EPTCS 401

Proceedings of the

15th Workshop on

Programming Language Approaches to Concurrency and Communication-cEntric Software

Luxembourg City, Luxembourg, 6th April 2024

Edited by: Diana Costa and Raymond Hu

Published: 6th April 2024
DOI: 10.4204/EPTCS.401
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Diana Costa
Universidade de Lisboa, PT
dfdcosta@ciencias.ulisboa.pt

Raymond Hu
Queen Mary University of London, UK
r.hu@qmul.ac.uk

This volume contains the proceedings of PLACES 2024, the 15th edition of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software. The workshop is scheduled to take place in Luxembourg City, Luxembourg on April 6, 2024, as a satellite event of ETAPS, the European Joint Conferences on Theory and Practice of Software.

PLACES offers a forum for exchanging new ideas on how to address the challenges of concurrent and distributed programming, and how to improve the foundations of modern and future computer applications. PLACES welcomes researchers from various fields, and its topics include the design of new programming languages, models for concurrent and distributed systems, type systems, program verification, and applications in various areas (e.g. microservices, sensor networks, blockchains, event processing, business process management).

The Programme Committee of PLACES 2024 consisted of:

- Laura Bocchi, University of Kent, UK
- Cinzia Di Giusto, Université Côte d'Azur, CNRS, FR
- Juliana Franco, DeepMind, UK
- Lorenzo Gheri, University of Liverpool, UK
- Ping Hou, University of Oxford, UK
- Cosimo Laneve, University of Bologna, IT
- Matthew Alan Le Brun, University of Glasgow, UK
- Rumyana Neykova, Brunel University, UK
- Kirstin Peters, Universität Augsburg, DE
- Diogo Poças, LASIGE, Universidade de Lisboa, PT
- Shoji Yuen, Nagoya University, JP

After a thorough reviewing process, the Programme Committee has accepted five research papers (out of six original submissions): such papers are published in this volume. The Programme Committee has also accepted three talk proposals on preliminary or already-published work: the titles and abstracts of such talks are also listed in this volume. Each submission (research paper or talk proposal) was reviewed and discussed by three Programme Committee members on the EasyChair platform.

We would like to thank everyone who contributed to PLACES 2024: this includes the authors of submissions, the Programme Committee members, Mariangiola Dezani-Ciancaglini and Peter Müller for their keynote talks, the ETAPS 2024 organisers, the EasyChair and EPTCS administrators. Finally, special thanks to the Steering Committee of PLACES - Simon Gay, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida.

3 April 2024
Diana Costa and Raymond Hu

D. Costa, R. Hu (Eds.): Programming Language Approaches to Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. ii–ii, doi:10.4204/EPTCS.401.0

Keynote Talk: Simple MultiParty Sessions

Mariangiola Dezani-Ciancaglini
Università di Torino, IT
dezani@di.unito.it

Simple MultiParty Sessions (SMPS) do not have channels, session initiators and the distinction between compile time and run time syntax. They are based only on participant names and input/output processes. They are equipped with global types without requiring projections and local types. In this presentation we will discuss pros and cons of SMPS. On the good side SMPS allow to easily describe internal delegation, partial typing and session composition. On the bad side SMPS are less expressive than standard MultiParty Sessions. In particular interleaved sessions with crossing delegations cannot be represented.

D. Costa, R. Hu (Eds.): Programming Language Approaches to Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. iii–iii, doi:10.4204/EPTCS.401.0.1

Keynote Talk: Verified Secure Routing

Peter Müller
ETH Zurich, CH
peter.mueller@inf.ethz.ch

SCION is a new Internet architecture that addresses many of the security vulnerabilities of today's Internet. Its clean-slate design provides, among other properties, route control, failure isolation, and multi-path communication. The verifiedSCION project is an effort to formally verify the correctness and security of SCION. It aims to provide strong guarantees for the entire architecture, from the protocol design to its concrete implementation. The project uses stepwise refinement to prove that the protocol withstands increasingly strong attackers. The refinement proofs assume that all network components such as routers satisfy their specifications. This property is then verified separately using deductive program verification in separation logic. This talk will give an overview of the verifiedSCION project and explain, in particular, how we verify code-level properties such as memory safety, I/O behavior, and information flow security.

D. Costa, R. Hu (Eds.): Programming Language Approaches to
Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. iv–iv, doi:10.4204/EPTCS.401.0.2

Presentations of Preliminary or Already-Published Work

Diana Costa
Universidade de Lisboa, PT
dfdcosta@ciencias.ulisboa.pt

Raymond Hu
Queen Mary University of London, UK
r.hu@qmul.ac.uk

PLACES 2024 welcomed the submissions of talk proposals (describing preliminary or already-published work) that could spark interesting discussion during the workshop. This is the list of all accepted talk proposals.

Multiparty Session Type Projection and Subtyping with Automata

Elaine Li – New York University, USA
Felix Stutz – MPI-SWS, DE
Thomas Wies – New York University, USA

Multiparty session types (MSTs) are a type-based approach to verifying communication protocols. Central to MSTs is a projection operator: a partial function that maps protocols represented as global types to correct-by-construction implementations for each participant, represented as a communicating state machine (CSM). Existing projection operators are syntactic in nature, and trade efficiency for completeness. In the first part of the talk, I will present the first projection operator that is sound and complete. I will highlight the automata-theoretic nature of our projection operator, which separates synthesis from checking implementability, and can be computed in PSPACE.

While our projection operator always computes a candidate implementation if one exists, it may not always compute the best candidate. In the second part of the talk, I motivate three variations of the subtyping problem for MSTs. I highlight differences between the problems in terms of complexity, compositionality and context-dependence. I show how our previous solution to implementability gives rise to solutions to subtyping problems for MSTs, with unrestricted CSMs as implementation model.

Hapi – Implementing the asynchronous π-calculus with multiparty session types

Lasse Nielsen – Denmark
Nobuko Yoshida – University of Oxford, UK

We introduce the language hapi, implementing the asynchronous π-calculus with multiparty session types. Hapi offers type verification and compilation of the calculus into C++. The direct syntax representation of the π-calculus with multiparty session types in hapi enables a concise declaration of message-passing processes. Hapi's type verification guarantees a linear usage of channels, type safety, no race-conditions and progress properties of hapi processes.

Multiparty Session Type Inference for a Rust DSL

Walid Nawfal Sabihi, Martin Vassor and Nobuko Yoshida – University of Oxford, UK

D. Costa, R. Hu (Eds.): Programming Language Approaches to Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. v–vi, doi:10.4204/EPTCS.401.0.3

Multiparty Session Types (MPSTs) have traditionally been used in a top-down manner, starting from a global type which provides a specification for the system, that is then projected onto local types that are used to type-check each participant in the protocol. This approach is challenging to adopt for existing systems that need to gradually integrate MPSTs but have not yet defined a global type that fully covers their behaviour.

In this talk, we propose an inference system for a subset of Rust programs, synthesising anonymous local session types. We also discuss a participant allocation algorithm that generates a global session type from a set of anonymous local session types, with guarantees for correctness and completeness.

Linear Contextual Metaprogramming and Session Types

Pedro Ângelo
LIACC, Faculdade de Ciências da Universidade do Porto, Portugal
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
pjangelo@ciencias.ulisboa.pt

Atsushi Igarashi
Kyoto University, Kyoto, Japan
igarashi@kuis.kyoto-u.ac.jp

Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
vmvasconcelos@ciencias.ulisboa.pt

We explore the integration of metaprogramming in a call-by-value linear lambda-calculus and sketch its extension to a session type system. We build on a model of contextual modal type theory with multi-level contexts, where contextual values, closing arbitrary terms over a series of variables, may then be boxed and transmitted in messages. Once received, one such value may then be unboxed (with a let-box construct) and locally applied before being run. We present a series of examples where servers prepare and ship code on demand via session typed messages.

1 Introduction

Metaprogramming manipulates code in order to generate and evaluate code at runtime, allowing in particular to explore the availability of certain arguments to functions in order to save computational effort. In this paper we are interested in programming languages where the code produced is typed by construction and where code may refer to a context providing types for the free variables, commonly known as contextual typing [8, 11, 12, 13]. On an orthogonal axis, session types have been advocated as a means to discipline concurrent computations, by accurately describing protocols for the channels used to exchange messages between processes [1, 5, 6, 7, 15, 16, 17].

The integration of session types with metaprogramming allows to setup code-producing servers that run in parallel with the rest of the program and provide code on demand, exchanged on typed channels. Linearity is central to session types, but current metaprogramming models lack support for such a feature. We extend a simple model of contextual modal type theory (with monomorphic contexts) with support for linear types, to obtain a call-by-value linear lambda calculus with multi-level contexts. We then sketch how to extend this language with support for concurrency and session types.

Our development is based on Davies and Pfenning [4], where we use a box modality to distinguish generated code. We further allow code to refer to variables in a context, described by contextual types, along the lines of Nanevski et al. [13]. Mœbius [8] further adds to modal contextual type theory the provision for pattern matching on code, for generating polymorphic code, and for generating code that depends on other code fragments. We forgo the first two directions, and base our development on the last. We propose a multilevel contextual modal *linear* lambda calculus, where in particular the composition of code fragments avoids creating extraneous administrative redexes due to boxing and unboxing.

An alternative starting point would have been the Fitch- or Kripke-style formulation, providing for the Lisp quote/unquote, where typing contexts are viewed as stacks modeling the different stages of computation [11]. It seemed to us that the let-box approach would simplify the extension to the linear setting, and then to session types.

D. Costa, R. Hu (Eds.): Programming Language Approaches to Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. 1–10, doi:10.4204/EPTCS.401.1

To motivate the problem let us start with the issue of generating code to send a fixed number of integers on a stream. The type of streams, as seen from the side of processes writing on the stream is as follows.

```
type Stream = ⊕{More: !Int.Stream, Done: Close}
```

The writer chooses between selecting More values or selecting Done. In the former case, the writer sends an integer value and "goes back to the beginning"; in the latter case the writer must close the channel. Function sendFives accepts an integer value and returns a code fragment that requires a Stream and, when executed, produces a unit value, written [Stream ⊢ Unit]. We proceed by pattern-matching on the parameter.

```
sendFives : Int → [Stream ⊢ Unit]
sendFives 0 = box (y. close (select Done y))
sendFives n = let box u = sendFives (n − 1)
              in box (x. u[send 5 (select More x)])
```

When all values have been sent in the stream (when n is 0), all it remains is to **select** Done and then close the channel. The **box** expression generates code under a variable environment (an evaluation context), in this case containing variable y alone, the channel endpoint. Otherwise, we recursively compute code to send n−1 values, unbox it storing the result in u, and then prepare code to send the n-th value. Expression u[**send** 5 (**select** More x)] (an applied variable) applies expression **send** 5 (**select** More x) of type Stream to u (a contextual value of type (Stream ⊢ **Unit**) to obtain an expression of type **Unit**. If u is the contextual value y. **close** (**select** Done y), then the explicit substitution evaluates to **close** (**select** Done **send** 5 (**select** More x)).

We may now compute and run code to send a fixed number of integer values.

```
send4Fives : Stream → Unit
send4Fives c = let box u = sendFives 4 in u[c]
```

Expression sendFives 4 is a boxed code fragment of type [Stream ⊢ **Unit**]. Then, u is an (unboxed) code fragment of contextual type (Stream ⊢ **Unit**). We provide the code fragment with an explicit substitution [c]. The whole **let** expression then amounts to running the code

```
close (select Done (send 5 (select More (...send 5 (select More c)...))))
```

without calling function sendFives or using recursion in any other form.

The next example transmits code on channels. Imagine a server preparing code on behalf of clients. The server uses a channel to interact with its clients: it first receives a number n, then replies with code to send n fives, and finally waits for the channel to be closed. The type of the communication channel is as follows.

```
type Builder = ?Int.![Stream ⊢ Unit].Wait
```

The server receives n on a given channel and computes the code using a call to sendFives:

```
serveFives : Builder → Unit
serveFives c =
  let (n,c) = receive c in wait (send (sendFives n) c)
```

On the other end of the channel sits a client: it sends a number (4 in this case), receives the code (of type [Stream ⊢ **Unit**]), closes the channel and evaluates the code received.

```
sendFives' : Dual Builder → Stream → Unit
sendFives' c d =
```

```
  let (code,c) = receive (send 4 c) in close c ;
  let box u = code in u[d]
```

The **Dual** operator on session types provides a view of the other end of the channel. In this case, **Dual** Builder is the type !**Int**.?[Stream ⊢ **Unit**].**Close**, where ! is turned into ? and **Wait** is turned into **Close** (and conversely in both cases). Notice that code is a boxed code fragment of type [Stream ⊢ **Unit**], hence u is the corresponding code fragment (of type (Stream ⊢ **Unit**)) and u[d] runs the code on channel d.

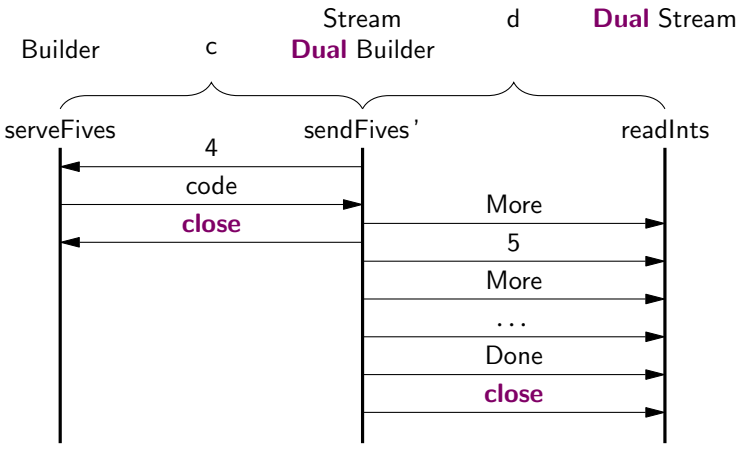
To complete the example we need a function for reading streams, a consumer of type **Dual** Stream → **Unit**. Function readInts reads and discards all integer values on the stream and then waits for the stream to be closed.

```
readInts : Dual Stream → Unit
readInts (Done c) = wait c
readInts (More c) = let (_, c) = receive c in readInts c
```

Finally, the main thread forks two threads—one running serveFives, the other to collect the integer values (readInts)—and continues with sendFives'. We take advantage of a primitive function, forkWith that expects a suspended computation (a thunk), creates a new channel, forks the thunk on one end of the channel, and returns the other end of the channel for further interaction.

```
main : Unit
main =
  let c = forkWith (λ_→serveFives) in -- c : Dual Builder
  let d = forkWith (λ_→readInts) in   -- d : Stream
  sendFives' c d
```

The interaction among the three processes is depicted as follows



where the code produced by function serveFives and transmitted to sendFives' is

```
box (close (select Done (send 5 (select More (...(select More c)...)))))
```

In the rest of the paper we develop our system. In section 2 we introduce the call-by-value linear lambda calculus with multi-level contexts and in section 3 we sketch the extensions required to type and run the examples in this section. We conclude in section 4.

2 Linear staged metaprogramming

This section introduces the call-by-value linear lambda calculus with multi-level contexts.

Syntax Our language is defined over the syntactic categories of set of variables, x, y, z. We write $\overline{X}$ for a sequence of objects $X_1 \cdots X_n$ with $n \geq 0$. The empty sequence (when $n = 0$) is denoted by ε.

Contextual type	τ	::=	$(\overline{\tau} \vdash T)$
Type	T, U	::=	$* \mid T \to T \mid \Box\tau$
Contextual value	ρ, σ	::=	$\overline{x}.M$
Value	v	::=	$* \mid \lambda x.M \mid \mathsf{box}\,\sigma$
Term	M, N	::=	$v \mid x[\overline{\sigma}] \mid \mathsf{let}\ * = M\ \mathsf{in}\ M \mid MM \mid \mathsf{let\ box}\ x = M\ \mathsf{in}\ M$
Level	k, m, n	$\in$	$\mathbb{N}$
Typing context	Γ, Δ	::=	$\varepsilon \mid \Gamma, x :^n \tau$

Types include the unit ($*$) linear type, the linear arrow type $T \to U$, and the linear boxed contextual type $\Box\tau$. A contextual type τ of the form $(\overline{\tau} \vdash T)$ represents code of type T, parameterized on variables typed by a list of contextual types $\overline{\tau}$, called *contexts* [8, 11].

To objects of the form $\overline{x}.M$ we call *contextual values*. They denote code fragments M parameterized by the variables in sequence $\overline{x}$. The values in the language include $*$ (introducing type $*$), lambda abstraction (introducing the arrow type), and the box term (introducing the contextual modal type $\Box\tau$). Terms include values, contextual term variables $x[\overline{\sigma}]$ applying contextual values $\overline{\sigma}$ to the code fragment described by x, the let $*$ (eliminating $*$), lambda application (eliminating the arrow type), and the let box term (eliminating the contextual modal type $\Box\tau$).

When $\overline{x}$ is the empty sequence we sometimes write M in place of the contextual value $\varepsilon.M$. Similarly, when $\overline{\sigma}$ is the empty sequence we sometimes write x instead of the contextual term variable $x[\varepsilon]$. Furthermore, in examples we write Unit in place of $*$, and $[\overline{\tau} \vdash T]$ in place of $\Box(\overline{\tau} \vdash T)$.

The bindings in the language are the following. Variable x is bound in M (but not in N) in terms $\lambda x.M$ and $\mathsf{let\ box}\ x = N\ \mathsf{in}\ M$. The set of bound and free variables in terms (free M) are defined accordingly. We follow the variable convention whereby terms that differ only in the names of the bound variables are interchangeable in all contexts [14].

Contexts for local and outer variables Typing contexts bind contextual types to variables; we annotate each entry with its level n. We assume that contexts contain no duplicate variables and write Γ, Δ for the context containing the entries in both Γ and Δ, provided they feature disjoint sets of variables.

We introduce two predicates on typing contexts. If $\Gamma = x_1 :^{k_1} \tau_1, \ldots, x_m :^{k_m} \tau_m$, then $\Gamma^{<n}$ holds when n is greater than all the levels of the bindings in Γ (that is, $k_1, \ldots, k_m$), and $\Gamma^{\geq n}$ holds when n is smaller or equal than all the levels in Γ. More precisely,

$$\Gamma^{<n} \text{ holds when } \max(0, k_1, \ldots k_m) < n \qquad \Gamma^{\geq n} \text{ holds when } m = 0 \text{ or } \min(k_1, \ldots k_m) \geq n$$

Intuitively, $\Gamma^{<n}$ denotes the local variables in a code fragment of level n, whereas $\Gamma^{\geq n}$ denotes the outer variables. Notice that there is no context Γ for which $\Gamma^{<0}$ (not even the empty context), meaning that code fragments start at level 1.

Substitution Substitution is that of the linear lambda calculus except for the fact that we use applied modal variables $x[\overline{\sigma}]$ rather than ordinary variables x. We denote by $[\sigma/x]M$ the term obtained by replac-

ing variable x by the contextual value σ in term M. We detail some illustrative cases.

$$[\overline{z}.M/x](x[\overline{\rho}]) = [\overline{\rho}/\overline{z}]M$$

$$[\sigma/x](\mathsf{let\ box}\ y = M\ \mathsf{in}\ N) = \begin{cases} \mathsf{let\ box}\ y = [\sigma/x]M\ \mathsf{in}\ N & \text{if } x \in \text{free } M \\ \mathsf{let\ box}\ y = M\ \mathsf{in}\ [\sigma/x]N & \text{if } x \in \text{free } N \end{cases}$$

Substitution on let $*$ and on application is similar to let box. Substitution on the remaining type constructors is an homomorphism; for example $[\sigma/x](\mathsf{box}\,(\overline{y}.M)) = \mathsf{box}\,(\overline{y}.[\sigma/x]M)$. Substitution on applied variables, $[\overline{z}.M/x](y[\overline{\rho}])$, is defined only when x and y coincide and when $\overline{z}$ and $\overline{\rho}$ are sequences of the same length (the variable convention ensures that the variables in $\overline{z}$ are pairwise distinct and not free outside M, hence the substitution of the various variables in $\overline{z}$ can be performed in sequence). Substitution in let box is defined only when $x \in \text{free}\,(MN)$; it is undefined when x is both free in M and in N. The typing system guarantees substitution is defined for all typable processes. Typing judgments are of the form $\Gamma \vdash M : T$ stating that term M has type T under typing context Γ; the rules are introduced below.

