analysis is fully-automated, does not require type annotations, and is *contextual*: processes are allowed to sent a channel outside the group if the context does not have read access to the channel.

To the best of our knowledge, most interpreters for distributed calculi supporting channel-over-channel passing do not rely on channel-based mechanisms at the target language level. The implementation of languages inspired by the pi calculus has been pioneered by [31,25,29]. Central to this line of work is the notion of Turner machine [31], which allows to simulate non-determinism and concurrent executions in uniprocessors by interleaving the execution of processes; this is de-facto mechanism for concurrency in most process calculi implementations (e.g. [13]). Previous attempts to develop calculi-inspired languages with native support for channel-over-channel passing include JoCaml [12], where mobility is now discontinued [22] for the sake of compatibility with OCaml.

Recently, a behavioural static analysys of Go programs based on multiparty session types (*MPST*, [18]) have been presented in [19,20]. The approach followed in this line of work consists in analysing existing Go programs, in order to ensure stronger properties at compile-time, e.g. deadlock-freedom; this is done by extracting the program's behaviour as a global MPST. None of these works, however, support channel-over-channel passing. In [8], the authors present a framework to translate distributed MPST written in the Scribble protocol language into a Go API. MPST types are mapped into Go types and methods. Existing Go clients can use the API to ensure a form of practical safety: the API dynamically generates errors (e.g. *panic*) when a Go program tries to break the protocol's safety, i.e. its linearity. On contrast, we generate both the server and client's code in a fully automated way, and we do not rely on dynamic analysis, because the generated Go code aims at being correct-by-construction, that is safety is tackled statically by means of type inference of pi calculus channels.

2 Example: designing a secret chat protocol

To illustrate our construction, we consider the example of a messaging application with support for secret chat, that are chats that cannot be *forwarded*. We believe that in a concurrent setting this feature can be naturally interpreted as a secret channel, one that cannot be forwarded outside its designed scope. The

$$
\begin{aligned}
\mathit{Alice} &\triangleq \text{alice?(chat).(group?(friend).friend!chat} \mid \text{chat!}\texttt{helloAlice}\text{)} \\
\mathit{Bob} &\triangleq \text{bob?(chat).chat!}\texttt{helloBob} \mid \text{bot?}(m)\text{.getMsg!}m \mid \\
&\qquad \text{getRandom?}(r).r\text{?(stranger).getMsg?}(m)\text{.stranger!}\ m \\
\mathit{Carl} &\triangleq \text{carl?(chat).chat!}\texttt{helloCarl} \\
\mathit{Board} &\triangleq \text{board?(secret).}S\text{(secret)} \\
\mathit{ChatServer} &\triangleq \text{setup?(user).} \\
&\qquad [\textbf{hide}\,\text{chat}][\text{user!chat} \mid \text{board!chat} \mid \mathit{Alice} \mid \mathit{Bob} \mid \mathit{Carl} \mid \mathit{Board}] \\
\mathit{Chat} &\triangleq \text{setup!alice.(bot!group} \mid \text{group!bob} \mid \text{group!carl)} \\
P &\triangleq \langle \text{board, getMsg, setup} \rangle (\textbf{new}\ \text{bot, group, alice, bob, carl})(\mathit{Chat} \mid \mathit{ChatServer}) \\
S(x) &\overset{\text{rec}}{=} x\text{?(message).(}\textbf{print} :: \text{message} \mid S(x)\text{)}
\end{aligned}
$$

Fig. 1: Secret chat protocol in the LSpi specification language

scenario may be the following. *Alice, Bob*, and *Carl* create a private group in the app in order to chat among themselves. After the creation phase, the app installs a *chat server* waiting for a request of a user of the group to *setup* a chat: once the request has been pick up, the server creates a *hidden chat* channel with static scope including Alice, Bob, Carl, and the chat board, and send it to the user. Finally, the user forwards the chat channel to her friends, and from now on the group can use the channel to exchange messages on the board. In order to do not contain errors, the protocol must maintain the invariant that the scope of a secret channel cannot be enlarged: we thus need to ensure that the design of our programs satisfies properties of this form, and that these properties are preserved when programs are deployed in well-behaved contexts.

Figure 1 presents a specification of the protocol in the *LSpi* language, a programming-oriented variant of the secret pi calculus [15]; incidentally, we note that the specification adds a feature to Bob, that is to communicate with random strangers (off the board). The LSpi language follows a minimalistic approach and presents a small number of primitives for sending and receiving values over channels (noted `!` and `?`, respectively), where values are channels or `base values`, restricting and hiding channels (new and hide, respectively), recursion (noted $\overset{\text{rec}}{=}$), parallel composition (noted $\mid$), printing, declaration of linear channels that must be used exactly once in input, and once in output (noted $\langle\cdot\rangle$), and let-process definitions (noted $\triangleq$). The aim of the language is to provide for a correct-by-construction specification of concurrent programs and security protocols.

The three main protection ingredients in Figure 1 are *hiding*, *restriction*, and *linearity*: hiding and linearity represent semantic protection, while restriction is syntactic protection. Restriction is the core mechanism of pi calculus [23], and naturally corresponds to local scope in programming languages: most channels are declared as restricted in P in order to avoid direct access from the context. Channel protection by linearity (e.g. *board*) offers (at least) the same guarantee

by relying on types: contexts read/write accessing a linear channel will be ruled out. Declaring *chat* with the *hide* constructor aims at forbidding the enlarging of the scope of the channel, in *all well-behaved computations*.

