tcpst todo

todo

February 7, 2023

1 Introduction

2 Background

2.1 Internet Protocol Standardisation

2.2 Session Types

Session types are a behavioural typing discipline that were first introduced by Honda [1] as a typed formalism for concurrency. Session types represent a series of actions that each have a type and a direction of data exchange. Importantly, session types dictate the protocols that the channels must use to communicate. Consider an example where a client **C** sends two strings to a server **S** representing the username and password of the user needed for authentication and expects a boolean, confirming if the user is to be authenticated and then terminates:

$$\mathbf{C} \;=\; \texttt{!String.!String.?Boolean.end}$$

This session type describes that **C**, sends (`!`) two strings and then receives (`?`) a boolean and then terminates (`end`). From the perspective of the server, we have the *dual* session type:

$$\mathbf{S} \;=\; \texttt{?String.?String.!Boolean.end}$$

We can see that the server cannot proceed in a way that would be incompatible with the client. This is guaranteed by the notion of *duality*, which means that the sending and receiving sides have consistent typing. For example, when **C** sends a string, **S** expects to receive a string.

We can further understand the importance of duality in a slightly more complex example. Consider now that client **C** can first choose an authentication method, credentials or token, provide the chosen method and terminate, while the server **S** will offer two authentication methods and then proceed accordingly depending on the method chosen by the client and terminate:

$$\begin{aligned}\mathbf{C} \;=\; \oplus\,\{\; &\mathit{cred} : \texttt{!String.!String.?Boolean.end},\\ &\mathit{token} : \texttt{!String.?Boolean.end}\;\}\end{aligned}$$

$$\begin{aligned}\mathbf{S} \;=\; \&\{\; & \mathit{cred} : \texttt{?String.?String.!Boolean.end}, \\ & \mathit{token} : \texttt{?String.!Boolean.end} \;\}\end{aligned}$$

We now have a session type where the client chooses ($\oplus$) one of the possible branches that the server offers (&), which is represented via the constructs of *selection* and *branching*. Due to the duality concept, the client can't select an authentication method not provided by the server. Hence, we can see how duality ensures *communication safety* by ensuring the compatibility of session types.

Having defined what is meant by session types, it is important to note that the examples provided used *binary session types*; that is, all communication is handled on one channel by two participants. To model more complex protocols where two or more participants interact, *multiparty session types* [2] (MPST) can be used. In classic MPST theory, we define a *global type* that describes the communication of all participants and then *project* this into *local session types*, which represent the communication from the viewpoint of each participant. While the classic MPST theory gives us a global view of communication, it is also overly restrictive in the processes that can be typed and caused flaws in subject reduction. In this work we use the more general multiparty session type theory (LM) presented by Scalas and Yoshida [6]. Unlike the classic theory, LM does not base itself on the concepts of global types and duality. This decoupling from binary session type theory makes the LM calculus more general.

The LM view of multiparty session types is based on the concept of a weak typing context *safety invariant*. A LM type system is parametric on the safety invariant, that is, by changing the parameter we can control what processes are typed based on the properties that we want. The weakest of the invariants is simply safety. The safe invariant ensures that we have corresponding branches for selections, that recursive unfoldings are safe and that reductions are also safe if the context being reduced is safe. In order to be typeable under any LM type system a protocol must be at least safe. The authors present several typing context properties centered around deadlock freedom and liveness. Consider an example of a protocol in LM:

$$\begin{aligned}\Gamma \;=\;\; & s[a] : \texttt{b}\&\texttt{l1(int).c}\oplus\texttt{l3(int).end}, \\ & s[b] : \texttt{c}\&\texttt{l2(int).a}\oplus\texttt{l1(int)}, \\ & s[c] : \texttt{a}\&\texttt{l3(int).b}\oplus\texttt{l2(int).end}\end{aligned}$$

In this context we have a communication between three participants: a, b and c. We use $l_{i\in I}$ to denote labels and the type between brackets denotes the message type. Unlike the previous binary session type example we use $\oplus$ to denote sending and & to denote receiving. This is simply because sending is a case of selection and receving is a case of branching where each only have one option. This syntax choice can similarly be made in binary session types.