Lemma 2.1 (Substitution principle)**.**

$$\frac{\Gamma, x :^n \tau \vdash M : T \qquad \Delta^{\geq n} \vdash \sigma : \tau}{\Gamma, \Delta^{\geq n} \vdash [\sigma/x]M : T}\ \text{SUBS}$$

When the contextual term variable denotes a parameterless code fragment, we recover the conventional substitution principle in linear type systems. If we write $(\varepsilon \vdash T)$ simply as T, and the contextual value $\varepsilon.N$ as N, we have

$$\frac{\Gamma, x :^n (\varepsilon \vdash U) \vdash M : T \qquad \Delta^{\geq n} \vdash \varepsilon.N : (\varepsilon \vdash U)}{\Gamma, \Delta^{\geq n} \vdash [\varepsilon.N/x]M : T} \quad \text{abbreviated to} \quad \frac{\Gamma, x : U \vdash M : T \qquad \Delta \vdash N : U}{\Gamma, \Delta \vdash [N/x]M : T}$$

Evaluation Evaluation on terms is given by the relation $M \longrightarrow N$ defined by the axioms

$$(\lambda x.M)v \longrightarrow [\varepsilon.v/x]M$$
$$\mathsf{let} * = * \ \mathsf{in}\ M \longrightarrow M$$
$$\mathsf{let\ box}\ x = \mathsf{box}\,\sigma\ \mathsf{in}\ M \longrightarrow [\sigma/x]M$$

compounded by the usual congruence rules of the call-by-value λ-calculus. We do not allow reduction inside a box term (a value) as we do not allow reduction under a λ (another value).

Typing contextual terms Contextual terms are closures of the form $\overline{x}.M$, closing term M over a sequence of variables $\overline{x}$. Such a contextual term is given a contextual type $(\overline{\tau} \vdash T)$ when M is of type T under the hypothesis that the variables in $\overline{x}$ are of the types in $\overline{\tau}$.

$$\frac{\Gamma^{\geq n}, \overline{x :^k \tau}^{<n} \vdash M : T}{\Gamma^{\geq n} \vdash \overline{x}.M : (\overline{\tau} \vdash T)}\ \text{CTXT}$$

Intuitively, the free variables of M are split in two groups: local and outer. The local variables, $\overline{x}$, are distinguished in the contextual term $\overline{x}.M$ and typed under context $\overline{x :^k \tau}^{<n}$ where n is an upper bound of the levels k_i assigned to each local variable. The level of each local variable is arbitrary, as long as it is lower than n. Outer variables are typed under context $\Gamma^{\geq n}$. Natural number n thus denotes the level at which the code fragment $\overline{x}.M$ is typed. In general, a contextual term σ can be typed at different levels. All resources (variables) in the context Γ in the conclusion are consumed in the premise; furthermore resources $\overline{x}$ are consumed in the derivation of M (implying that they must be free in M).

Typing variables Variables are of the form $x[\sigma_1 \cdots \sigma_m]$, denoting a contextual term applied to contextual values $\sigma_1 \cdots \sigma_m$. To type each σ_i we need a separate context which we call Γ_i. The type of x must be a contextual type of the form $(\tau_1 \cdots \tau_m \vdash T)$ and each contextual term σ_i must be of type τ_i. We can easily see that all resources in the typing context in the conclusion are consumed: the various Γ_i are consumed in the premises, that for x is consumed at the right of the turnstile. This guarantees that all resources are used and thus that the rule includes no implicit form of weakening.

$$\frac{\overline{\Gamma} \vdash \overline{\sigma} : \overline{\tau}}{\overline{\Gamma}, x :^n (\overline{\tau} \vdash T) \vdash x[\overline{\sigma}] : T}\ \text{VAR}$$

The lengths of all sequences, $\overline{\Gamma}$, $\overline{\sigma}$ and $\overline{\tau}$, must coincide. When the sequences are empty, the rule becomes an axiom. In fact rule VAR is the natural generalization of the axiom in the linear λ-calculus, obtained when writing type $(\varepsilon \vdash T)$ in the context as T, writing applied variable $x[\varepsilon]$ as x, and omitting the level of the variable in the context.

$$\frac{}{x :^n (\varepsilon \vdash T) \vdash x[\varepsilon] : T} \qquad \text{abbreviated to} \qquad \frac{}{x : T \vdash x : T}$$

Typing λ abstraction and application We now address the conventional typing rules for the introduction and elimination of the arrow in the linear lambda calculus; the rules are below.

$$\frac{\Gamma, x :^0 (\varepsilon \vdash T) \vdash M : U}{\Gamma \vdash \lambda x.M : T \to U}\ {\to}\text{I} \qquad \frac{\Gamma \vdash M : T \to U \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash MN : U}\ {\to}\text{E}$$

The λ-bound variable x is local to M, hence of level 0.

Local soundness. Notice that $\Delta^{\geq 0}$ is true of all contexts Δ.

$$\frac{\dfrac{\Gamma, x :^0 (\varepsilon \vdash T) \vdash M : U}{\Gamma \vdash \lambda x.M : T \to U}\ {\to}\text{I} \qquad \Delta \vdash v : T}{\Gamma, \Delta \vdash (\lambda x.M)v : U}\ {\to}\text{E} \quad \Rightarrow \quad \frac{\Gamma, x :^0 (\varepsilon \vdash T) \vdash M : U \qquad \dfrac{\Delta \vdash v : T}{\Delta \vdash \varepsilon.v : (\varepsilon \vdash T)}\ \text{CTXT}}{\Gamma, \Delta \vdash [\varepsilon.v/x]M : U}\ \text{SUBS}$$

Local completeness:

$$\Gamma, \Delta \vdash M : T \to U \quad \Rightarrow \quad \frac{\dfrac{\Gamma, \Delta \vdash M : T \to U \qquad \dfrac{}{x :^0 (\varepsilon \vdash T) \vdash x[\varepsilon] : T}\ \text{VAR}}{\Gamma, \Delta, x :^0 (\varepsilon \vdash T) \vdash M(x[\varepsilon]) : U}\ {\to}\text{E}}{\Gamma, \Delta \vdash \lambda x.M(x[\varepsilon]) : T \to U}\ {\to}\text{I}$$

Typing * introduction and elimination Rules:

$$\frac{}{\vdash * : *}\ {*}\text{I} \qquad \frac{\Gamma \vdash M : * \qquad \Delta \vdash N : T}{\Gamma, \Delta \vdash \mathsf{let}\ * = M\ \mathsf{in}\ N : T}\ {*}\text{E}$$

Local soundness and local completeness:

$$\frac{\dfrac{}{\vdash * : *}\ {*}\text{I} \qquad \Gamma \vdash M : T}{\vdash \mathsf{let}\ * = *\ \mathsf{in}\ M : T}\ {*}\text{E} \quad \Leftrightarrow \quad \Gamma \vdash M : T$$

Typing box introduction and elimination The box introduction and elimination rules are as follows.

$$\frac{\Gamma \vdash \sigma : \tau}{\Gamma \vdash \mathsf{box}\,\sigma : \Box\tau}\ \Box\text{I} \qquad\qquad \frac{\Gamma^{\geq n} \vdash M : \Box\tau \qquad \Delta, x :^{n} \tau \vdash N : T}{\Gamma^{\geq n}, \Delta \vdash \mathsf{let\ box}\ x = M\ \mathsf{in}\ N : T}\ \Box\text{E}$$

In rule $\Box$I, the type of the box is the contextual modal type $\Box\tau$ if the contextual term σ has type τ. The context for the box elimination rule, $\Box$E, is split into two: one part ($\Gamma^{\geq n}$) to type M, the other (Δ) to type N. Term M must denote a code fragment, hence the type of M must be a contextual modal type $\Box\tau$. Term N is typed under context Δ extended with an entry for x. The context in the conclusion is formed by the juxtaposition of the contexts in the premises thus ensuring that all resources are fully consumed. The level n of variable x determines the level of code M. In contrast to preceding work [4, 8, 12], level control is shifted from box introduction to the rule that types contextual values. A rule derived from CTXT followed by $\Box$I coincides with the box introduction rule of Mœbius [8]:

$$\frac{\Gamma^{\geq n}, \overline{x :^{k} \tau}^{<n} \vdash M : T}{\Gamma^{\geq n} \vdash \mathsf{box}\,(\overline{x}.M) : \Box(\overline{\tau} \vdash T)}$$

Local soundness:

$$\frac{\dfrac{\Gamma^{\geq n} \vdash \sigma : \tau}{\Gamma^{\geq n} \vdash \mathsf{box}\,\sigma : \Box\tau}\ \Box\text{I} \qquad \Delta, x :^{n} \tau \vdash M : T}{\Gamma^{\geq n}, \Delta \vdash \mathsf{let\ box}\ x = \mathsf{box}\,\sigma\ \mathsf{in}\ M : T}\ \Box\text{E} \quad \Rightarrow \quad \frac{\Delta, x :^{n} \tau \vdash M : T \qquad \Gamma^{\geq n} \vdash \sigma : \tau}{\Gamma^{\geq n}, \Delta \vdash [\sigma/x]M : T}\ \text{SUBS}$$

Local completeness:

$$\Gamma^{\geq 2} \vdash M : \Box\tau \quad \Rightarrow \quad \frac{\Gamma^{\geq 2} \vdash M : \Box\tau \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x_i :^{1} (\varepsilon \vdash U_i) \vdash x_i[\varepsilon] : U_i}\ \text{VAR}}{x_i :^{1} (\varepsilon \vdash U_i) \vdash \varepsilon.x_i[\varepsilon] : (\varepsilon \vdash U_i)}\ \text{CTXT}}{u :^{2} \tau, \overline{x :^{1} (\varepsilon \vdash U)} \vdash u[\overline{\varepsilon.x[\varepsilon]}] : T}\ \text{VAR}}{u :^{2} \tau \vdash \overline{x}.u[\overline{\varepsilon.x[\varepsilon]}] : \tau}\ \text{CTXT}}{u :^{2} \tau \vdash \mathsf{box}\,(\overline{x}.u[\overline{\varepsilon.x[\varepsilon]}]) : \Box\tau}\ \Box\text{I}}{\Gamma^{\geq 2} \vdash \mathsf{let\ box}\ u = M\ \mathsf{in\ box}\,(\overline{x}.u[\overline{\varepsilon.x[\varepsilon]}]) : \Box\tau}\ \Box\text{E}$$

where τ abbreviates $(\overline{(\varepsilon \vdash U)} \vdash T)$ and $\overline{U} = U_1 \cdots U_m$ and similarly for $\overline{x}$.

An example from Mœbius Concrete syntax apart, the β-reduction for box terms is that of Mœbius [8]. We adapt an example to make it linear.

```
let box r = box (y. y + 2) in
let box u = box (c,x. 3 * z + c[2 * x]) in
            box (y. u[y. r[y], y])
```

reduces to

```
let box u = box (c,x. 3 * z + c[2 * x]) in
            box (y. u[y. y + 2, y])
```

under substitution `[(y. y + 2) / r]`, which in turn reduces to

```
box (y. 3 * z + 2 * y + 2)
```

under substitution [(c,x. 3 * z + c[2 * x]) / u], generating no administrative redexes. The original redex (or any contractum) has type $\Box(\vdash \mathbf{Int})$ under typing context $z :^3 (\vdash \mathbf{Int})$, assuming suitable rules for integer constants and arithmetic operators. Contextual value (y. y + 2), and hence also the contextual variable r, is typed at level 2. Contextual value (c,x. 3 * z + c[2 * x]), and hence also contextual variable u, is typed at level 2. Variable z can be typed at level 2.

3 From linear staged to session staged metaprogramming

This section briefly outlines what it takes to bridge the gap between the linear lambda calculus with multi-level contexts and the language required to type and run the examples in the introduction.

First and foremost, *session types* must be introduced in the syntax of types. In the absence of polymorphism, a new syntactic category S may be introduced for session types. Session types include input and output ($?T.S$ and $!T.S$), branch and select ($\&\{l : S_l\}^{l\in L}$ and $\oplus\{l : S_l\}^{l\in L}$), and channel closing (Wait and Close), where l denotes a label and L a label set. In addition, and in order to express the Stream type in the example, we need recursive types. They are usually introduced in the form of a μ (or rec) constructor and type references (sometimes called type variables), and treated equi-recursively. Session types then become an extra constructor for types T [7, 17]. For a more ambitious setting one may consider the sequential composition of types $(R;S)$ and continuation-less input and output ($?T$ and $!T$) as in context-free session types [1, 16].

At the level of *terms*, we require a few session-related primitives. We need constants to receive, to send, to select a choice, to wait for a channel to be closed and to close a channel. These can be given type schemes. For example, the type of constant send can be given by a type of the form $T \to !T.S \to S$, a function receiving a value to be sent and a channel on which to send the value, and returning the channel on which to continue interaction. We further need a constant to fork a new thread whose type can be given by type $(* \to *) \to *$, accepting a thunk and giving back a unit vale. Branching (achieved by pattern-matching in the examples) cannot be given by a constant. We then add a new term constructor match M with $\{l : M_l\}^{l\in L}$. Finally we need a primitive to create a new channel, usually of the form new S, returning the two end points of the newly created channel. Both the primitive receive operation and channel creation new S return a pair; we then need *linear* pairs of type $T \times U$, with introduction (M,N) and elimination let $(x,y) = M$ in N terms. Details can be found in Gay and Vasconcelos [5].

The description so far produces a *linear* session typed language. In particular we cannot take advantage of recursive types for there is no support for consuming such a type. Recursive functions are the usual means of consuming recursive types, a Stream for example, but they constitute the finished example of non-linear resources. A simple way out is to annotate entries in the typing context with the number of times a resource can be used, along the lines of Linear Haskell [3] or Quantitative Type Theory [2, 9] (also used in Nominal Session Types [10]). An alternative would be to introduce shared resources, functions in particular, in the linear type system [5].

The fork operator creates a new thread. Yet the term language depicted so far features no support for threads running concurrently. This is usually achieved by introducing a separate language for *processes*, denoted by P,Q. The basic processes are terms, for example of the form $\langle M \rangle$, composed by means of the parallel composition of two processes, $P \mid Q$, and of scope restriction, $(\nu xy)P$, describing the two end points x and y of a channel circumscribed to process P. Processes come equipped with a notion of reduction, featuring axioms for output-input, select-branch, and close-wait interactions, complemented

with suitable congruence rules. For example, the axiom, for output-input might be as follows,

$$(\nu xy)(\langle E[\mathsf{send}\, v\, x]\rangle \mid \langle F[\mathsf{receive}\, y]\rangle) \to (\nu xy)(\langle E[x]\rangle \mid \langle F[(v,y)]\rangle)$$

where the $\mathsf{send}\, v\, x$ term in evaluation context E is rewritten in channel x (so that term $E[x]$ may continue interaction on x) and the $\mathsf{receive}\, y$ term in evaluation context F is rewritten in a pair featuring the value received v and the continuation channel y (so that term $F[(v,y)]$ may use the value v in the message while continuing the interaction on y). Details can be found in different sources [1, 5].

4 Conclusion and future work

We show how to integrate staged metaprogramming into a linear lambda calculus and sketch how to extend the language to concurrency and session types. The type system we propose is deliberately non-algorithmic. We believe that standard techniques—including an explicitly typed and level-annotated syntax [8] and having the typing rules "return" the unused part of the context [18]—would lead to algorithmic type checking.

Acknowledgements This work was partly supported by JSPS Invitational Short-Term Fellowships for Research in Japan. It was further supported by the FCT through project Safe-Sessions (doi: 10.54499/PTDC/CCI-COM/6453/2020), by the LASIGE Research Unit (doi: 10.54499/UIDB/00408/2020 and doi: 10.54499/UIDP/00408/2020), and by the LIACC Research Unit (doi: 10.54499/UIDB/00027/2020 and doi: 10.54499/UIDP/00027/2020),

References

[1] Bernardo Almeida, Andreia Mordido, Peter Thiemann & Vasco T. Vasconcelos (2022): *Polymorphic lambda calculus with context-free session types*. *Inf. Comput.* 289(Part), p. 104948, doi:10.1016/J.IC.2022.104948.

[2] Robert Atkey (2018): *Syntax and Semantics of Quantitative Type Theory*. In: *LICS*, ACM, pp. 56–65, doi:10.1145/3209108.3209189.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2018): *Linear Haskell: practical linearity in a higher-order polymorphic language*. *Proc. ACM Program. Lang.* 2(POPL), pp. 5:1–5:29, doi:10.1145/3158093.

[4] Rowan Davies & Frank Pfenning (2001): *A modal analysis of staged computation*. *J. ACM* 48(3), pp. 555–604, doi:10.1145/382780.382785.

[5] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. *J. Funct. Program.* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[6] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR*, *LNCS* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[7] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP*, *LNCS* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[8] Junyoung Jang, Samuel Gélineau, Stefan Monnier & Brigitte Pientka (2022): *Mœbius: metaprogramming using contextual types: the stage where system F can pattern match on itself*. *Proc. ACM Program. Lang.* 6(POPL), pp. 1–27, doi:10.1145/3498700.

[9] Conor McBride (2016): *I Got Plenty o' Nuttin'*. In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 9600, Springer, pp. 207–233, doi:10.1007/978-3-319-30936-1_12.

[10] Andreia Mordido, Janek Spaderna, Peter Thiemann & Vasco T. Vasconcelos (2023): *Parameterized Algebraic Protocols*. *Proc. ACM Program. Lang.* 7(PLDI), pp. 1389–1413, doi:10.1145/3591277.

[11] Yuito Murase, Yuichi Nishiwaki & Atsushi Igarashi (2023): *Contextual Modal Type Theory with Polymorphic Contexts*. In: *ESOP*, *Lecture Notes in Computer Science* 13990, Springer, pp. 281–308, doi:10.1007/978-3-031-30044-8_11.

[12] Aleksandar Nanevski & Frank Pfenning (2005): *Staged computation with names and necessity*. *J. Funct. Program.* 15(5), pp. 893–939, doi:10.1017/S095679680500568X.

[13] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual modal type theory*. *ACM Trans. Comput. Log.* 9(3), pp. 23:1–23:49, doi:10.1145/1352582.1352591.

[14] Benjamin C. Pierce (2002): *Types and programming languages*. MIT Press.

[15] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE*, *LNCS* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[16] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-free session types*. In: *ICFP*, ACM, pp. 462–475, doi:10.1145/2951913.2951926.

[17] Vasco T. Vasconcelos (2012): *Fundamentals of session types*. *Inf. Comput.* 217, pp. 52–70, doi:10.1016/J.IC.2012.05.002.

[18] David Walker (2005): *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems, pp. 3–44. The MIT Press.