One question that we face is the following: does the protocol's specification P in Figure 1 contain programming errors that can compromise the protocol logic? For instance, can we statically detect if P, once immersed in a well-behaved context and executed in a channel-based runtime system, leaks secret channels? This is a fundamental prerequisite to the generation and execution of code, since our goal is to deploy correct-by-construction concurrent programs that at runtime preserve the high-level specification properties while communicating through message-passing without instrumentation (e.g. monitoring the exchanges). Interestingly, the protocol contains a subtle vulnerability that may cause a security breach; the attack, detected by the *GoPi* tool [2], is outlined below.

Our static analysis relies on types, and detects attempts to open the scope of an hidden channel at compile-time. Protocol P in Figure 1 *does not type check* since a well-typed context interacting with P can open the scope of channel *chat*. While *Bob* legitimately relies on a *bot* channel to produce messages for random strangers, the *Chat* component erroneously (or maliciously) sends the *group* channel over *bot*. An attacker may use the exploit to receive channel *group* from P, and in turn to send a *fake* friend on *group*: the *fake* channel can be non-deterministically pick up by *Alice*, who in turn will send channel *chat* over *fake*, thus disclosing its secrecy. To *patch* the specification, we remove the thread in red from Figure 1; the resulting process is accepted by GoPi, which automatically generates and runs the process' Go code: the output of an execution is below, where randomization allowed Carl to join the chat. Note that throwing a timeout (on waiting on all input channels) causes the Go runtime to detect a deadlock; this is fine, since all blocked channels in the goroutines (i.e. the parallel threads) are unrestricted, while our static analysis prevent deadlocks on linear channels.

```
Retrieved alice from setup. Retrieved chat from board. Retrieved chat from alice.
Retrieved carl from group. Retrieved chat from carl. Retrieved helloAlice from chat.
Print helloAlice. Retrieved helloCarl from chat. Print helloCarl.
TIMEOUT. Fatal error: all goroutines are asleep - deadlock! Exit status 2
```

3 Discussion

This abstract introduces two main techniques to assist the concurrent software developement process: (a) an high-level specification *language* that supports primitives for secrecy and linearity to design *correct-by-construction* protocols; (b) a fully automated *tool* that (1) infers the type of channels of programs having a good behaviour *without* relying on *decorations* of the source code, and (2) generates executable Go code featuring *channel-over-channel* passing. The techniques are framed in an open and ongoing project that aims at developing and maintaining a compiler for a language with built-in support for mobility, security, resource-awareness, and deadlock-resolution. The development of the

tool has reached a stable phase; we release the code in GitHub [2]. We plan several improvements, among which the most interesting are: support for channel subtyping [24], deployment of mechanized proofs of correctness [1,10], and embedding program transformation techniques to statically resolve deadlocks [16].

References

1. Coq 8.9.0 – Reference Manual, `https://coq.inria.fr/distrib/current/refman`, accessed May 2019
2. The GoPi compiler, `https://github.com/marcogiunti/gopi`
3. Abadi, M., Blanchet, B., Fournet, C.: The applied pi calculus: Mobile values, new names, and secure communication. J. ACM **65**(1), 1:1–1:41 (2018)
4. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: POPL. pp. 90–101. ACM (2009)
5. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: CSFW. pp. 82–96. IEEE (2001)
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. Ferraiolo, D., Kuhn, D.R., Chandramouli, R.: Role-based access control. Artech House (2003)
10. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: ESOP. LNCS, vol. 7792, pp. 125–128. Springer (2013)
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 (2010)
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. Franco, J., Vasconcelos, V.T.: A concurrent programming language with refined session types. In: SEFM 2013. Lecture Notes in Computer Science, vol. 8368, pp. 15–28. Springer (2014). https://doi.org/10.1007/978-3-319-05032-4_2
14. Giunti, M.: Static semantics of secret channel abstractions. In: NORDSEC. LNCS, vol. 8788, pp. 165–180. Springer (2014)
15. Giunti, M., Palamidessi, C., Valencia, F.D.: Hide and New in the Pi-Calculus. In: EXPRESS/SOS. EPTCS, vol. 89, pp. 65–79 (2012)
16. Giunti, M., Ravara, A.: Towards static deadlock resolution in the π-calculus. In: TGC 2013. LNCS, vol. 8358, pp. 136–158. Springer (2014)
17. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy. pp. 11–20. IEEE (1982)
18. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016)
19. 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)

20. 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
21. 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 (2008). https://doi.org/10.1145/1346281.1346323
22. Mandel, L., Maranget, L.: The JoCaml language, `http://jocaml.inria.fr/doc`, Release 4.01, March 14, 2014
23. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Information and Computation **100**(1), 1–77 (1992)
24. Pierce, B.C., Sangiorgi, D.: Typing and subtyping for mobile processes. Mathematical Structures in Computer Science **6**(5), 409–453 (1996)
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. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
27. Sandhu, R.S.: Role-based access control. In: Advances in computers, vol. 46, pp. 237–286. Elsevier (1998)
28. Sewell, P., Vitek, J.: Secure composition of untrusted code: Box pi, wrappers, and causality. J. Comp. Sec. **11**(2), 135–188 (2003)
29. 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
30. 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
31. Turner, D.N.: The Polymorphic Pi-calculus:Theory and Implementation. Ph.D. thesis, University of Edinburgh (1995)
32. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: CCS. pp. 1789–1806. ACM (2017)