If we instantiate Γ with $\phi = \mathit{safe}$, and derive the needed typing judgement then we will see that this protocol type checks. However, by instantiating $\phi = \mathit{deadlock\text{-}free}$, and deriving the typing judgement, we will see that the protocol is not well typed. To emphasise, we have made a *static* typing judgement to derive the *runtime* behaviour of a process that abides by Γ. This distinction is crucial as the authors show that the typing context properties are *decidable*, while the rutime process properties are not. Hence, if we implement a protocol that follows the session typed context Γ, we know that it will not be deadlock free. A further advantage of this flexible theory is its ability to be integrated into various systems. Specifically, the authors express ϕ as a formula in the μ-calulus and develop the *mpstk* tool that translates session typed contexts to the *mcrl2* [**todo**] model checker to verify various safety invariants.

3 Formalising Internet Protocols Using Session Types

Based on this, we leverage session type theory to first ensure that an implementation of a protocol follows its corresponding RFC and then to check the kind of runtime properties the protocol has. We do this through the following contributions:

- An IR to be used in the RFC to extract a session type of the communication state machine of the protocol.
- A tool to generate the protocol's session type based on the RFC in the Rust programming language, thus ensuring that the implementation has to follow the described protocol.
- A supporting implementation of multiparty session types in the Rust programming language that is generated alongside the protocol's session type.
- A tool to generate the corresponding *mpstk* typing context, giving us the ability to check for various properties such as deadlock freedom and liveness.

We have chosen to focus on TCP as a demonstration of the tooling chain because TODO: .

TODO: IR TODO: impl and generation - explain how session types help us in Rust (example where something has to follow the session type) TODO: explain mpstk, syntax, model checking, etc

4 Discussion On Limitations Of The Theory

TODO:

- We can't express notion of throughput
- Generally unclear how we should describe "how much data is being sent" - is this the number of send calls or does the message need to be extended to express this?
- There are calculi that do timed and timeouts but these do not have the tooling
- Implementation limitations - heterogeneous channels (whats the overhead?), no variadic generics in Rust, will probably encounter other

5 Future Directions

TODO:

- other properties we may want to parametrise?

- model checker reporting - where is property violated?
- considering other protocols to exploit other theory of STs? - context free, multiplicities?

$$
\begin{array}{rcl}
c & ::= & s[r] : S \\
S & ::= & r \oplus \{l_i(P_i).S_i\}_{i \in I} \;\mid\; r \,\&\, \{l_i(P_i).S_i\}_{i \in I} \;\mid\; \mu(t).(S) \;\mid\; t \;\mid\; end \\
P & ::= & \{TcpPayloadTypes\}
\end{array}
$$

Figure 1: The syntax of session type contexts used to create the protocol models in *mpstk*.

TODO: explain why TCP is multiparty, check: is this the first ST model that considers TCP to be multiparty?

The syntax for the session type language that the protocols are written in is presented in Figure 1. We do not omit this as the syntax of the language used for *mpstk* differs to that of LM. Additionally, we only need to consider a simplified subset of the calculus. Note that P can not be an instance of S, that is, we do not consider sending sessions across channels. Although this is standard to the theory of π-calculus, the presented network protocols would not benefit from such a mechanism. Hence, the payload types are restricted to the defined TCP payload types.

References

[1] Kohei Honda. "Types for dyadic interaction". In: *CONCUR'93*. Springer, 1993, pp. 509–523. ISBN: 978-3-540-47968-0. DOI: 10.1007/3-540-57208-2_35.

[2] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types". In: *Journal of the ACM* 63.1 (Mar. 2016), 9:1–9:67. ISSN: 0004-5411. DOI: 10.1145/2827695. URL: https://doi.org/10.1145/2827695 (visited on 11/28/2021).

[3] Kenneth L. McMillan and Lenore D. Zuck. "Compositional Testing of Internet Protocols". In: 2019. DOI: 10.1109/SecDev.2019.00031.

[4] Stephen McQuistin et al. "Investigating Automatic Code Generation for Network Packet Parsing". In: 2021. DOI: 10.23919/IFIPNetworking52078.2021.9472829.

[5] Stephen McQuistin et al. "Parsing Protocol Standards to Parse Standard Protocols". In: 2020. DOI: 10.1145/3404868.3406671. URL: https://doi.org/10.1145/3404868.3406671.

[6] Alceste Scalas and Nobuko Yoshida. "Less is More: Multiparty Session Types Revisited". In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290343. URL: https://doi.org/10.1145/3290343.