Towards a Semantic Characterisation of Global Type Well-formedness

Ilaria Castellani*
INRIA, Université Côte d'Azur, France
ilaria.castellani@inria.fr

Paola Giannini†
DiSSTE, Università del Piemonte Orientale, Italy
paola.giannini@uniupo.it

We address the question of characterising the well-formedness properties of multiparty session types semantically, i.e., as properties of the semantic model used to interpret types. Choosing Prime Event Structures (PESs) as our semantic model, we present semantic counterparts for the two properties that underpin global type well-formedness, namely projectability and boundedness, in this model. As a first step towards a characterisation of the class of PESs corresponding to well-formed global types, we identify some simple structural properties satisfied by such PESs.

1 Introduction

This paper builds on our previous work [4], where we investigated the use of Event Structures (ESs) as a denotational model for multiparty session types (MPSTs). That paper presented an ES semantics for both sessions and global types, using respectively Flow Event Structures (FESs) and Prime Event Structures (PESs), and showed that if a session is typable with a given global type, then the FES associated with the session and the PES associated with the global type yield isomorphic domains of configurations.

The ES semantics proposed in [4] abstracts away from the syntax of global types, by making explicit the concurrency relation between independent communications. This abstraction is expected since ESs are a "true concurrency" model, where concurrency is treated as a primitive notion. However, [4] focussed on the equivalence between the FES of a session and the PES of its global type, without drawing all the consequences of its results and demonstrating the full benefits of the ES semantics for MPSTs.

In the present paper, we move one step further by studying how the well-formedness property of global types considered in [4] is reflected in their interpretation as Prime Event Structures (PESs). In [4], global type well-formedness is the conjunction of a *projectability* condition and a *boundedness* condition. Having semantic counterparts for these conditions will enable us to reason directly on PESs, taking advantage of their faithful account of concurrency and of their graphical representation.

We prove that: 1) all global types that type the same network yield identical PESs, 2) our proposed properties of *semantic projectability* and *semantic boundedness* for PESs reflect the corresponding properties of global types, and 3) PESs obtained from global types enjoy some simple structural properties.

The rest of the paper is organised as follows. In Section 2 and Section 3 we recall the necessary background from [4] and present the result 1) above. In Section 4 we define our semantic notions of projectability and boundedness and prove the result 2) above. Section 5 is devoted to the result 3). Finally, in Section 6 we discuss related work and sketch some directions for future work.

In the paper, all theorems are given with proofs while all lemmas are stated without proofs.

*This research has been supported by the ANR17-CE25-0014-01 CISC project.
†This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the Università del Piemonte Orientale.

D. Costa, R. Hu (Eds.): Programming Language Approaches to Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. 11–21, doi:10.4204/EPTCS.401.2

2 Networks and Global Types

To set up the stage for our study, we recall the definitions of sessions and global types from [4]. In the core multiparty session calculus of [4], sessions are described as networks of sequential processes, and processes coincide with local types. Session *participants* are denoted by $\mathsf{p},\mathsf{q},\mathsf{r}$, and *messages* by λ,λ'.

Definition 2.1 (Processes and networks)

- *Processes are defined by:* $P ::=^{coind} \bigoplus_{i\in I} \mathsf{p}!\lambda_i;P_i \;\mid\; \sum_{i\in I}\mathsf{p}?\lambda_i;P_i \;\mid\; \mathbf{0}$
  where I is a finite non-empty index set and $\lambda_h \neq \lambda_k$ for $h \neq k$.
- Networks *are defined by:* $\mathsf{N} = \mathsf{p}_1[\![P]\!] \parallel \cdots \parallel \mathsf{p}_n[\![P]\!]$ *with $n \geq 1$ and $\mathsf{p}_h \neq \mathsf{p}_k$ for $h \neq k$.*

The symbol $::=^{coind}$ in the definition of processes indicates that the definition is coinductive. This allows infinite processes to be defined without using an explicit recursion operator. However, in order to achieve decidability we focus on regular processes, namely those with a finite number of distinct subprocesses. In writing processes, we will omit trailing $\mathbf{0}$'s and when $|I| = 1$ we will omit the choice symbol.

A network is a parallel composition of processes, each located at a different participant. The LTS semantics of networks is specified by the unique rule:

$$\mathsf{p}[\![\textstyle\bigoplus_{i\in I}\mathsf{q}!\lambda_i;P_i]\!] \parallel \mathsf{q}[\![\textstyle\sum_{j\in J}\mathsf{p}?\lambda_j;Q_j]\!] \parallel \mathsf{N} \xrightarrow{\mathsf{pq}\lambda_k} \mathsf{p}[\![P_k]\!] \parallel \mathsf{q}[\![Q_k]\!] \parallel \mathsf{N} \quad \text{where } k \in I\cap J \qquad [\textsc{Comm}]$$

Definition 2.2 (Global types) *Global types* G *are defined by:* $\mathsf{G} ::=^{coind} \mathsf{p}\to\mathsf{q}:\{\lambda_i;\mathsf{G}_i\}_{i\in I} \mid \mathsf{End}$
where I is finite non-empty index set and $\lambda_h \neq \lambda_k$ for $h \neq k$.

Here again, $::=^{coind}$ indicates that the definition is coinductive, and we focus on *regular* global types. We will omit trailing End's and when $|I| = 1$ we will write a global type $\mathsf{p}\to\mathsf{q}:\{\lambda;\mathsf{G}\}$ simply as $\mathsf{p}\xrightarrow{\lambda}\mathsf{q};\mathsf{G}$.

A *communication* $\mathsf{pq}\lambda$ represents the transmission of label λ on the channel pq from p to q. Communications are ranged over by α,β. The following notion of trace will be extensively used in the sequel.

Definition 2.3 (Traces) *A* trace σ,τ *is a finite sequence of communications, i.e.* $\sigma ::= \varepsilon \mid \alpha\cdot\sigma$.
The set of traces is denoted by Traces.

It is useful to define sets of participants also for communications and traces. We define $\mathsf{part}(\mathsf{pq}\lambda) = \{\mathsf{p},\mathsf{q}\}$, and we lift this definition to traces by letting $\mathsf{part}(\varepsilon)=\emptyset$ and $\mathsf{part}(\alpha\cdot\sigma) = \mathsf{part}(\alpha)\cup\mathsf{part}(\sigma)$.

As observed in [4], global types may be viewed as trees whose internal nodes are decorated by channels pq, leaves by End, and edges by labels λ. Given a global type, the sequences of decorations of nodes and edges on the path from the root to an edge in the tree of the global type are traces. We denote by $\mathsf{Tr}^+(\mathsf{G})$ the set of traces of G. By definition, $\mathsf{Tr}^+(\mathsf{End}) = \emptyset$ and each trace in $\mathsf{Tr}^+(\mathsf{G})$ is non-empty.

The set of *participants of a global type* G, $\mathsf{part}(\mathsf{G})$, is the union of the sets of participants of its traces, i.e. $\mathsf{part}(\mathsf{G}) = \bigcup_{\sigma\in\mathsf{Tr}^+(\mathsf{G})}\mathsf{part}(\sigma)$. The regularity assumption ensures that $\mathsf{part}(\mathsf{G})$ is finite for any G.

The semantics of global types is given by the standard LTS presented in Figure 1, where transitions are labelled by communications.

The projection of a global type onto participants is given in Figure 2. As usual, projection is defined only when it is defined on all participants. Due to the simplicity of our calculus, the projection of a global type, when defined, is simply a process. The definition is the standard one from [9, 10]: the projection of a choice type on the sender or the receiver yields an output choice or an input choice, while its projection on a third participant is its projection on the continuation of the branch, which must be equal on all branches. Our coinductive definition is more permissive than the standard one for infinite types (see [4]).

$$\mathsf{p} \to \mathsf{q} : \{\lambda_i; \mathsf{G}_i\}_{i\in I} \xrightarrow{\mathsf{pq}\lambda_j} \mathsf{G}_j \quad j \in I \quad [\textsc{Ecomm}] \qquad \frac{\mathsf{G}_i \xrightarrow{\alpha} \mathsf{G}_i' \quad \text{for all } i \in I \quad \mathsf{part}(\alpha) \cap \{\mathsf{p},\mathsf{q}\} = \emptyset}{\mathsf{p} \to \mathsf{q} : \{\lambda_i; \mathsf{G}_i\}_{i\in I} \xrightarrow{\alpha} \mathsf{p} \to \mathsf{q} : \{\lambda_i; \mathsf{G}_i'\}_{i\in I}} \quad [\textsc{Icomm}]$$

Figure 1: LTS for global types.

$$\mathsf{G} \upharpoonright \mathsf{r} = \mathbf{0} \text{ if } \mathsf{r} \notin \mathsf{part}(\mathsf{G}) \qquad (\mathsf{p} \to \mathsf{q} : \{\lambda_i; \mathsf{G}_i\}_{i\in I}) \upharpoonright \mathsf{r} = \begin{cases} \Sigma_{i\in I} \mathsf{p}?\lambda_i; \mathsf{G}_i \upharpoonright \mathsf{r} & \text{if } \mathsf{r} = \mathsf{q}, \\ \bigoplus_{i\in I} \mathsf{q}!\lambda_i; \mathsf{G}_i \upharpoonright \mathsf{r} & \text{if } \mathsf{r} = \mathsf{p}, \\ \mathsf{G}_1 \upharpoonright \mathsf{r} & \text{if } \mathsf{r} \notin \{\mathsf{p},\mathsf{q}\} \text{ and } \mathsf{r} \in \mathsf{part}(\mathsf{G}_1) \\ & \text{and } \mathsf{G}_i \upharpoonright \mathsf{r} = \mathsf{G}_1 \upharpoonright \mathsf{r} \text{ for all } i \in I \end{cases}$$

Figure 2: Projection of global types onto participants.

A global type G is *projectable* if $\mathsf{G} \upharpoonright \mathsf{p}$ is defined for all p.

The following property of boundedness for global types is used to ensure progress.

Definition 2.4 (Depth and boundedness) *The two functions $\delta(\mathsf{p},\sigma)$ and $\delta(\mathsf{p},\mathsf{G})$ are defined by:*

$$\delta(\mathsf{p},\sigma) = \begin{cases} |\, \sigma_1 \cdot \alpha \,| & \text{if } \sigma = \sigma_1 \cdot \alpha \cdot \sigma_2 \text{ and } \mathsf{p} \notin \mathsf{part}(\sigma_1) \text{ and } \mathsf{p} \in \mathsf{part}(\alpha) \\ 0 & \text{otherwise} \end{cases}$$

$$\delta(\mathsf{p},\mathsf{G}) = \sup(\{\delta(\mathsf{p},\sigma) \mid \sigma \in \mathsf{Tr}^+(\mathsf{G})\})$$

A global type G is bounded *if $\delta(\mathsf{p},\mathsf{G}')$ is finite for each participant p and each subtree G' of G.*

If $\delta(\mathsf{p},\mathsf{G})$ is finite, then there is no path in the tree of G in which p is delayed indefinitely. Note that if $\delta(\mathsf{p},\mathsf{G})$ is finite, G may have subtrees G' for which $\delta(\mathsf{p},\mathsf{G}')$ is infinite.

Definition 2.5 (Well-formed global types) *A global type G is* well formed *if it is projectable and bounded.*

We conclude this section by recalling the type system for networks. The unique typing rule for networks is Rule [Net] in Figure 3. It relies on a preorder on processes, $P \leq Q$, meaning that *process P can be used where we expect process Q*. This preorder plays the same role as the standard subtyping for local types, except that it is invariant for output processes (rather than covariant). This restriction is imposed in [4] in order to obtain bisimilar LTSs for networks and their global types, a property which in turn is used to prove our main result there (isomorphism of the configuration domains of the two ESs). The preorder rules are interpreted coinductively, since processes may have infinite (regular) trees.

A network is well typed if all its participants behave as specified by the projections of the same global type G. Rule [Net] is standard for MPSTs, so we do not discuss it further.

$$\mathbf{0} \leq \mathbf{0} \; [\leq\text{-}\mathbf{0}] \qquad \frac{P_i \leq Q_i \quad i \in I}{\Sigma_{i\in I\cup J} \mathsf{p}?\lambda_i; P_i \leq \Sigma_{i\in I} \mathsf{p}?\lambda_i; Q_i} [\leq\text{-In}] \qquad \frac{P_i \leq Q_i \quad i \in I}{\bigoplus_{i\in I} \mathsf{p}!\lambda_i; P_i \leq \bigoplus_{i\in I} \mathsf{p}!\lambda_i; Q_i} [\leq\text{-Out}]$$

$$\frac{P_i \leq \mathsf{G} \upharpoonright \mathsf{p}_i \quad i \in I \quad \mathsf{part}(\mathsf{G}) \subseteq \{\mathsf{p}_i \mid i \in I\}}{\vdash \prod_{i\in I} \mathsf{p}_i[\![P_i]\!] : \mathsf{G}} \; [\textsc{Net}]$$

Figure 3: Preorder on processes and network typing rule.

3 Event Structure Semantics of Global Types

In this section we present the interpretation of global types as Prime Event Structures, as proposed in our previous work [4]. We start by recalling the definition of *Prime Event Structure* (PES) and configuration from [12]. All the following definitions (from Definition 3.3 to Definition 3.10) are required background taken from [4], with some minor variations. The new material starts immediately after Definition 3.10.

Definition 3.1 ([12] Prime Event Structure) *A prime event structure* (PES) *is a tuple* $S = (E, \leq, \#)$ *where* E *is a denumerable set of events;* $\leq \subseteq (E \times E)$ *is a partial order relation, called the* causality *relation;* $\# \subseteq (E \times E)$ *is an irreflexive symmetric relation, called the* conflict *relation, satisfying the property of* conflict hereditariness*:* $\forall e, e', e'' \in E : e \# e' \leq e'' \Rightarrow e \# e''$.

A PES configuration is a set of events that may have occurred at some stage of computation.

Definition 3.2 ([12] PES configuration) *Let* $S = (E, \leq, \#)$ *be a prime event structure. A configuration of* S *is a finite subset* $\mathscr{X}$ *of* E *which is (1) downward-closed:* $e' \leq e \in \mathscr{X} \Rightarrow e' \in \mathscr{X}$*; and (2) conflict-free:* $\forall e, e' \in \mathscr{X}, \neg(e \# e')$.

The semantics of a PES S is given by its poset of configurations ordered by set inclusion, where $\mathscr{X}_1 \subset \mathscr{X}_2$ means that S may evolve from $\mathscr{X}_1$ to $\mathscr{X}_2$.

The events of the PES associated with a global type will be equivalence classes of particular traces. We introduce some notations for traces σ. We denote by $\sigma[i]$ the i-th element of σ. If $i \leq j$, we define $\sigma[i \ldots j] = \sigma[i] \cdots \sigma[j]$ to be the subtrace of σ consisting of the $(j - i + 1)$ elements starting from the i-th one and ending with the j-th one. If $i > j$, we convene that $\sigma[i \ldots j]$ denotes the empty trace ε.

A permutation equivalence on *Traces* is used to swap communications with disjoint participants.

Definition 3.3 (Permutation equivalence) *The permutation equivalence on Traces is the least equivalence* $\sim$ *such that*

$$\sigma \cdot \alpha \cdot \alpha' \cdot \sigma' \sim \sigma \cdot \alpha' \cdot \alpha \cdot \sigma' \quad \text{if} \quad \mathsf{part}(\alpha) \cap \mathsf{part}(\alpha') = \emptyset$$

We denote by $[\sigma]_\sim$ *the equivalence class of* σ*, and by* $Traces/\sim$ *the set of equivalence classes on Traces.*

The events of the PES associated with a global type are equivalence classes of particular traces that we call *pointed*. Intuitively, a pointed trace "points to" its last communication, in that all the preceding communications in the trace should cause some subsequent communication in the trace. Formally:

Definition 3.4 (Pointed trace) *A non empty trace* $\sigma = \sigma[1 \ldots n]$ *is said to be* pointed *if*

$$\forall i \,.\, 1 \leq i < n,\ \exists j \,.\, i < j \leq n \,.\ \mathsf{part}(\sigma[i]) \cap \mathsf{part}(\sigma[j]) \neq \emptyset$$

Note that the condition of Definition 3.4 is vacuously satisfied by any trace of length $n = 1$, since in that case there is no i such that $1 \leq i < n$.

Let us also point out that Definition 3.4 is slightly different from (but equivalent to) the definition of pointed trace given in [4].

For example, let $\alpha_1 = \mathsf{pq}\lambda_1$, $\alpha_2 = \mathsf{rs}\lambda_2$ and $\alpha_3 = \mathsf{rp}\lambda_3$. Then $\sigma_1 = \alpha_1$ and $\sigma_3 = \alpha_1 \cdot \alpha_2 \cdot \alpha_3$ are pointed traces, while $\sigma_2 = \alpha_1 \cdot \alpha_2$ is *not* a pointed trace.

If σ is non empty, we use $\mathsf{last}(\sigma)$ to denote the last communication of σ. It is easy to prove that, if σ is a pointed trace and $\sigma \sim \sigma'$, then σ' is a pointed trace and $\mathsf{last}(\sigma) = \mathsf{last}(\sigma')$.

Definition 3.5 (Global event) *Let* $\sigma = \sigma' \cdot \alpha$ *be a pointed trace. Then* $\gamma = [\sigma]_\sim$ *is a* global event*, also called* g-event*, with communication* α*, notation* $\mathsf{cm}(\gamma) = \alpha$.

Notice that, due to the observation above, $\mathsf{cm}(\gamma)$ is well defined. We denote by $\mathscr{GE}$ the set of g-events.

We now introduce an operator that adds a communication α in front of a g-event γ, provided α is a cause of some communication in the trace of γ. This ensures that the operator always transforms a g-event into another g-event. We call this operator *causal prefixing of a g-event by a communication*.

Definition 3.6 (Causal prefixing of a g-event by a communication)

1. *The* causal prefixing *of a g-event γ by a communication α is defined by:*
$$\alpha \circ \gamma = \begin{cases} [\alpha \cdot \sigma]_\sim & \text{if } \gamma = [\sigma]_\sim \text{ and } \mathsf{part}(\alpha) \cap \mathsf{part}(\sigma) \neq \emptyset \\ \gamma & \text{otherwise} \end{cases}$$
2. *The operator $\circ$ naturally extends to traces by: $\varepsilon \circ \gamma = \gamma \quad (\alpha \cdot \sigma) \circ \gamma = \alpha \circ (\sigma \circ \gamma)$*

An easy consequence of Clause 2 is that $(\sigma' \cdot \sigma) \circ \gamma = \sigma' \circ (\sigma \circ \gamma)$ for all σ and σ'.

Using causal prefixing, we can define the mapping $\mathsf{ev}(\cdot)$ which, applied to a trace σ, yields the g-event representing the communication $\mathsf{last}(\sigma)$ prefixed by its causes occurring in σ.

Definition 3.7 *The* g-event generated by a non-empty trace *is defined by:* $\mathsf{ev}(\sigma \cdot \alpha) = \sigma \circ [\alpha]_\sim$

Clearly, $\mathsf{ev}(\sigma)$ is a subtrace of σ and $\mathsf{cm}(\mathsf{ev}(\sigma)) = \mathsf{last}(\sigma)$. Observe that the function $\mathsf{ev}(\cdot)$ is not injective on the set of traces of a global type. For example, let $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_1 ; \mathsf{r} \xrightarrow{\lambda} \mathsf{s}, \lambda_2 ; \mathsf{r} \xrightarrow{\lambda} \mathsf{s}\}$. Let $\sigma_1 = \mathsf{pq}\lambda_1 \cdot \mathsf{rs}\lambda$ and $\sigma_2 = \mathsf{pq}\lambda_2 \cdot \mathsf{rs}\lambda$. Then $\sigma_1, \sigma_2 \in \mathsf{Tr}^+(\mathsf{G})$ and $\mathsf{ev}(\sigma_1) = \mathsf{ev}(\sigma_2) = [\mathsf{rs}\lambda]_\sim$.

Lemma 3.8 *If* $\mathsf{part}(\alpha_1) = \mathsf{part}(\alpha_2)$ *and* $\mathsf{ev}(\sigma \cdot \alpha_1) = [\sigma' \cdot \alpha_1]_\sim$*, then* $\mathsf{ev}(\sigma \cdot \alpha_2) = [\sigma' \cdot \alpha_2]_\sim$.

We proceed now to define the causality and conflict relations on g-events.

Definition 3.9 (Causality and conflict relations on g-events) *The* causality *relation $\leq$ and the* conflict *relation $\#$ on the set of g-events $\mathscr{GE}$ are defined by:*

1. *$\gamma \leq \gamma'$ if $\gamma = [\sigma]_\sim$ and $\gamma' = [\sigma \cdot \sigma']_\sim$ for some σ, σ';*
2. *$\gamma \# \gamma'$ if $\gamma = [\sigma \cdot \mathsf{pq}\lambda_1 \cdot \sigma_1]_\sim$ and $\gamma' = [\sigma \cdot \mathsf{pq}\lambda_2 \cdot \sigma_2]_\sim$ for some $\sigma, \sigma_1, \sigma_2, \mathsf{p}, \mathsf{q}, \lambda_1, \lambda_2$ where $\lambda_1 \neq \lambda_2$.*

If $\gamma = [\sigma \cdot \alpha \cdot \sigma' \cdot \alpha']_\sim$, then the communication α must be done before the communication α'. This is expressed by the causality $[\sigma \cdot \alpha]_\sim \leq \gamma$. An example is $[\mathsf{pq}\lambda]_\sim \leq [\mathsf{rs}\lambda' \cdot \mathsf{pq}\lambda \cdot \mathsf{sq}\lambda'']_\sim$. As regards the conflict relation, an example is $[\mathsf{rs}\lambda \cdot \mathsf{pq}\lambda_1 \cdot \mathsf{qr}\lambda]_\sim \# [\mathsf{pq}\lambda_2 \cdot \mathsf{rs}\lambda]_\sim$, since $\mathsf{pq}\lambda_2 \cdot \mathsf{rs}\lambda \sim \mathsf{rs}\lambda \cdot \mathsf{pq}\lambda_2$.

Definition 3.10 ([4] Event structure of a global type) *The* event structure of the global type G *is the triple $\mathscr{S}(\mathsf{G}) = (\mathscr{E}(\mathsf{G}), \leq_\mathsf{G}, \#_\mathsf{G})$ where: $\mathscr{E}(\mathsf{G}) = \{\mathsf{ev}(\sigma) \mid \sigma \in \mathsf{Tr}^+(\mathsf{G})\}$ and $\leq_\mathsf{G}$ and $\#_\mathsf{G}$ are the restrictions of $\leq$ and $\#$ to $\mathscr{E}(\mathsf{G})$.*

When clear from the context, we shall omit the subscript G in the relations $\leq_\mathsf{G}$ and $\#_\mathsf{G}$.

In the sequel, a PES obtained from a global type by Definition 3.10 will often be called a g-PES.

It should be stressed that Definition 3.10 only makes sense for global types that are projectable. Such global types are guaranteed to be *realisable* by some distributed implementation, i.e., to type some network. For these global types it has been shown in [4] that the semantics in Definition 3.10 preserves and reflects the operational semantics, namely that G performs a transition sequence labelled by a trace σ in the LTS of Figure 1 if and only if the associated PES admits the configuration $X = \{\mathsf{ev}(\sigma') \mid \sigma' \sqsubseteq \sigma\}$.

Example 3.11 *The global type* $\mathsf{G} = \mathsf{p} \to \mathsf{q} : \{\lambda_1 ; \mathsf{r} \xrightarrow{\lambda_3} \mathsf{s}, \lambda_2 ; \mathsf{r} \xrightarrow{\lambda_3} \mathsf{s}\}$ *is projectable, with projections:*

$$\mathsf{G} \upharpoonright \mathsf{p} = P = \mathsf{q}!\lambda_1 \oplus \mathsf{q}!\lambda_2 \qquad \mathsf{G} \upharpoonright \mathsf{q} = Q = \mathsf{p}?\lambda_1 + \mathsf{p}?\lambda_2 \qquad \mathsf{G} \upharpoonright \mathsf{r} = R = \mathsf{s}!\lambda_3 \qquad \mathsf{G} \upharpoonright \mathsf{s} = U = \mathsf{r}?\lambda_3$$

Clearly, G *types the network* $\mathsf{p}[\![P]\!] \parallel \mathsf{q}[\![Q]\!] \parallel \mathsf{r}[\![R]\!] \parallel \mathsf{s}[\![U]\!]$*. The PES S associated with* G *by Definition 3.10 has three events* $\gamma_1 = [\mathsf{pq}\lambda_1]_\sim, \gamma_2 = [\mathsf{pq}\lambda_2]_\sim, \gamma_3 = [\mathsf{rs}\lambda_3]_\sim$*, with* $\leq_\mathsf{G} = Id$ *and* $\gamma_1 \#_\mathsf{G} \gamma_2$.

Consider now the global type $\mathsf{G}' = \mathsf{p} \to \mathsf{q} : \{\lambda_1 ; \mathsf{End}, \lambda_2 ; \mathsf{r} \xrightarrow{\lambda_3} \mathsf{s}\}$*, where the first branch of the choice has no continuation. Clearly,* G' *is* not *projectable. However, Definition 3.10 associates the same PES S with* G'*, whereas* G' *and S do not have the same operational semantics, since* $\mathsf{G}' \xrightarrow{\mathsf{pq}\lambda_1} \mathsf{End}$ *while in the PES S the configuration* $\mathscr{X} = \{\gamma_1\}$ *can be extended to the configuration* $\mathscr{X}' = \{\gamma_1, \gamma_3\}$.

Our PES interpretation of global types explicitly brings out the concurrency between communications that is left implicit in the syntax of global types. We prove now that all well-formed global types that type the same network yield the same PES (Theorem 3.14). We start by proving a weaker theorem, which follows from results established in [12] and [4]. We say that two well-formed global types G and G' are *equivalent* if $\vdash \mathsf{N} : \mathsf{G}$ and $\vdash \mathsf{N} : \mathsf{G}'$ for some network N. Let $\cong$ denote PES isomorphism.

Theorem 3.12 (Equivalent well-formed global types yield isomorphic PESs) *Let* G *and* G' *be well-formed global types. If* $\vdash \mathsf{N} : \mathsf{G}$ *and* $\vdash \mathsf{N} : \mathsf{G}'$ *for some network* N*, then* $\mathscr{S}(\mathsf{G}) \cong \mathscr{S}(\mathsf{G}')$.

Proof. It was shown in [4] (Theorem 8.18 p 25) that if $\vdash \mathsf{N} : \mathsf{G}$ then the domain of configurations of $\mathscr{S}(\mathsf{G})$ is isomorphic to the domain of configurations of the Flow Event Structure associated with N. Then, from $\vdash \mathsf{N} : \mathsf{G}$ and $\vdash \mathsf{N} : \mathsf{G}'$ it follows that $\mathscr{S}(\mathsf{G})$ and $\mathscr{S}(\mathsf{G}')$ have isomorphic domains of configurations. A classical result in [12] (Theorem 9 p. 102) establishes that a PES S is isomorphic to the PES whose events are the prime elements of the domain of configurations of S, with causality relation given by set inclusion and conflict relation given by set inconsistency. We conclude that $\mathscr{S}(\mathsf{G}) \cong \mathscr{S}(\mathsf{G}')$. □

We now wish to go a step further by showing that $\mathscr{S}(\mathsf{G})$ and $\mathscr{S}(\mathsf{G}')$ are actually *identical* PESs. We write $\mathsf{G} \xrightarrow{\sigma} \mathsf{G}'$ if $\mathsf{G} \xrightarrow{\alpha_1} \mathsf{G}_1 \cdots \xrightarrow{\alpha_n} \mathsf{G}_n$ where $\alpha_1 \cdots \alpha_n = \sigma$ and $\mathsf{G}_n = G'$, and $\mathsf{G} \xrightarrow{\sigma}$ if there exists G' such that $\mathsf{G} \xrightarrow{\sigma} \mathsf{G}'$. A similar notation will be used for transition sequences of a network N.

Our next theorem relies on the following key lemma.

Lemma 3.13 *Let* G *be a global type and* σ *be a trace. Then:*

1. $\sigma \in \mathsf{Tr}^+(\mathsf{G})$ *implies* $\mathsf{G} \xrightarrow{\sigma}$*;*
2. $\mathsf{G} \xrightarrow{\sigma}$ *implies* $\mathsf{ev}(\sigma) \in \mathscr{E}(\mathsf{G})$.

Theorem 3.14 (Equivalent well-formed global types yield identical PESs) *Let* G *and* G' *be well-formed global types. If* $\vdash \mathsf{N} : \mathsf{G}$ *and* $\vdash \mathsf{N} : \mathsf{G}'$ *for some network* N*, then* $\mathscr{S}(\mathsf{G}) = \mathscr{S}(\mathsf{G}')$.

Proof. It is enough to show that $\mathscr{S}(\mathsf{G})$ and $\mathscr{S}(\mathsf{G}')$ have exactly the same sets of events, since events are defined syntactically and the relations of causality and conflict can be extracted from their syntax.

We prove that $\mathscr{E}(\mathsf{G}) = \mathscr{E}(\mathsf{G}')$. Let $e \in \mathscr{E}(\mathsf{G})$. By Definition 3.10 there exists $\sigma \in \mathsf{Tr}^+(\mathsf{G})$ such that $\mathsf{ev}(\sigma) = e$. Then $\mathsf{G} \xrightarrow{\sigma}$ by Lemma 3.13(1), from which we deduce $\mathsf{N} \xrightarrow{\sigma}$ by the Session Fidelity result in [4] (Theorem 6.11 p. 16). Then $\mathsf{G}' \xrightarrow{\sigma}$ by the Subject Reduction result in [4] (Theorem 6.10 p. 16). By Lemma 3.13(2) this implies $\mathsf{ev}(\sigma) \in \mathscr{E}(\mathsf{G}')$, i.e., $e \in \mathscr{E}(\mathsf{G}')$.

□

4 Semantic Well-formedness

We now investigate semantic counterparts for the syntactic well-formedness property of global types. Recall that type well-formedness is the conjunction of two properties: projectability and boundedness. We start by defining a notion of *semantic projectability* for PESs. We first give some auxiliary definitions.

Definition 4.1 *Let* G *be a global type and* $\mathscr{S}(\mathsf{G}) = (\mathscr{E}(\mathsf{G}), \leq, \#)$*. Two events* $\gamma_1, \gamma_2 \in \mathscr{E}(\mathsf{G})$ *are in* initial conflict*,* $\gamma_1 \#_{in} \gamma_2$*, if* $\gamma_1 = [\sigma \cdot \mathsf{pq}\lambda_1]_\sim$ *and* $\gamma_2 = [\sigma \cdot \mathsf{pq}\lambda_2]_\sim$ *for some* $\sigma, \mathsf{p}, \mathsf{q}, \lambda_1, \lambda_2$ *such that* $\lambda_1 \neq \lambda_2$.

Definition 4.2 (Projection of traces on participants) *The* projection of σ on r*,* $\sigma@\mathsf{r}$*, is defined by:*

$$\varepsilon@\mathsf{r} = \varepsilon \qquad (\mathsf{pq}\lambda \cdot \sigma)@\mathsf{r} = \begin{cases} \mathsf{q}!\lambda \cdot \sigma@\mathsf{r} & \text{if } \mathsf{r} = \mathsf{p} \\ \mathsf{p}?\lambda \cdot \sigma@\mathsf{r} & \text{if } \mathsf{r} = \mathsf{q} \\ \sigma@\mathsf{r} & \text{if } \mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\} \end{cases}$$

Definition 4.3 (Semantic projectability) *Let* G *be a global type and* $\mathscr{S}(\mathsf{G}) = (\mathscr{E}(\mathsf{G}), \leq, \#)$. *We say that* $\mathscr{S}(\mathsf{G})$ *is* semantically projectable *if for all* $\gamma_1, \gamma_2 \in \mathscr{E}(\mathsf{G})$ *in initial conflict:*

if there is $\gamma_1' = [\sigma_1 \cdot \alpha_1]_\sim$ *with* $\gamma_1 \leq_{\mathsf{G}} \gamma_1'$ *and* $\mathsf{r} \in \mathsf{part}(\alpha_1) \backslash \mathsf{part}(\mathsf{cm}(\gamma_1))$,

then there is $\gamma_2' = [\sigma_2 \cdot \alpha_2]_\sim$ *with* $\gamma_2 \leq_{\mathsf{G}} \gamma_2'$ *and* $\alpha_2 = \alpha_1$ *and* $\sigma_2 @ \mathsf{r} = \sigma_1 @ \mathsf{r}$.

Note that if G is semantically projectable, then also any subterm G' of G is semantically projectable.

We now show that, if a global type G is projectable, then all initial conflicts between two participants p and q in the event structure $\mathscr{S}(\mathsf{G})$ reflect branching points between p and q in the tree of G. In general, the mapping from branching points in the tree of G to initial conflicts in $\mathscr{S}(\mathsf{G})$ is not injective, namely, there may be several branching points in the tree of G that give rise to the same initial conflict in $\mathscr{S}(\mathsf{G})$.

Lemma 4.4 (Initial conflicts in $\mathscr{S}(\mathsf{G})$ reflect branching points in the tree of G) *Let* G *be a global type and* $\mathscr{S}(\mathsf{G}) = (\mathscr{E}(\mathsf{G}), \leq, \#)$. *Let* $\gamma_1, \gamma_2 \in \mathscr{E}(\mathsf{G})$ *be in initial conflict and* $\gamma_i = [\sigma \cdot \alpha_i]_\sim$ *for* $i \in \{1,2\}$. *If* G *is projectable then there exists* σ' *such that* $\sigma' \cdot \alpha_i \in \mathsf{Tr}^+(\mathsf{G})$ *and* $\mathsf{ev}(\sigma' \cdot \alpha_i) = [\sigma \cdot \alpha_i]_\sim$ *for* $i \in \{1,2\}$.

Let $\sigma \in \mathsf{Tr}^+(\mathsf{G})$. We denote by G_σ the *subterm of* G *after* σ, which is easily defined by induction on the length of σ. The converse of Lemma 4.4 is immediate, since any subterm of G is G_σ for some $\sigma \in \mathsf{Tr}^+(\mathsf{G})$. So if $\mathsf{G}_\sigma = \mathsf{p} \to \mathsf{q} : \{\lambda_i; \mathsf{G}_i'\}_{i \in I}$ with $\{1,2\} \subseteq I$, then the two events $\gamma_1 = \mathsf{ev}(\sigma \cdot \mathsf{pq}\lambda_1)$ and $\gamma_2 = \mathsf{ev}(\sigma \cdot \mathsf{pq}\lambda_2)$ are in initial conflict because by Lemma 3.8 there exists σ' such that $\mathsf{ev}(\sigma \cdot \mathsf{pq}\lambda_i) = [\sigma' \cdot \mathsf{pq}\lambda_i]_\sim$ for $i \in \{1,2\}$.

Theorem 4.5 (Projectability preservation) *If* G *is projectable then* $\mathscr{S}(\mathsf{G})$ *is semantically projectable.*

Proof. Let $\gamma_1, \gamma_2 \in \mathscr{E}(\mathsf{G})$ and $\gamma_1 \#_{in} \gamma_2$. By definition, $\gamma_1 = [\sigma \cdot \mathsf{pq}\lambda_1]_\sim$ and $\gamma_2 = [\sigma \cdot \mathsf{pq}\lambda_2]_\sim$ for some $\sigma, \mathsf{p}, \mathsf{q}, \lambda_1, \lambda_2$ such that $\lambda_1 \neq \lambda_2$. Let $\alpha_i = \mathsf{pq}\lambda_i$ for $i \in \{1,2\}$.
Since G is projectable, by Lemma 4.4 there exists σ' such that $\sigma' \cdot \alpha_i \in \mathsf{Tr}^+(\mathsf{G})$ and $\mathsf{ev}(\sigma' \cdot \alpha_i) = [\sigma \cdot \alpha_i]_\sim$ for $i \in \{1,2\}$. Then $\mathsf{G}_{\sigma'} = \mathsf{p} \to \mathsf{q} : \{\lambda_i; \mathsf{G}_i'\}_{i \in I}$ with $\{1,2\} \subseteq I$. Since G is projectable, also $\mathsf{G}_{\sigma'}$ is projectable. Thus, for any $\mathsf{r} \notin \{\mathsf{p}, \mathsf{q}\}$ we get $\mathsf{G}_1' \upharpoonright \mathsf{r} = \mathsf{G}_2' \upharpoonright \mathsf{r}$.
Let $\gamma_1' \in \mathscr{E}(\mathsf{G})$, with $\gamma_1 \leq_{\mathsf{G}} \gamma_1'$, $\mathsf{cm}(\gamma_1') = \beta$, and $\mathsf{r} \in \mathsf{part}(\beta) \backslash \{\mathsf{p}, \mathsf{q}\}$. Since $\gamma_1 \leq_{\mathsf{G}} \gamma_1'$, it must necessarily be $\gamma_1' = [\sigma \cdot \alpha_1 \cdot \sigma_1 \cdot \beta]_\sim = \mathsf{ev}(\sigma' \cdot \alpha_1 \cdot \sigma_1' \cdot \beta)$ for some σ_1, σ_1'. Then $\sigma_1' \cdot \beta$ is a path in G_1'.
Since $\mathsf{G}_{\sigma'}$ is projectable, $\mathsf{G}_1' \upharpoonright \mathsf{r} = \mathsf{G}_2' \upharpoonright \mathsf{r}$. Then there must be a path σ_2' of G_2' such that $\sigma_1' \cdot \beta @ \mathsf{r} = \sigma_2' \cdot \beta @ \mathsf{r}$.
We want to show that $\gamma_2' = \mathsf{ev}(\sigma' \cdot \alpha_2 \cdot \sigma_2' \cdot \beta) = [\sigma' \cdot \alpha_2 \cdot \sigma_2 \cdot \beta]_\sim$ for some σ_2, i.e., that $\gamma_2 \leq \gamma_2'$. Now, if $\mathsf{part}(\beta) \cap \{\mathsf{p}, \mathsf{q}\} \neq \emptyset$, we can conclude immediately. So, let us assume $\mathsf{part}(\beta) \cap \{\mathsf{p}, \mathsf{q}\} = \emptyset$.
Since $\gamma_1' = \mathsf{ev}(\sigma' \cdot \alpha_1 \cdot \sigma_1' \cdot \beta) = [\sigma \cdot \alpha_1 \cdot \sigma_1 \cdot \beta]_\sim$ we know that $\alpha_1 \cdot \sigma_1 \cdot \beta$ is a pointed trace. So, there must be a *bridging communication sequence* between α_1 and β, namely there must be a subtrace $\beta_1 \cdots \beta_n$ of σ_1 for some $n \geq 1$ such that

$\mathsf{part}(\alpha_1) \cap \mathsf{part}(\beta_1) \neq \emptyset \qquad \mathsf{part}(\beta_n) \cap \mathsf{part}(\beta) \neq \emptyset \qquad \mathsf{part}(\beta_i) \cap \mathsf{part}(\beta_{i+1}) \neq \emptyset \;\text{ for }\; 1 \leq i < n$

Correspondingly, we will have $\sigma_1' = \widehat{\sigma}_1 \cdot \beta_1 \cdots \widehat{\sigma}_n \cdot \beta_n$. There are now two possible cases:

- $\mathsf{part}(\beta_i) \neq \{\mathsf{p}, \mathsf{q}\}$ for every $i = 1, \ldots, n$. Since $\mathsf{G}_{\sigma'}$ is projectable, $\mathsf{G}_1' \upharpoonright \mathsf{s} = \mathsf{G}_2' \upharpoonright \mathsf{s}$ for all $\mathsf{s} \notin \{\mathsf{p}, \mathsf{q}\}$, i.e., $\sigma_1' @ \mathsf{s} = \sigma_2' @ \mathsf{s}$. Therefore all the β_i's for $1 \leq i \leq n$ must occur in the same order in σ_2', i.e. $\sigma_2' = \tau_1 \cdot \beta_1 \cdots \tau_n \cdot \beta_n$ for some $\tau_1, \ldots, \tau_n$. Hence $\mathsf{ev}(\sigma' \cdot \alpha_2 \cdot \sigma_2' \cdot \beta) = [\sigma' \cdot \alpha_2 \cdot \sigma_2 \cdot \beta]_\sim$ for some σ_2.

- $\mathsf{part}(\beta_j) = \{\mathsf{p}, \mathsf{q}\}$ for some j, $1 \leq j \leq n$. Let k be the maximum such index j. Then we know that $\mathsf{part}(\beta_h) \neq \{\mathsf{p}, \mathsf{q}\}$ for every $h, k+1 \leq h \leq n$, and either $\mathsf{p} \in \mathsf{part}(\beta_{k+1})$ or $\mathsf{q} \in \mathsf{part}(\beta_{k+1})$. Therefore all the β_h's for $k+1 \leq h \leq n$ must occur in the same order in σ_2', i.e. $\sigma_2' = \tau_k \cdot \beta_{k+1} \cdots \tau_n \cdot \beta_n$ for some $\tau_k, \ldots, \tau_n$. Hence, $\mathsf{ev}(\sigma' \cdot \alpha_2 \cdot \sigma_2' \cdot \beta) = [\sigma' \cdot \alpha_2 \cdot \sigma_2 \cdot \beta]_\sim$ for some σ_2. □

The converse is not true, i.e., semantic projectability of $\mathscr{S}(\mathsf{G})$ does not imply projectability of G, as shown by the global type G' in Example 3.11. Note that there is no network behaving as prescribed by G'. We conjecture that for realisable global types semantic projectability implies projectability.

We now define a notion of semantic boundedness for PESs, which is *global* in that it looks simultaneously at all occurrences of each participant p in the g-events whose last communication involves p. Let $S=(E,\leq,\#)$ be a g-PES. For any participant p, let $\mathsf{p}\in\mathsf{part}(S)$ if there exists $\gamma\in E$ such that $\mathsf{p}\in\mathsf{part}(\gamma)$.

Definition 4.6 (Semantic k-depth and semantic boundedness) *Let $S=(E,\leq,\#)$ be a g-PES. The two functions $\delta^k_{\mathsf{sem}}(\mathsf{p},\gamma)$ and $\delta_{\mathsf{gsem}}(\mathsf{p},S)$ are defined by:*

$$\delta^k_{\mathsf{sem}}(\mathsf{p},[\sigma]_\sim)=\begin{cases}|\,\sigma\,| & \text{if } \sigma=\sigma_1\cdot\alpha_1\cdots\sigma_k\cdot\alpha_k \text{ and } \mathsf{p}\in\mathsf{part}(\alpha_i) \text{ for } i=1,\ldots,k\\ & \qquad\text{and } \mathsf{p}\notin\mathsf{part}(\sigma_i) \text{ for } i=1,\ldots,k\\ 0 & \text{otherwise}\end{cases}$$

$$\delta^k_{\mathsf{gsem}}(\mathsf{p},S)=\sup(\{\delta^k_{\mathsf{sem}}(\mathsf{p},\gamma)\mid\gamma\in E\}) \text{ for every } k\in\mathbb{N}$$

S is semantically bounded *if $\delta^k_{\mathsf{gsem}}(\mathsf{p},S)$ is finite for each participant $\mathsf{p}\in\mathsf{part}(S)$ and each $k\in\mathbb{N}$.*

Theorem 4.7 (Boundedness preservation) *If G is bounded, then $\mathscr{S}(\mathsf{G})$ is semantically bounded.*

Proof. Let G be bounded and $\mathscr{S}(\mathsf{G})=(\mathscr{E}(\mathsf{G}),\leq,\#)$. We want to show that $\delta^k_{\mathsf{gsem}}(\mathsf{p},\mathscr{S}(\mathsf{G}))$ is finite for each participant $\mathsf{p}\in\mathsf{part}(\mathscr{S}(\mathsf{G}))$ and each $k\in\mathbb{N}$. Fix some p. If $\mathsf{p}\notin\mathsf{part}(\mathsf{G})$ then $\mathsf{p}\notin\mathsf{part}(\mathscr{S}(\mathsf{G}))$ and thus the statement is vacuously true. So, assume $\mathsf{p}\in\mathsf{part}(\mathsf{G})$. We show now, by induction on k, that there exists $n_k\in\mathbb{N}$ such that $\delta^k_{\mathsf{sem}}(\mathsf{p},\gamma)\leq n_k$ for any $\gamma\in\mathscr{E}(\mathsf{G})$.

- Case $k=1$. We may assume $\gamma=[\sigma_1\cdot\alpha_1]_\sim\in\mathscr{E}(\mathsf{G})$ with $\mathsf{p}\notin\mathsf{part}(\sigma_1)$ and $\mathsf{p}\in\mathsf{part}(\alpha_1)$, since for any γ' not of this shape we have $\delta^1_{\mathsf{sem}}(\mathsf{p},\gamma')=0$ and we can immediately conclude. Then there exists $\sigma'_1\cdot\alpha_1\in\mathsf{Tr}^+(\mathsf{G})$ such that $\gamma=\mathsf{ev}(\sigma'_1\cdot\alpha_1)$. Note that it must be $\mathsf{p}\notin\mathsf{part}(\sigma'_1)$, since otherwise there would be some β in σ'_1 such that $\mathsf{part}(\beta)\cap\mathsf{part}(\alpha_1)\neq\emptyset$ and the function $\mathsf{ev}(\cdot)$ would keep this β, contradicting the hypothesis $\mathsf{p}\notin\mathsf{part}(\sigma_1)$. Let $n_1=\delta(\mathsf{p},\mathsf{G})=\sup(\{\delta(\mathsf{p},\sigma)\mid\sigma\in\mathsf{Tr}^+(\mathsf{G})\})$. By Definition 2.4 $\delta(\mathsf{p},\sigma'_1\cdot\alpha_1)=|\,\sigma'_1\cdot\alpha_1\,|\leq n_1$. Then by Definition 4.6 we get $\delta^1_{\mathsf{sem}}(\mathsf{p},[\sigma_1\cdot\alpha_1]_\sim)=|\,\sigma_1\cdot\alpha_1\,|\leq|\,\sigma'_1\cdot\alpha_1\,|\leq n_1$.

- Case $k>1$. Assume that $\sup(\{\delta^{k-1}_{\mathsf{sem}}(\mathsf{p},\gamma)\mid\gamma\in\mathscr{E}(\mathsf{G})\})\leq n_{k-1}$. Let $\sigma=\sigma_1\cdot\alpha_1\cdots\sigma_k\cdot\alpha_k$ be such that $\mathsf{p}\notin\mathsf{part}(\sigma_i)$ and $\mathsf{p}\in\mathsf{part}(\alpha_i)$ for every $i=1,\ldots,k$, and let $\gamma=[\sigma]_\sim\in\mathscr{E}(\mathsf{G})$. Then there exists $\sigma'=\sigma'_1\cdot\alpha_1\cdots\sigma'_k\cdot\alpha_k\in\mathsf{Tr}^+(\mathsf{G})$ such that $\gamma=\mathsf{ev}(\sigma')$. For each $i=1,\ldots,k$ we must have $\mathsf{p}\notin\mathsf{part}(\sigma'_i)$, because otherwise we would contradict the hypothesis $\mathsf{p}\notin\mathsf{part}(\sigma_i)$ (as argued in the previous case). Let now $\sigma''=\sigma'_1\cdot\alpha_1\cdots\sigma'_{k-1}\cdot\alpha_{k-1}$, and consider the subterm $\mathsf{G}_{\sigma''}$ of G.

Since G is bounded and $\mathsf{G}_{\sigma''}$ is a subtree of G, by Definition 2.4 we get $\delta(\mathsf{p},\mathsf{G}_{\sigma''})=\sup(\{\delta(\mathsf{p},\sigma)\mid\sigma\in\mathsf{Tr}^+(\mathsf{G}_{\sigma''})\})=m$ for some $m\in\mathbb{N}$. Therefore $\delta^1_{\mathsf{sem}}(\mathsf{p},[\sigma_k\cdot\alpha_k]_\sim)=|\,\sigma_k\cdot\alpha_k\,|\leq|\,\sigma'_k\cdot\alpha_k\,|=\delta(\mathsf{p},\sigma'_k\cdot\alpha_k)\leq m$. $\delta(\mathsf{p},\sigma'_k\cdot\alpha_k)=|\,\sigma'_k\cdot\alpha_k\,|\leq m$. Let $n_k=n_{k-1}+m$. We may conclude that $\delta^k_{\mathsf{sem}}(\mathsf{p},[\sigma_1\cdot\alpha_1\cdots\sigma_k\cdot\alpha_k]_\sim)=\delta^{k-1}_{\mathsf{sem}}(\mathsf{p},[\sigma_1\cdot\alpha_1\cdots\sigma_{k-1}\cdot\alpha_{k-1}]_\sim)+\delta^1_{\mathsf{sem}}(\mathsf{p},[\sigma_k\cdot\alpha_k]_\sim)\leq n_{k-1}+m=n_k$.

□

5 Structural Properties of g-PESs

In this section we discuss some additional properties of the PESs we obtain by interpreting global types. Some of these properties do not depend on the well-formedness of global types but only on their syntax. For instance, since we adopt for global types the *directed choice* construct of [9, 10]: $\mathsf{p}\to\mathsf{q}:\{\lambda_i;\mathsf{G}_i\}_{i\in I}$, every branch of a choice uses the same channel pq. As a consequence, g-PESs satisfy the property of *initial conflict uniformity*: in every set $X=\{\gamma_1,\ldots,\gamma_n\}$ of initially conflicting g-events, every $\gamma_i\in X$ uses the same channel pq in its last communication, i.e., $\mathsf{cm}(\gamma_i)=\mathsf{pq}\lambda_i$ for some λ_i. Moreover, since global types have deterministic LTSs, where no state can perform two different transitions with the same label,

the same holds for g-PESs: if $\mathscr{X}, \mathscr{X} \cup \{\gamma_1\}, \mathscr{X} \cup \{\gamma_2\}$ are configurations of the same g-PES, then $\gamma_1 \neq \gamma_2$ implies $\mathsf{cm}(\gamma_1) \neq \mathsf{cm}(\gamma_2)$. If moreover $\neg(\gamma_1 \# \gamma_2)$, we additionally have $\mathsf{part}(\mathsf{cm}(\gamma_1)) \cap \mathsf{part}(\mathsf{cm}(\gamma_2)) = \emptyset$.

Note that our core session calculus may be viewed as a *linear* subcalculus of Milner's calculus CCS, where parallel composition appears only at top level and any pair of processes P and Q, run by participants p and q, can communicate only via two unidirectional channels: channel pq for communication from p to q, and channel qp for communication from q to p. Hence, *restricted parallel composition*, which is the kind of parallel composition used in session calculi, where processes are only allowed to communicate with each other but not to proceed independently, becomes an associative operation, while it is not associative in full CCS (as observed by Milner in his 1980 book, see [11] page 21).

Then, a natural question is: how does our ES semantics for the linear subcalculus of CCS compare to the ES semantics proposed in [2, 3] for other fragments of CCS? To carry out this comparison, we would need to take a more extensional view of g-PESs, forgetting about the syntactic structure of g-events and retaining only their last communication. In other words, we should consider *Labelled PESs*, where events are labelled by communications $\mathsf{pq}\lambda$ and have no specific structure. Moreover, some care should be taken since, unlike our session calculus, our language for global types is *not* a subcalculus of CCS: indeed, while the syntax of global types is included in that of CCS with guarded sums, their semantics is not the same as that of CCS processes, since some communications may be performed under guards.

More in detail, the work [2] provides a characterisation of the class of Labelled PESs obtained by interpreting the fragment of CCS built from actions $a, b, \ldots$ by means of the three constructors $+, ;, \|$, denoting respectively choice, sequential composition and parallel composition with no communication. As a matter of fact, [2] uses slightly more relaxed PESs where conflict is not required to be hereditary - let us call them r-PESs - and shows that the Labelled r-PESs obtained for that fragment of CCS are exactly those satisfying two structural properties called *triangle freeness* and *N-freeness*. In conjunction, these two properties express the possibility of extracting a head operator among $+, ;, \|$ from the structure of the r-PES. We recall from [2] the definition of these properties. Let $\smile$ denote the *concurrency* relation on the events of a PES $S = (E, \leq, \#)$, defined by $\smile = (E \times E) - (\leq \cup \geq \cup \#)$. Let $\diamond = (\leq \cup \geq)$ denote *causal connection*. By definition the three relations $\diamond, \#$ and $\smile$ set a partition over $E \times E$. Then triangle freeness (or ∇-freeness) is defined as the absence of a triple of events e, e', e'' such that $e \diamond e' \# e'' \smile e$ (see [2] page 41). Note that one half of triangle-freeness, where $\diamond$ is replaced by $\geq$, is implied by conflict hereditariness in g-PESs. We conjecture that g-PESs satisfy also the other half of triangle-freeness, where $\diamond$ is replaced by $\leq$, namely they do not feature the pattern $e \leq e' \# e'' \smile e$, a situation known as *asymmetric confusion* in Petri nets. The property of N-freeness is slightly more involved. For any $R \in \{\leq, \#, \smile\}$, let R^ε be the reflexive and symmetric closure of R and $\ddagger(R)$ be the $R-$*incomparability* relation defined by $\ddagger(R) = (E \times E) - R^\varepsilon$. Then the N-freeness property is stated as follows:

N-freeness $\quad \forall R \in \{\leq, \#, \smile\}$: $(\, e_0 R e_1 \wedge e_0 \ddagger(R) e_2 \wedge e_2 R e_3 \wedge e_1 \ddagger(R) e_3 \,) \implies (\, e_0 R e_3 \implies e_2 R e_1 \,)$

This property does not hold for g-PESs, e.g., it does not hold for the g-PES of the global type $\mathsf{G} = \mathsf{p} \xrightarrow{\lambda_0} \mathsf{q}; \mathsf{p} \xrightarrow{\lambda_1} \mathsf{t}; \mathsf{r} \xrightarrow{\lambda_2} \mathsf{s}; \mathsf{q} \xrightarrow{\lambda_3} \mathsf{s}$, with $e_0 = [\mathsf{pq}\lambda_0]_\sim, e_1 = [\mathsf{pq}\lambda_0 \cdot \mathsf{pt}\lambda_1]_\sim, e_2 = [\mathsf{rs}\lambda_2]_\sim, e_3 = [\mathsf{pq}\lambda_0 \cdot \mathsf{rs}\lambda_2 \cdot \mathsf{qs}\lambda_3]_\sim$.

However, we may show that g-PESs satisfy particular instances of N-freeness, for instance when R is the covering relation of $\leq$ and $e_1 \#_{in} e_3$ (in which case conflict hereditariness enforces $e_0 \smile e_2$).

The paper [3], on the other hand, presents a Flow Event Structure semantics for the whole calculus CCS. Our conjecture is that this semantics should coincide with the Flow ES semantics proposed for sessions in [4]. However, this is not entirely trivial since the semantics of [3] uses self-conflicting events, a specific feature of Flow ESs, to interpret restricted parallel composition, while the semantics of [4] uses a pre-processing phase to rule out the g-events that do not satisfy a causal well-foundedness condition, and these events are a superset of those that are self-conflicting in the semantics of [3]. However, one

may already observe that the Flow ESs obtained by interpreting sessions in [4] trivially satisfy the axiom Δ put forward in [5] in order to guarantee that CCS parallel composition is a categorical product.

6 Conclusion

We conclude by further discussing related work and by sketching some directions for future work.

Related work. In Section 5 we compared our PES semantics for global types to existing ES semantics for other fragments of CCS. In that case, the comparison was somewhat hindered by the fact that the target ESs were not exactly the same (PESs vs r-PESs vs Flow ESs). We now turn to other proposals of denotational models for MPSTs. The models that are closest to ours are the *graphical choreographies* by Guanciale and Tuosto [14], the *choreography automata* by Barbanera, Lanese and Tuosto [1], the *global choreographies* by de'Liguoro, Melgratti and Tuosto [6], and the *branching pomsets* by Edixhoven et al. [8]. It should be noted that most of these works deal with asynchronous communication, so "events" (or communications) are split into send events and receive events. Common well-formedness conditions proposed in these works are *well-branchedness* [1, 6, 8], which in our case is enforced by the syntax of global types, and *well-sequencedness* [1, 6], which is automatically enforced by our PES semantics. As regards the use of ESs to model MPSTs, the paper [6] also uses PESs to model (asynchronous) choreographies, but it needs an additional type system to obtain projectability (so, the resulting notion of projectability is not totally semantic). In [8], asynchronous choreographies are modelled with branching pomsets, a model featuring both concurrency and choice, which is compared with various classes of ESs.

Future work. In this paper, we have devised semantic counterparts for the well-formedness conditions of global types. However, we have only gone half the way in establishing a characterisation of the class of Prime ESs representing well-formed global types. To achieve such a characterisation, we should prove the converse of Theorem 4.7 and the following weaker form of the converse of Theorem 4.5:

Conjecture *[Projectability reflection]* Let G be a global type. If $\mathscr{S}(\mathsf{G})$ is semantically projectable then there exists a projectable global type G' such that $\mathscr{S}(\mathsf{G}') = \mathscr{S}(\mathsf{G})$.

If this conjecture were true, then our PES semantics for global types would also provide a way to "sanitise" ill-formed global types. For instance, starting from the g-PES $\mathscr{S}(\mathsf{G}')$ of the ill-formed global type G' of Example 3.11, we would be able to get back to the well-formed global type G of the same example or to the well-formed global type $\mathsf{G}'' = \mathsf{r} \xrightarrow{\lambda_3} \mathsf{s};\mathsf{p} \to \mathsf{q} : \{\lambda_1;\mathsf{End}, \lambda_2;\mathsf{End}\}$. Once we achieve a characterisation for this class of g-PESs, the next step would be to propose an algorithm to synthesise a well-formed global type (or directly a network) from a g-PES of this class. A further goal would be to semantically characterise less restrictive notions of projection, such as the one proposed in [7].

Since g-PESs are images of regular trees, it would be worth investigating their connection with Regular Event Structures [13]. Moreover, as argued in the previous section, extensional g-PESs, where g-events have no structure and are labelled by their last communication, may be viewed as Labelled ESs whose observable behaviour (the communications) is deterministic. Hence, such extensional g-PESs could be characterised by the trace language they recognise[1].

Acknowledgments. We are grateful to the anonymous reviewers for their useful suggestions. The first author would also like to acknowledge interesting discussions with Nobuko Yoshida, Francisco Ferreira and Raymond Hu during her visits to Oxford University and Queen Mary University of London in 2023.

[1] Here,"trace" should be intended as a Mazurkiewicz trace, namely as an equivalence class of standard traces with respect to an independence relation I on the alphabet of the language, which in our case is given by $\mathsf{pq}\lambda \; I \; \mathsf{rs}\lambda'$ if $\{\mathsf{p},\mathsf{q}\} \cap \{\mathsf{r},\mathsf{s}\} = \emptyset$.

References

[1] Franco Barbanera, Ivan Lanese & Emilio Tuosto (2020): *Choreography Automata*. In Simon Bliudze & Laura Bocchi, editors: *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020*, 12134, Springer, pp. 86–106, doi:10.1007/978-3-030-50029-0_6.

[2] Gérard Boudol & Ilaria Castellani (1988): *Concurrency and atomicity*. *Theoretical Computer Science* 59(1-2), pp. 25–84, doi:10.1016/0304-3975(88)90096-5.

[3] Gérard Boudol & Ilaria Castellani (1988): *Permutation of transitions: an event structure semantics for CCS and SCCS*. In J.W. de Bakker, W.-P. de Roever & G. Rozenberg, editors: *REX School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Noordwijkerhout, *Lecture Notes in Computer Science* 354, Springer-Verlag, pp. 411–427, doi:10.1007/BFb0013028.

[4] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2023): *Event structure semantics for multiparty sessions*. *J. Log. Algebraic Methods Program.* 131, p. 100844, doi:10.1016/j.jlamp.2022.100844.

[5] Ilaria Castellani & Guo Qiang Zhang (1997): *Parallel product of event structures*. *Theoretical Computer Science* 179(1-2), pp. 203–215, doi:10.1016/S0304-3975(96)00104-1.

[6] Ugo de'Liguoro, Hernán C. Melgratti & Emilio Tuosto (2022): *Towards refinable choreographies*. *J. Log. Algebraic Methods Program.* 127, p. 100776, doi:10.1016/j.jlamp.2022.100776.

[7] Pierre-Malo Deniélou & Nobuko Yoshida (2013): *Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types*. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska & David Peleg, editors: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, *Lecture Notes in Computer Science* 7966, Springer, pp. 174–186, doi:10.1007/978-3-642-39212-2_18.

[8] Luc Edixhoven, Sung-Shik Jongmans, José Proença & Ilaria Castellani (2024): *Branching pomsets: Design, expressiveness and applications to choreographies*. *J. Log. Algebraic Methods Program.* 136, p. 100919, doi:10.1016/J.JLAMP.2023.100919.

[9] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In George C. Necula & Philip Wadler, editors: *POPL*, ACM Press, New York, pp. 273–284, doi:10.1145/1328897.1328472.

[10] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *Journal of ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[11] Robin Milner (1980): *A Calculus of Communicating Systems*. *Lecture Notes in Computer Science* 92, Springer, doi:10.1007/3-540-10235-3.

[12] Mogens Nielsen, Gordon Plotkin & Glynn Winskel (1981): *Petri Nets, Event Structures and Domains, Part I*. *Theoretical Computer Science* 13(1), pp. 85–108, doi:10.1016/0304-3975(81)90112-2.

[13] Mogens Nielsen & P. S. Thiagarajan (2002): *Regular Event Structures and Finite Petri Nets: The Conflict-Free Case*. In Javier Esparza & Charles Lakos, editors: *Applications and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002, Adelaide, Australia, June 24-30, 2002, Proceedings*, *Lecture Notes in Computer Science* 2360, Springer, pp. 335–351, doi:10.1007/3-540-48068-4_20.

[14] Emilio Tuosto & Roberto Guanciale (2018): *Semantics of global view of choreographies*. *J. Log. Algebraic Methods Program.* 95, pp. 17–40, doi:10.1016/j.jlamp.2017.11.002.

Session Types for the Transport Layer: Towards an Implementation of TCP*

Samuel Cavoj
samuel@cavoj.net
University of Glasgow

Ivan Nikitin
ivan@niktivan.org
University of Glasgow

Colin Perkins
csp@csperkins.org
University of Glasgow

Ornela Dardha
ornela.dardha@glasgow.ac.uk
University of Glasgow

Session types are a typing discipline used to formally describe communication-driven applications with the aim of fewer errors and easier debugging later into the life cycle of the software. Protocols at the transport layer such as TCP, UDP, and QUIC underpin most of the communication on the modern Internet and affect billions of end-users. The transport layer has different requirements and constraints compared to the application layer resulting in different requirements for verification. Despite this, to our best knowledge, no work shows the application of session types at the transport layer. In this work, we discuss how multiparty session types (MPST) can be applied to implement the TCP protocol. We develop an MPST-based implementation of a subset of a TCP server in Rust and test its interoperability against the Linux TCP stack. Our results highlight the differences in assumptions between session type theory and the way transport layer protocols are usually implemented. This work is the first step towards bringing session types into the transport layer.

1 Introduction

Session types [11] are a typing discipline for communication protocols. They can describe the sequence of messages exchanged between participants over a communication channel and can be used to verify that the protocol is implemented correctly or has certain desirable properties. Further, session types can be realised within programming languages and used to type-check the implementation of a protocol against a session type definition, with type errors indicating inconsistencies between implementation and the session type. Session types have been an active area of research since the beginning of the 1990s [11] and have been implemented in a number of programming languages including C [26], Java [13] and Rust [14, 15] and other programming languages [9, 16, 25, 27, 29].

Network protocols that are part of the Internet Protocol suite (TCP/IP) are the foundation of the Internet. They are responsible for interoperability between different devices, operating systems, and applications. To ensure that different implementations of the same protocol are compatible, they must adhere to a technical specification which, in the case of Internet protocols, is defined in a series of documents, known as RFCs [8], developed by the Internet Engineering Task Force (IETF). Specifically, the latest version of the TCP protocol specification is defined in RFC 9293 [7].

The IETF follows a consensus-based process when developing standards [4, 30], with protocol specifications being developed in working group meetings and on mailing lists over a multi-year period. The resulting RFCs are written primarily in English prose, allowing the documents to be used in the

*Supported in part by the UK EPSRC grants EP/X027309/1 and EP/S036075/1.

D. Costa, R. Hu (Eds.): Programming Language Approaches to Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. 22–36, doi:10.4204/EPTCS.401.3

consensus-building process, but the natural language can be ambiguous and unclear and this can lead to inconsistent and non-conforming implementations. [22, 23, 28]. In this sense, ensuring the correctness of Internet protocols is vital. Developing formalised models of the protocols described in RFCs is one way to achieve this. Session types are one such modelling technique that has not previously been explored for transport-layer protocols, such as TCP.

In this paper, we implement a core subset of the TCP protocol in the Rust programming language and use session types to describe the network operations. Session types are encoded into native Rust types and the type checker is used to verify that the implementation follows the session type specification. In this way, the Rust compiler verifies that the implementation of the protocol is correct in terms of the types of messages exchanged and the order in which they are exchanged, i.e., that it follows the declared session type, for a session type model describing a synchronous subset of TCP. Additionally, session types are used to describe the application interface, so we can verify that the application uses the TCP implementation correctly.

Our contributions are as follows:

1. **Session Types Libraries.** We develop[1] the libraries required for encoding the session type model into native Rust types in an ergonomic fashion (§4.1).
2. **Implementation.** We implement a subset of the TCP protocol [7], including key aspects of both the user/TCP interface and the TCP/lower-level interface, in Rust while adhering to the session type model. This is done in a way such that the Rust compiler can detect deviation from the session type (§4.4).
3. **Testing.** We test our implementation against a real TCP stack (§5).

The remainder of this paper is structured as follows. Section 2 briefly reviews the multiparty session type model we use. Section 3 outlines key properties of TCP and its state machine. Section 4 describes our session typed implementation of TCP in Rust. Section 5 evaluates the correctness of our implementation. Finally, Section 6 reviews related work and concludes.

2 Session types

Session types [11] describe communication among participants in a distributed system in terms of the types and order of messages that are exchanged. A single session type describes the sequence of messages sent or received from the perspective of one of the participants. The theory of session types was later extended to multiparty session types (MPST) which can describe protocols between any number of participants [12].

In this paper, the bottom-up multiparty session type approach [31] is used to describe TCP. An example of a simple ping-pong protocol using this approach is demonstrated in Equation 1. When type-checking any type using the bottom-up approach, we must additionally choose a *safety invariant*. Safety invariants are parameters associated with the properties a protocol may demonstrate during runtime, such as deadlock-freedom and liveness. Each safety invariant is accompanied by specific typing rules (not presented here) that guarantee the maintenance of the corresponding invariant. If the protocol successfully type-checks with the instantiation of the safety invariant, it will manifest the property represented by the invariant during its runtime.

[1] Our session type library and TCP implementation is available at `https://github.com/sammko/tcpst2`

$$\begin{aligned}\Gamma_1 \;=\;& \mathtt{s}[\mathtt{a}] : \mathtt{b} \oplus l_1(\mathtt{ping}) \,.\, \mathtt{b} \;\&\; l_3(\mathtt{pong}) \,.\, \mathtt{end},\\ & \mathtt{s}[\mathtt{b}] : \mathtt{a} \;\&\; l_2(\mathtt{ping}) \,.\, \mathtt{a} \oplus l_1(\mathtt{pong}) \,.\, \mathtt{end}\end{aligned} \tag{1}$$

The implications of this approach are that global types and the concept of duality are not used. Instead of duality, the compatibility invariant is used to check that actions are dual between the given types. However, a protocol can still be described using session types even if safety does not hold.

3 Transmission Control Protocol (TCP)

The TCP transport is layered on top of the datagram service provided by the Internet Protocol (IP). The IP layer provides an unreliable, best-effort, datagram service, where packets may be lost, duplicated, delayed, or re-ordered in transit. TCP segments, sent within IP packets, contain sequence numbers and acknowledgements such that, upon detection of a lost packet, either triggered by a timer expiration or receipt of a triple-duplicate acknowledgement, the sender can re-transmit the lost segment.

TCP is usually used in a client-server manner, but also supports a rarely used simultaneous open mode with peer-to-peer connections. In the context of this paper, we assume client-server usage, with one side being a passive server listening for incoming connections, while the other is an active client initiating the connection. We describe the operation of the TCP state machine below and provide a diagram of the TCP state transitions in Figure 1.

The establishment of a reliable connection between two network devices is facilitated by the TCP three-way handshake. It commences with the initiation of a connection with the client sending a TCP segment with the SYN (synchronise) bit set in the header and containing the client's initial sequence number. The server responds with a segment with the SYN and ACK bits set, acknowledging the client's initial sequence number and providing the initial sequence number the server will use. Finally, the client confirms the establishment of the connection by sending a segment with the ACK (acknowledge) bit set. This sequence ensures both sides agree on their initial sequence numbers and confirm their willingness to communicate.

TCP uses a sliding window algorithm to manage data transmission by sending segments with sequence numbers. The window size determines the number of unacknowledged segments in transit. The receiver discards unacceptable segments falling outside the expected sequence range, leading to retransmission by the sender. Acknowledgements are sent upon receiving new data, indicating the next expected contiguous sequence number. TCP handles packet loss or reordering at the IP layer by detecting duplicate acknowledgements; a triple-duplicate acknowledgement triggers retransmission. Additionally, TCP utilises a retransmission timeout (RTO) mechanism, dynamically adjusted based on network conditions. TCP buffers play a crucial role on both the sender and receiver sides, with the send buffer holding outgoing segments awaiting acknowledgement and the receive buffer storing incoming segments yet to be delivered to the application.

The TCP closing handshake, another three-way handshake involving packets with the FIN (finish) and ACK (acknowledge) bits signifies the end of a connection. The initial party sends a FIN packet, followed by an acknowledgement from the other party, culminating in a reciprocal FIN-ACK exchange. The final step includes an acknowledgement from the original sender, leading to the TIME-WAIT state. This state ensures a reliable closure, allowing the handling of delayed or duplicate IP packets before concluding the connection.

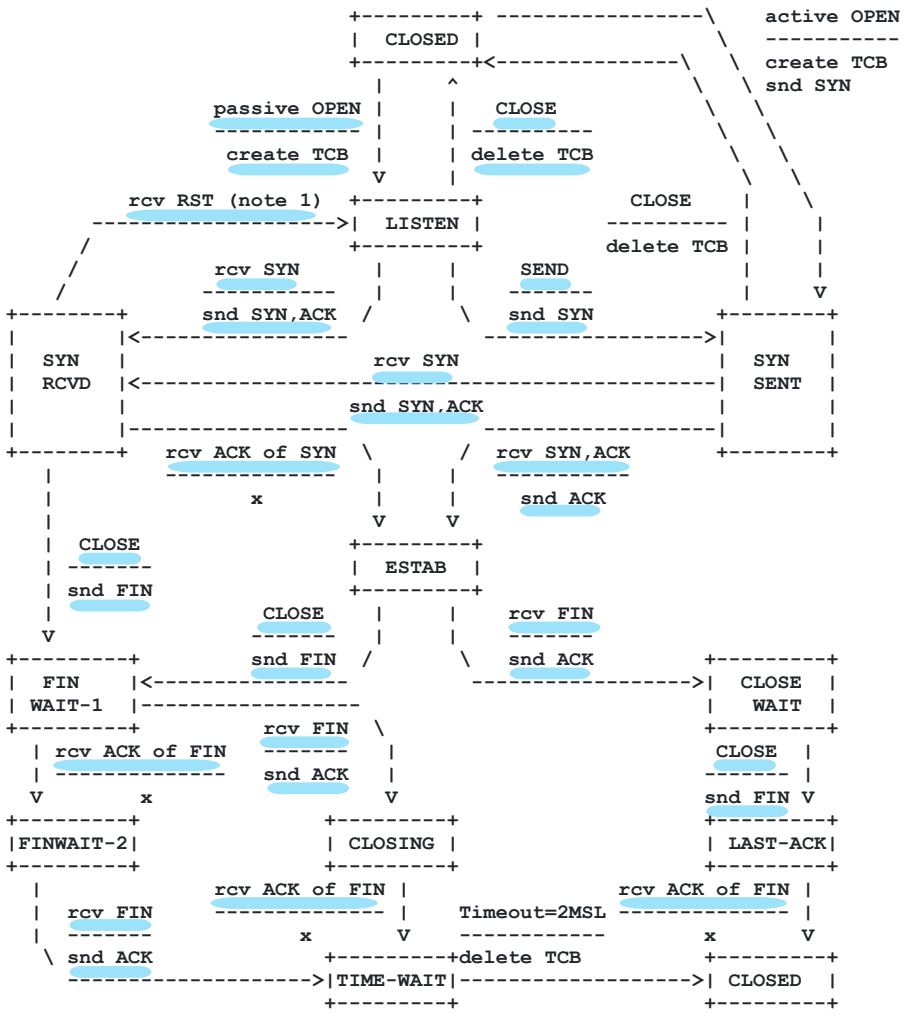


Figure 1: The state transition diagram of TCP in RFC9293 [7]. We annotate the diagram with the messages and transitions modelled in our implementation. Note that we do not model timeouts as part of the type system, hence, the TIME-WAIT to CLOSED transition is not implemented using session types. Additionally, we do not implement the active OPEN case of the handshake for simplicity (as this would not demonstrate any new modelling or implementation techniques), this is however possible using our implementation.

4 Implementation

We implement the basic functionality of the TCP server protocol while modelling both the network and the application interface using session types. Note that more information on the implementation can be found in the Appendix. Under the *less is more* formalisation of multiparty session types [31], the roles we are considering the following:

Server User The server application using the TCP protocol.

Server System The TCP implementation.

Client System The TCP implementation on the other end of the network.

The channel between the Server System and the Client System represents the network. The messages exchanged between the Server User and the Server System are a formalisation of the user/TCP (i.e., application programming) interface and do not pass over the network. The system call interfaces, representing the user and the system (in this paper simulated through threads), each have a session type which prescribes their behaviour relative to the other roles. The Client System role has no associated session type in our implementation as it is assumed to be another host on the Internet and not part of our program.

4.1 Defining session types

The basic building blocks of our implementation are the generic structs `OfferOne`, `OfferTwo`, `SelectOne`, and `SelectTwo`. All of these implement the trait `Action` which represents a general session type. The type parameters of the structs encode the role the action is performed with respect to, the types of messages exchanged, and the continuation of the session. In addition, the `End` struct is also an `Action` and represents the end session type.

The `OfferTwo` struct has five type parameters. The first is the peer role, and the next two are the types of messages exchanged in either of the two branches of the offer and the final two parameters are the session types of the continuations of the two branches.

```
pub struct OfferTwo<R, M1, M2, A1, A2>
where R: Role, M1: Message, M2: Message,
    A1: Action, A2: Action,
{
    phantom: PhantomData<(R, M1, M2, A1, A2)>,
}
```

The `OfferTwo` struct, as a way of encoding a session type construct in Rust, has type parameters but contains no data. The `PhantomData`-typed field contained within the struct is a zero-sized marker type that simulates a field of the given type to support the Rust type checker.[2]

The `SelectTwo` struct has the same type parameters and is also a zero-sized type. Finally, the non-branching actions `OfferOne` and `SelectOne` have only three type parameters: the peer role, the message type, and the continuation type, but are otherwise analogous.

To define a session type one can define a type alias for the root action of the session. For example a simple session type for a client-server interaction could be defined as follows:

```
type ServerSt = OfferOne<Client, Request, SelectOne<Client, Response, End>>;
type ClientSt = SelectOne<Server, Request, OfferOne<Server, Response, End>>;
```

This basic syntax, however, quickly becomes unwieldy when defining more complex session types. To address this, we have implemented a macro which converts a more readable syntax into the full definition of the type. Rust's `macro_rules!` mechanism is powerful enough to allow us to define a syntax which attempts to mimic the mathematical notation. The macro is called `St!` and the `ServerSt` type from the above example could be re-written as follows:

```
type ServerSt = St![(Client & Request).(Client + Response).end]
```

[2] `https://doc.rust-lang.org/nomicon/phantom-data.html`

The macro is recursive and supports arbitrary nesting of offers and selections. The full definition can be found in the `st_macros.rs` file of the source code.

4.2 Multi-way `Offer` branching

As Rust does not support variadic generic types, we are not aware of a way to implement a generic `Offer` type which would support a variable number of branches. Hence we implement `OfferOne` and `OfferTwo` as separate constructs with some repetition in the corresponding infrastructure such as the `offer_one`, `offer_two` and similar selection methods. These are described in §4.4.

However, support for more than two branches is required in practice. A simple way to do this is to implement `OfferThree`, `OfferFour`, ..., in the same way, along with the code supporting this. This leads to more code duplication, but does not increase complexity, and the usage is straightforward.

As an alternative, to avoid duplication, we chose a nesting approach where a branching of arity N is transformed into a two-way branching between the first case and an $N-1$ branching of the other cases.[3] This is recursively expanded until it finally results in a tree of two-way forks, where each *left* branch represents a single case from the original N. All *right* branches except the bottom-most one lead to a virtual node which was not present in the original type.

4.3 Recursive session types

Type aliases in Rust cannot be recursive. The reason for this is that a type alias does not create a new type and is merely another name for the same type. For instance, defining a type alias `type A = X<A>` is not allowed because the expansion would be infinite – the name `X<A>` would expand to `X<X<A>>`, etc. However, we somehow need to represent recursive session types.

Fortunately, this is not difficult to circumvent. Whereas type *aliases* cannot be recursive, there is no such restriction for types themselves, as long as the size of the type is finite. As such, types which contain a recursive cycle with no indirection are not allowed as the size of the type is infinite. But inserting indirection into the cycle (such as a reference `&T` or `Box<T>`) resolves this problem since the size of a reference does not depend on the size of the target type `T`.

4.4 Using session types

A channel provides methods to send and receive messages which consume a corresponding session type and return the continuation. The type of the channel is generic over the roles between which it exists and the method signatures ensure that they can be only called with an appropriate session type instance and message. Consider a channel of type `Channel<R1, R2>` which we define as the endpoint belonging to role `R1`, i.e. it can send to or receive from `R2`. Then its `select_one` method could have the following signature:

```
fn select_one<M, A>(&mut self, _o: SelectOne<R2, M, A>, message: M) -> A
where M: Message, A: Action;
```

It is generic over the message type, but it has to match the one prescribed by the provided session typed *token*. The role `R2` is already bound by the channel type. The token is moved into this function, so the owner cannot re-use it. The continuation type from the token is instantiated and returned to the

[3] Naturally, it would be better to split into halves instead, reducing the expansion depth from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$ but this is more difficult to implement and provides little practical benefit in all but the most extreme branching cases.

caller for further operations. And, of course, the message is transmitted over the underlying transport the nature of which is not restricted by this abstraction. The only requirement is that the `Message` trait can be converted to a representation that the channel can process, which is the reason for the trait in the first place.

The implementation of the `offer` methods is slightly more involved. Once a message is received from the underlying transport we must determine which branch of the offer to take and convert it to the appropriate message type. We outsource the decision to a function we receive as an argument called the *picker*. We find that in our particular use case, having the capability to differentiate branches based on external context is necessary. This allows us to distinguish the receipt of an expected packet from the error condition when an unexpected packet is received

4.5 Establishing a Connection

A TCP connection is established via a three-way handshake as described in Section 3. We define the `ServerSystemSessionType` to describe creation of the server socket (receipt of `Open` from the server user), creating the internal state (the "TCB"; §A.1), waiting for a SYN from the client, and generating the SYN-ACK segment, corresponding to the transition through the LISTEN state of Figure 1 into the SYN RCVD state:

```
pub type ServerSystemSessionType = St![
    (RoleServerUser & Open).
    (RoleServerUser + TcbCreated).
    (RoleClientSystem & Syn).
    (RoleClientSystem + SynAck).
    ServerSystemSynRcvd
];
```

The `ServerSystemSynRcvd` type describes the SYN RCVD state, with branches indicating the transition to the ESTAB state in `ServerSystemCommLoop` if the received ACK is acceptable or closing the connection if not.

```
Rec!(pub ServerSystemSynRcvd, [
    (RoleClientSystem & {
        Ack. // acceptable (i.e., matches the SYN-ACK sent)
            (RoleServerUser + Connected).
            ServerSystemCommLoop,
        Ack. // unacceptable
            (RoleClientSystem + {
                Ack.ServerSystemSynRcvd, Rst.(RoleServerUser + Close).end
            })
     })
]);
```

The implementation of three-way handshake is further described in Appendix A.1.

4.6 Data Transmission and Re-transmission

When a TCP segment goes unacknowledged for a certain amount of time, it is retransmitted. There are two implementation choices that could be made here: incorporate timeouts into the type system, or leave

them out and instead signal session type transitions using external timeouts. The session type theory we are using does not have a notion of timeouts, nor does any session type work containing timeouts [1, 2, 5] have the ability to model the operations needed for TCP timeouts. Hence, we opt to emulate timeouts by introducing a virtual message type and adding it as another branch to the offer session type. In this branch, we continue with a select operation, retransmitting an ACK message and then recursively receiving the next message. The `offer` method on the network channel now accepts another argument, specifying the timeout duration or `None` if no timeout is desired. If the retransmission queue is empty, no timeout should be employed as we run into an issue if it expires – the session type requires a segment to be sent, but there is nothing to send. Further details around data transmission are in Appendix A.2.

4.7 Closing the connection

Closing a TCP connection is a two-step process usually combined into a three-way handshake, as shown in the lower half of Figure 1. Each direction of the stream can be closed independently by sending a segment with the FIN bit set. The Server System session type describes receiving a FIN first and then deciding to close eventually, after allowing the user to send more data using the `ServerSystemCloseWait` session type:

```
Rec!(pub ServerSystemCloseWait, [
    (RoleServerUser & {
        Data.
            (RoleClientSystem + Ack).
            (RoleClientSystem & Ack /* empty ack */).
            ServerSystemCloseWait,
        Close.
            (RoleClientSystem + FinAck).
            (RoleClientSystem & Ack).
            end
    })
]);
```

The case where the server closes first is handled by the `ServerSystemFinWait1` type:

```
pub type ServerSystemFinWait1 = St![
    (RoleClientSystem & {
        Ack. // ACK of FIN
            ServerSystemFinWait2,
        FinAck. // FIN and ACK of our FIN at the same time
            (RoleClientSystem + Ack).
            end
    })];
```

The branch in the `ServerSystemFinWait1` type represents the ways in which the closing handshake can proceed after sending a FIN to close the connection and entering into the FINWAIT-1 state (see Figure 1): either a segment containing an ACK is received causing the system to transition to FINWAIT-2, waiting for a segment containing a FIN indicating that the peer has also finished; or a segment with both FIN and ACK is received causing the final ACK to be sent and terminating the connection via the implied CLOSING and TIME-WAIT states. The `ServerSystemFinWait2` implementation is analogous, but elided due to space constraints.

Finally, in a full TCP implementation, a "simultaneous close" situation can occur where both peers decide to close at the same time. This is not handled by our implementation as it is rarely used and does not fit with the call-and-response style of interaction we model – there is no opportunity for the server to decide to close while waiting for the client.

5 Evaluation

To evaluate our Server System component, we have implemented a simple echo server in the Server User. Every piece of data it receives from the system is split into lines, each line is reversed and then sent back.

The functionality of the server tested is as follows:

Establishing a connection by running netcat and connecting to the server.

Exchanging data with the client by typing in messages manually.

Initiating connection close by sending an empty line to the server. The server user has been programmed to close the connection if an empty line is received.

Responding to connection close by typing `^C` which causes netcat to close the socket and therefore send a FIN to the server.

Correctly handling a FIN-ACK response to a FIN by piping an empty line immediately followed by EOF to netcat. In this situation netcat sends the empty line but does not shutdown the socket immediately. Instead it waits for the server to send a FIN-ACK and then sends a FIN-ACK in response.

We tested our TCP implementation primarily against the Linux kernel TCP stack, running our program and connecting to it using a Linux user-space TCP client (netcat). We have used Scapy [32], a packet manipulation framework, to emulate a misbehaving TCP client or network and evaluate the behaviour of our server in response to this. This included sending packets with invalid sequence numbers, invalid acknowledgement numbers, spurious retransmission or overlapping segments. Finally, we have tested our implementation against the Linux kernel TCP stack with the addition of simulated network errors using the `netem` module to introduce packet loss, delay and reordering. Our test script configures the `TCP_NODELAY` option on the socket and sends messages in a loop with a small delay between them. This ensures that the client sends a lot of small packets to observe the effect of packet loss and reordering. The received data was then compared to the expected output. In all cases, we utilised a packet sniffer to monitor the communication.

All of the presented test cases were found to be handled correctly provided the server is in the ESTABLISHED state. During the opening three-way handshake, after the initial SYN segment, handling is robust as well. We found that the server can handle packet loss and reordering errors and the connection can recover once the impairments are lifted. The server does not cache out of order segments in the receive window affecting performance, since these segments will need to be re-transmitted, but not correctness. This is a limitation inherent in using synchronous session types to model an synchronous protocol that permits reordering and packet loss, and suggests future work to extend the modelling approach.

6 Related Work and Conclusion

Network protocols have been used as examples for various session type theories. The main protocols used as a demonstration in many works are SMTP [3, 6, 17, 18, 19, 20, 21] and POP3 [10, 24]. Both of

these protocols are application-layer protocols. Due to this, models of SMTP and POP3 can assume the guarantees provided by the underlying transport layer protocol – in most cases this is TCP. Specifically, any faults, retransmissions and packet re-orderings are handled by the transport layer. In addition to this, these works do implement or model the protocols exactly from the specification with some works only implementing SMTP partially. The specific challenges presented by the network link are not considered and the communication channel is considered only in an abstract manner. This also means that, unlike our implementation of TCP, the works are not shown to connect or work with existing protocol stacks, such as the kernel. To our best knowledge, ours is the first work to consider the implementation of transport layer protocols from their specification using session types.

In this paper, we have modelled TCP of the Internet protocol suite [7] using MPST [31] and implemented a proof of concept in the Rust programming language, leveraging the Rust type system and borrow checker to verify that the implementation complies with the session type. We have successfully tested our implementation using manual testing against the Linux kernel TCP stack as well as manually constructed TCP segments. In future work, we aim to address limitations of our implementation such as a lack of timeouts in the type system and the synchronous nature of our implementation. We additionally aim to model important aspects of the protocol such as congestion control in the future.

References

[1] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida & Fangyi Zhou (2022): *Generalised Multiparty Session Types with Crash-Stop Failures*. In Bartek Klin, Slawomir Lasota & Anca Muscholl, editors: *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, *LIPIcs* 243, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:25, doi:10.4230/LIPICS.CONCUR.2022.35.

[2] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos & Nobuko Yoshida (2019): *Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes*. In Luís Caires, editor: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, *Lecture Notes in Computer Science* 11423, Springer, pp. 583–610, doi:10.1007/978-3-030-17184-1_21.

[3] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos & Nobuko Yoshida (2019): *Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes*. In Luís Caires, editor: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, *Lecture Notes in Computer Science* 11423, Springer, pp. 583–610, doi:10.1007/978-3-030-17184-1_21.

[4] Scott O. Bradner (1996): *The Internet Standards Process – Revision 3*. RFC 2026, doi:10.17487/RFC2026. Available at `https://www.rfc-editor.org/info/rfc2026`.

[5] Matthew Alan Le Brun & Ornela Dardha (2023): *MAGπ: Types for Failure-Prone Communication*. In Thomas Wies, editor: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, *Lecture Notes in Computer Science* 13990, Springer, pp. 363–391, doi:10.1007/978-3-031-30044-8_14.

[6] Christian Bartolo Burlò, Adrian Francalanza & Alceste Scalas (2021): *On the Monitorability of Session Types, in Theory and Practice (Artifact)*. *Dagstuhl Artifacts Ser.* 7(2), pp. 02:1–02:3, doi:10.4230/DARTS.7.2.2.

[7] Wesley Eddy (2022): *Transmission Control Protocol (TCP)*. RFC 9293, doi:10.17487/RFC9293. Available at `https://www.rfc-editor.org/info/rfc9293`.

[8] Heather Flanagan (2019): *Fifty Years of RFCs*. RFC 8700, doi:10.17487/RFC8700. Available at `https://www.rfc-editor.org/info/rfc8700`.

[9] Simon Fowler (2016): *An Erlang Implementation of Multiparty Session Actors*. In Massimo Bartoletti, Ludovic Henrio, Sophia Knight & Hugo Torres Vieira, editors: *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016*, *EPTCS* 223, pp. 36–50, doi:10.4204/EPTCS.223.3.

[10] Simon Gay, Vasco Vasconcelos & António Ravara (2003): *Session Types for Inter-Process Communication*. Available at `https://www.dcs.gla.ac.uk/~simon/publications/TR-2003-133.pdf`.

[11] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 122–138, doi:10.1007/BFb0053567.

[12] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, Association for Computing Machinery, New York, NY, USA, pp. 273–284, doi:10.1145/1328438.1328472.

[13] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In Jan Vitek, editor: *ECOOP 2008 – Object-Oriented Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 516–541, doi:10.1007/978-3-540-70592-5_22.

[14] Thomas Bracht Laumann Jespersen, Philip Munksgaard & Ken Friis Larsen (2015): *Session Types for Rust*. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, Association for Computing Machinery, New York, NY, USA, pp. 13–22, doi:10.1145/2808098.2808100.

[15] Wen Kokke (2019): *Rusty Variation: Deadlock-free Sessions with Failure in Rust*. *Electronic Proceedings in Theoretical Computer Science* 304, pp. 48–60, doi:10.4204/eptcs.304.4.

[16] Wen Kokke & Ornela Dardha (2021): *Deadlock-free session types in linear Haskell*. In: *Haskell 2021: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell, Virtual Event, Korea, August 26-27, 2021*, ACM, pp. 1–13, doi:10.1145/3471874.3472979.

[17] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2016): *Typechecking protocols with Mungo and StMungo*. In: *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, ACM, pp. 146–159, doi:10.1145/2967973.2968595.

[18] Dimitrios Kouzapas, Ornela Dardha, Roly Perera & Simon J. Gay (2018): *Typechecking protocols with Mungo and StMungo: A session type toolchain for Java*. *Sci. Comput. Program.* 155, pp. 52–75, doi:10.1016/J.SCICO.2017.10.006.

[19] Nicolas Lagaillardie, Rumyana Neykova & Nobuko Yoshida (2022): *Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types (Artifact)*. *Dagstuhl Artifacts Ser.* 8(2), pp. 09:1–09:16, doi:10.4230/DARTS.8.2.9.

[20] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In Jan Vitek, editor: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, *Lecture Notes in Computer Science* 9032, Springer, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[21] Sam Lindley & J. Garrett Morris (2016): *Talking bananas: structural recursion for session types*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 434–447, doi:10.1145/2951913.2951921.

[22] Stephen McQuistin, Vivian Band, Dejice Jacob & Colin Perkins (2020): *Parsing Protocol Standards to Parse Standard Protocols*. In: *Proceedings of the Applied Networking Research Workshop*, Association for Computing Machinery, New York, NY, USA, p. 25–31, doi:10.1145/3404868.3406671.

[23] Stephen McQuistin, Vivian Band, Dejice Jacob & Colin Perkins (2021): *Investigating Automatic Code Generation for Network Packet Parsing*. In: *Proceedings of the IFIP Networking Conference*, pp. 1–9, doi:10.23919/IFIPNetworking52078.2021.9472829.

[24] Matthias Neubauer & Peter Thiemann (2004): *An Implementation of Session Types*. In Bharat Jayaraman, editor: *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004, Proceedings*, *Lecture Notes in Computer Science* 3057, Springer, pp. 56–70, doi:10.1007/978-3-540-24836-1_5.

[25] Nicholas Ng & Nobuko Yoshida (2016): *Static deadlock detection for concurrent go by global session graph synthesis*. In Ayal Zaks & Manuel V. Hermenegildo, editors: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, ACM, pp. 174–184, doi:10.1145/2892208.2892232.

[26] Nicholas Ng, Nobuko Yoshida & Kohei Honda (2012): *Multiparty Session C: Safe Parallel Programming with Message Optimisation*. In Carlo A. Furia & Sebastian Nanz, editors: *Objects, Models, Components, Patterns*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 202–218, doi:10.1007/978-3-642-30561-0_15.

[27] Luca Padovani (2017): *A simple library implementation of binary sessions*. *J. Funct. Program.* 27, p. e4, doi:10.1017/S0956796816000289.

[28] Vern Paxson (1997): *Automated packet trace analysis of TCP implementations*. In: *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 167–179, doi:10.1145/263105.263160.

[29] Riccardo Pucella & Jesse A. Tov (2008): *Haskell session types with (almost) no class*. In Andy Gill, editor: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, ACM, pp. 25–36, doi:10.1145/1411286.1411290.

[30] Pete Resnick (2014): *On Consensus and Humming in the IETF*. RFC 7282, doi:10.17487/RFC7282. Available at `https://www.rfc-editor.org/info/rfc7282`.

[31] Alceste Scalas & Nobuko Yoshida (2019): *Less is More: Multiparty Session Types Revisited*. *Proc. ACM Program. Lang.* 3(POPL), doi:10.1145/3290343.

[32] Scapy community: *Scapy*. `https://scapy.net/`.

A Appendix

A.1 Three-way handshake

The session type and the TCP state machine are initiated in the CLOSED state:

```
let st = ServerSystemSessionType::new();
let tcp = TcpClosed::new();
```

The *user* role calls the `Open` method and a TCB is created. This message is received using `offerone` by the system role:

```
let (_open, st) = system_user_channel.offer_one(st);
```

The system now transitions to the LISTEN state, waiting for a connection establishment to initiate, and sends a `TcbCreated` in response:

```
let tcp: TcpListen = tcp.open(LocalAddr { /* ... */ });
let st = system_user_channel.select_one(st, TcbCreated(()));
```

The next steps are to wait for a SYN segment from the network and respond with a SYN ACK segment. Once a SYN segment is received we transition to the SYN-RCVD state:

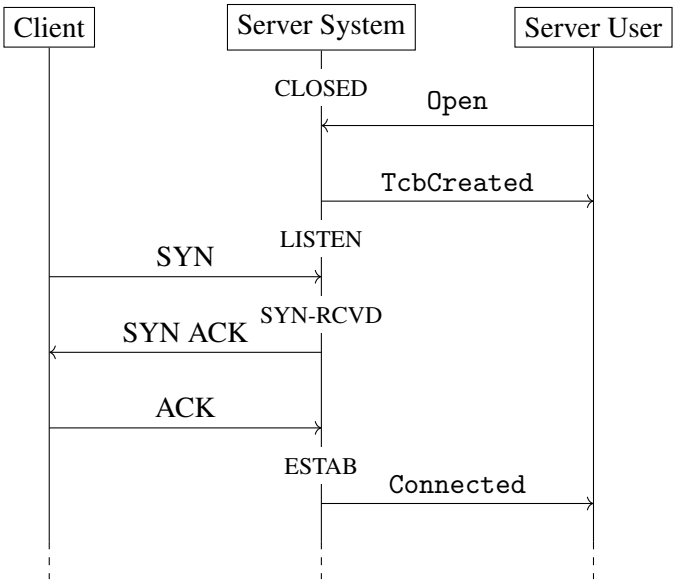


Figure 2: TCP three-way Handshake with all roles.

```
let (addr, syn, st) = net_channel.offer_one_with_addr(st, &tcp);

let (mut tcp /* Tcp<SynRcvd> */, synack) = tcp.recv_syn(addr, &syn);
let mut syn_rcvd = net_channel.select_one(st, addr, synack);
```

The recursive `SynRcvd` session type handles unacceptable acknowledgements of SYN ACK segments which need to be responded to with an ACK with the potential of a connection reset:

```
let (mut tcp, st) = loop {
    let st = syn_rcvd.inner();
    let tcp_for_picker = tcp.for_picker();
```

The `offer_two_filtered` method can now be called on the network channel. This method takes the session type, a picker function, and a channel filter:

```
    match net_channel.offer_two_filtered(
        st,
        |packet| match tcp_for_picker.acceptable(&packet) {
            ReactionInner::Acceptable(_, _) => Branch::Left(
                packet.into()),
            _ => Branch::Right(packet.into()),
        },
        &tcp,
    ) {
```

Note that the picker determines which branch to take based on the TCP state machine.

The left branch of the session type corresponds to an acceptable ACK segment:

```
        Branch::Left((acceptable, st)) => {
            let tcp: Tcp<Established> = tcp
```

```
                .recv_ack(&acceptable)
                .empty_acceptable()
                .expect("First␣ACK␣must␣be␣empty");
            break (tcp, st);
        }
```

However, if the ACK is not acceptable, the right branch is taken – an ACK is either sent back and the system waits for another ACK:

```
        Branch::Right((unacceptable, st)) => {
            let remote_addr = tcp.remote_addr();
            match tcp.recv_ack(&unacceptable) {
                Reaction::NotAcceptable(tcp2, Some(response)) => {
                    let st = net_channel.select_left(
                        st, tcp2.remote_addr(), response);
                    syn_rcvd = st;
                    tcp = tcp2;
                    continue;
                }
```

Alternatively, an RST, notifying the user that the connection is being reset:

```
                Reaction::Reset(Some(rst)) => {
                    let st = net_channel.select_right(
                        st, remote_addr, rst);
                    let end = system_user_channel.select_one(
                        st, Close(()));
                    net_channel.close(end);
                    system_user_channel.close(end);
                    return;
                }
```

Finally, once out of the loop, the implementation is in the ESTAB state and the system notifies the user that the connection is established:

```
let mut recursive = system_user_channel.select_one(st, Connected(()));
info!("established");
```

This concludes the implementation of the three-way handshake.

A.2 Exchanging data

The main loop of the implementation waits to receive a segment (using an `Offer` session type) and branches based on their type and whether it is acceptable or not.

```
Rec!(pub ServerSystemCommLoop, [
    (RoleClientSystem & {
```

Acceptable with payload where an acceptable segment is received and there is data present:

```
    Ack.
        (RoleClientSystem + Ack /* empty */).
        (RoleServerUser + Data).
        (RoleServerUser & {
            Data.
                (RoleClientSystem + Ack /* with data */).
                ServerSystemCommLoop,
            Close.
                (RoleClientSystem + FinAck).
                ServerSystemFinWait1
        }),
```

Initially, data acknowledgement is accomplished using an empty ACK segment. The data contained within the ACK segment is then transmitted to the server user within a message of type `Data`. The user has the option to respond by sending back a message, leading to the transmission of an ACK with payload. Alternatively, the user may choose to initiate the closure of the connection, resulting in the transmission of a FIN ACK.

Acceptable without payload In the case where these segments are acknowledgements of previously sent segments, we pass the ACK to the TCP state machine to update the state and update the retransmission queue:

```
Ack.ServerSystemCommLoop,
```

FIN ACK The peer has initiated closing the connection. In this case, the TCP state machine will transition from ESTABLISHED to the CLOSE-WAIT state. Note that due to the absence of timeouts in the type system, the timeout in the close-wait state is implemented outside the session typed state machine.

```
FinAck.
    (RoleClientSystem + Ack /* we ACK the FIN */).
    (RoleServerUser + Close).
    ServerSystemCloseWait,
```

Unacceptable A segment where the sequence numbers are not acceptable has been received. The server will respond with an ACK which serves to inform the peer about the current receive window start and length [7].

```
    Ack.
        (RoleClientSystem + Ack).
        ServerSystemCommLoop,


    })
]);
```

Behavioural Types for Heterogeneous Systems (Position Paper)

Simon Fowler
University of Glasgow, UK

Philipp Haller
Digital Futures, KTH Royal Institute of Technology, SE

Roland Kuhn
Actyx AG, DE

Sam Lindley
The University of Edinburgh, UK

Alceste Scalas
Technical University of Denmark, DK

Vasco T. Vasconcelos
University of Lisbon, PT

Behavioural types provide a promising way to achieve lightweight, language-integrated verification for communication-centric software. However, a large barrier to the adoption of behavioural types is that the current state of the art expects software to be written using *the same* tools and typing discipline throughout a system, and has little support for components over which a developer has no control.

This position paper describes the outcomes of a working group discussion at Dagstuhl Seminar 24051 (Next-Generation Protocols for Heterogeneous Systems). We propose a methodology for integrating multiple behaviourally-typed components, written in different languages. Our proposed approach involves an *extensible protocol description language*, a *session IR* that can describe data transformations and boundary monitoring and which can be compiled into program-specific *session proxies*, and finally a *session middleware* to aid session establishment.

We hope that this position paper will stimulate discussion on one of the most pressing challenges facing the widespread adoption of behavioural typing.

1 Introduction

Behavioural types provide a powerful and lightweight mechanism for language-integrated verification of behavioural properties: whereas traditional data types rule out errors such as adding an integer to a string, behavioural types can rule out behavioural errors such as forgetting to close a file handle or sending an invalid message on a communication channel.

Session types [20, 21] are a behavioural typing discipline for checking adherence to communication protocols: if a process is typed according to its session type, then it is guaranteed to fulfil its role in the communication protocol at runtime. Although originally designed for two communicating participants, work on *multiparty* session types (MPSTs) [22] extends session typing to handle systems with multiple components. If all components are either derived from a well-formed global type, or all components have compatible local types, then the entire system should not encounter any communication errors at runtime. Many MPST disciplines include further guarantees such as global progress or liveness.

Extensive research has gone into behavioural types, in particular session types, over the years. A recent workshop celebrated 30 years of session types [2], and a recent book concentrates on how decades of research into behavioural types has given rise to a plethora of tools [18]. Many international programming languages conferences typically have sessions dedicated to behavioural type systems.

Nevertheless, and notwithstanding efforts to overcome barriers to practical adoption (e.g., work on failure handling [7, 17], graphical user interfaces [16, 32], and subsessions [13]), behavioural typing has not yet seen widespread industrial use. Arguably *the* key issue facing the behavioural types community[1] is

[1] And indeed, other neighbouring communities such as choreographic programming [33], although we concentrate on behavioural types in this paper.

D. Costa, R. Hu (Eds.): Programming Language Approaches to Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. 37–48, doi:10.4204/EPTCS.401.4

the inability for behavioural types to work satisfactorily with *heterogeneous* systems, consisting of software components developed in different languages, using different tools, with different typing guarantees.

Heterogeneity may arise for practical reasons: for example, we may want to write some components in a given language due to better library support (e.g., using Python due to its rich support for data science), or because it is important to obtain stronger static guarantees for a particular participant or portion of a protocol. Heterogeneity may even arise in a single program, for example writing parts of an application in different programming languages (e.g., writing performance-critical code in a systems language such as C++ or Rust, and the remainder of the program in a managed language like Python). Furthermore, in any realistic setting, we would have to assume that some components are implemented using different languages or accessible only through an API (for example as is common with microservices). Similarly, we may have existing components that offer *similar* services that do not quite correspond to the expected types. We concentrate on the following scenario:

> **Scenario: Travel Booking System.** We want to design a travel booking system that includes both timing constraints and data refinements. A travel booking session consists of a client, travel agent, and flight provider; we are in control of the agent and client, but the provider is developed by an external company and accessible via an API. The client initiates a search, and receives a set of suitable flights. After receiving the results, the client has 6 minutes to select a flight before the results expire.
>
> When the client selects a flight, it receives a token, and should send the same token to the provider in order to continue the booking. Finally, the provider sends the client either a booking confirmation, or an error message.

Crucially, we will consider a heterogeneous version of this scenario in which the client, travel agent, and flight provider are implemented using different tools with quite different capabilities.

Paper structure. The paper proceeds as follows. Section 2 gives relevant background. Section 3 describes a motivating scenario of a travel agent application written in different tools with differing language features. Section 4 describes our proposed solution and speculates on several potential research challenges. Section 5 discusses related work, and Section 6 concludes.

2 Background

(Multiparty) Session Types. Session types are types for protocols: whereas a data type describes the shape of some data, ruling out errors such as adding an integer to a string, a session type describes both the type and direction of data to be communicated between participants [20, 21]. Session types were originally investigated in the *binary* setting between two participants, but later work on *multiparty* session types [22] describes communication between multiple communicating participants. We concentrate on multiparty session typing in the remainder of the paper.

The following example describes the classic *two-buyer* protocol where two participants collaborate in order to buy an expensive item (usually a book). `Buyer1` begins by sending the title to the `Seller`, who responds with a quote. `Buyer1` then sends the quote to `Buyer2`, who decides whether to accept the quote by sending their address to the `Seller` and subsequently receiving a delivery data, or declining the offer.

The *global type* describing all interactions in the system is described on the left. Global types can then be *projected* into *local types* for each participant; it is then possible to typecheck or monitor all participants against their local types. The local type for `Buyer2` is shown on the right.

```
Buyer1 → Seller : title(String) .
Seller → Buyer1 : quote(Int) .
Buyer1 → Buyer2 : share(Int) .
Buyer2 → Seller : {
  address(String) .
    Seller → Buyer2 : date(Date) . end,
  quit(Unit) . end
}
```

```
Buyer2 ≜ Buyer1 & share(Int) .
         Seller ⊕{
           address(String) .
             Seller & date(Date) . end,
           quit(Unit) . end
         }
```

A multitude of tools have been developed for checking against multiparty session types, for example in Java [28], Scala [40], Rust [12, 29], and F# [37]. However, each tool embeds the assumption that the entire system is written using that same tool; the possibility of combining heterogeneous components written using different tools and programming languages is not part of the tools' specification.

Runtime Monitoring against Session Types. Although the original work on session typing envisaged static checking, a correspondence between MPSTs and communicating automata [14] showed how it was possible to *monitor* processes against a session type, allowing a degree of runtime verification. The key idea is to translate a local type into a finite state machine where transition corresponds to a send or receive action; for each session endpoint, a monitor process observes incoming and outgoing messages — and upon receiving an invalid message, the monitor drops it and/or reports a violation. We describe runtime monitoring against session types in more depth in Section 5.

Multi-language Interoperability. There has been increasing attention given to semantic foundations for multi-language interoperability, for example through foreign function interfaces (FFIs). A major inspiration for our proposed solution is the approach of Patterson et al. [38] who introduce a methodology for interoperability by defining a common intermediate representation along with *convertibility relations* and *boundary conversions* and show how to verify semantic soundness using logical relations.

Our goal is to adopt an analogous methodology but for the world of message passing as opposed to shared memory. Rather than challenges such as linking and foreign function interfaces, our challenge is to describe the "glue" that can allow a program written in Go, for example, to safely interact with a program written in Java or Rust, while keeping as many of the guarantees that we would expect if a program was written in a single behaviourally-typed language.

3 Heterogeneous Multiparty Session Typing

We can start by writing an idealised global type (omitting some irrelevant timing constraints):

```
Customer → Agent : search(origin : AirportName, destination : AirportName) .
Agent → Customer : results(searchResults : [(FlightNum, Time, Price, Provider)]) .
Customer → Agent : {
  select{t ≤ 360}(flightNum : FlightNum) ↦
    Agent → Customer : providerRef(ref : ProviderRef) .
    Customer → Provider : book{token == ref}(token : ProviderRef, details : PaymentDetails) .
    Provider → Customer : {
      ok() ↦ end,
      error() ↦ end
    },
  timeout{t > 360}() ↦
    Customer → Provider : timeout() . end
}
```

Note that the clock at the customer can only send a select message within 360 seconds, and otherwise must send a timeout message. Similarly, the data refinement on the book message ensures that the *same* reference is sent to the provider as is received from the agent. Say that our system consisted of:

- The `Agent` in Python using the time-aware framework proposed by Neykova et al. [36]
- The `Customer` in F⋆ using SESSION⋆ by Zhou et al. [46] to statically verify the data refinement
- The `Provider`, developed by a different company and accessible only through an API

In the above, the `Agent` has inbuilt verification of timing constraints, the `Customer` has static verification of data refinements, and the `Provider` does not even have verification of communication patterns. There are several research problems posed by this scenario:

Extensibility. At present there are different, incompatible, *dialects* of session types for each extension.

Precision mismatch. However, each tool available to implement a constraint (Python for timing constraints; SESSION⋆ for data refinement) only works in a single language. Thus, some checks must take place at runtime using boundary monitors.

Message rejection. Existing work on monitoring against multiparty session types takes a *suppression-based* approach to monitoring: a monitor will drop any non-conforming messages that it receives (either silently, or reporting a violation). Although suppression maintains safety (by stopping any non-conforming messages from being processed by the program logic), dropping a message may mean that the remaining actions in the protocol cannot be fulfilled—thus breaking liveness.

As well as research questions, there are also some more practical issues:

Session Initiation. The session needs to be *established*, which involves discovering each component, inviting it to the session, and setting up the communication infrastructure.

Wire format. Communication needs to occur over a *standard message layout*. Most session typing systems either do not include any wire communication, or communicate using non-standard message layouts that vary per tool.

4 Proposed Solution

Figure 1 gives an overview of our proposed approach. The overall idea is:

- A developer designs the protocol in an *extensible, language-agnostic protocol description language*
- The protocol is projected and compiled down into a program-specific *session IR*, whose purpose is to describe any necessary dynamic checks, message re-orderings, and data transformations
- The session IR is used to generate *session proxies* that act as adapters to each program

Extensible Protocol Description Language The first step is to write a protocol in a language-agnostic protocol description language. The *Scribble* protocol description language [44, 45] provides a good starting point, but the key point is that the language should be extensible in a modular way by supporting *plugins* supporting individual language features (e.g., value-dependency or timeouts).

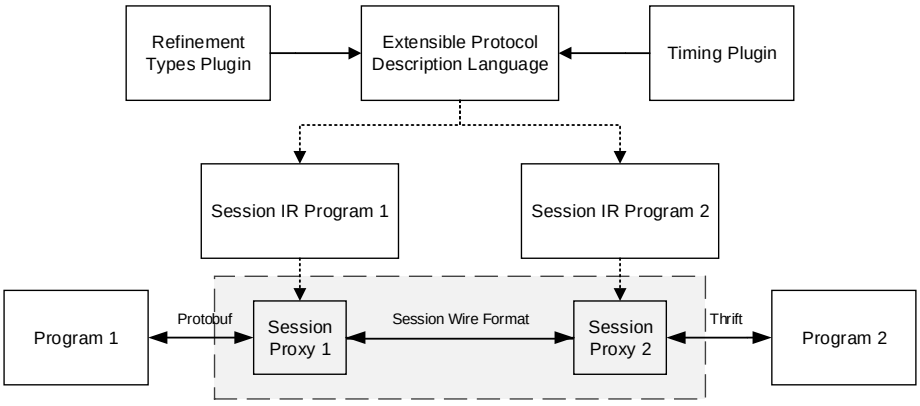


Figure 1: Proposed System

Session IR In traditional multiparty session programming, a global protocol description is projected as a *local type* for each participant. The local type serves as a language-agnostic *specification* for the communication actions that the participants should perform, and can be used for static type checking.

In addition to local types for static checking, we propose a *session intermediate representation* (Session IR). Whereas local types are meant to be implementation-agnostic, the Session IR is used to describe any monitoring or message transformations required in order to integrate a component with the system. In particular, we envisage the session IR being able to support at least the following:

- **Boundary Monitoring** To address the precision mismatches between the static checking supported by the tool, as well as maintaining timing properties, a core role of each session proxy is to perform boundary monitoring [8, 11] before an incoming message is delivered (and in some circumstances before an outgoing message is committed to the system). Monitor violations may result in *suppression* (i.e., dropping messages that are violated), but may also allow violations to be reported to the program in order to allow compensatory actions (e.g., raising an exception or requesting that the sender provides a revised message).
- **Message Insertion / Reordering** It may be that a component supports a compatible variation of its role in a protocol, but does not match the protocol exactly (for example, due to a version upgrade). In this case, the session IR can describe message reorderings (inspired by theoretical work on session type isomorphisms [5, 15]), or insertions of messages at a given point. This would enable migrating components to new versions with a different but compatible behaviour.
- **Wire Formatting** Each program will have its own expected communication mechanism (be that sockets, protocol buffers [19], or interface description languages like Thrift [3]). The final job of the session IR will be to describe transformations between the internal protocol representation and the wire protocol.

For trusted components that are statically checked to follow the protocol, the session proxy will only need to perform wire formatting and monitoring of incoming messages.

Session Proxies. Each session IR program can then be compiled into a *session proxy* to interact with each program. The session proxy acts as an adapter between the program and the other participants in the session. Interaction between session proxies happens using a standardised session wire format and communication medium.

There are several ways by which session proxies could be integrated with each application. We expect that for basic suppression-based monitors, it would suffice to leave the original application untouched and instead route all communication through the proxy (indeed, this would also support a level of session typing for components that are not programmed with session types). For more involved session proxies such as those that raise an application-level exception when a message violation occurs, it would likely be necessary to either adapt the current tooling to incorporate the session proxy or provide an API that a developer can code against.

Language Features. Another concern is the features that must be available to each language or tool in order for it to support any operations required by each session proxy. For suppression-based monitors, it is unlikely that any additional language features would be needed. If we would like to report any monitor violations to the program, however, then we would likely need some additional language support: for example, exception handling [17] in the case of needing to deal with linear resources, or additional failure handling callbacks in the case of a tool based around inversion-of-control [46].

Session Establishment. Figure 1 does not describe how sessions are established. We envisage a middleware application, similar to that described by Atzei et al. [6], is a potential solution: such a system would allow participants to register to take part in a session, discover other participants, and finally allow sessions to be established. Alternatively, session establishment could happen directly (as is done, for example, with explicit connection actions [24]).

4.1 Potential Challenges

Although we believe our proposed solution offers a promising framework for future research on heterogeneous session typing, we envisage several challenges, at least including the following:

Interactions between extensions. Our scenario considers two fairly orthogonal extensions: data refinements and timing. However, there could be other extensions (e.g., explicit connection actions [24] that require a more liberal syntax) that could pose challenges when combined with existing disciplines. How can we ensure that the extensible language is sufficiently general to both mediate between the different dialects of session typing, and how can we ensure that their combination does not lead to safety errors?

Formal guarantees. It is important to understand the desired guarantees to be given by the system. It seems reasonable to expect, at a minimum, that the system will ensure session fidelity (i.e., that every message that is exchanged will conform to the given protocol). Nevertheless, ensuring properties such as liveness is more challenging in a monitored setting. A further open challenge would be reasoning about the metatheory in a modular way.

Generality of the IR. The Session IR will at least need to be able to describe monitoring, message reordering, and message reformatting. However, there is a large design space and there are inherent trade-offs to ensuring the IR design remains sufficiently high-level while also sufficiently general to support the array of possible extensions.

Performance. Any runtime checking and message rewriting will inevitably incur a runtime overhead, so it will be necessary to ensure that any overhead is not prohibitive. This can be mitigated to an extent by only checking properties that are not guaranteed statically, and ensuring that the monitor is located on the same machine as the monitored process. In addition to runtime overheads, it is

possible that monitors may need to record some message history, for example to enforce dependent type constraints [41]. It is therefore necessary to ensure that any generated monitor does not require unbounded space.

Location of error reporting. In addition to the language design challenges in allowing applications to handle any errors, it is possible that there are multiple places to report a violation. Some notion of blame [43] is likely to be important, but defining "more" or "less" typed is likely to be challenging in the presence of multiple extensions.

5 Related Work

Protocol description languages. MPSTs were designed without a particular implementation in mind. A good starting point for heterogeneity is the language-agnostic Scribble protocol description language [35, 44] for describing MPST specifications; implementations of Scribble (e.g., [45]) support well-formedness checking, projection, and monitor generation. A necessary step in the pursuit of heterogeneity would be to generalise a language like Scribble to allow modular and composable extensions with different language features, as opposed to the current status quo of *ad-hoc* extensions for each new feature.

Monitoring against MPSTs. Most closely relevant is work on runtime monitoring against local types [8, 11], based on the correspondence between multiparty session types and communicating automata [14]. The core idea is to translate each local type into an FSM and check each incoming and outgoing message against the monitor, rejecting the message if the message does not match any transition. In particular, Bocchi et al. [8] show *safety* (monitors ensure that ill-behaved participants do not send messages that violate their specification) and *transparency* (monitors do not affect the behaviour of well-behaved components). However, these monitors discard non-conforming messages without any feedback to the sender or receiver. In turn, this means that (especially for non-recursive protocols), non-conforming messages cause the protocol to silently stop. Later work by van den Heuvel et al. [42] addresses the black-box monitoring of multiparty sessions through monitoring processes that are directly generated from global types, and (unlike Bocchi et al. [8]) actively report violations by stopping execution. Burlò et al. [9] proposes a prototype implementation of a "hybrid" verification approach where session-typed components can interoperate with heterogeneous (possibly untyped) components through autogenerated monitors — which, in turn, are session-typed (only for two-party sessions), and can translate messages between different wire formats, and suppress and report protocol-violating messages; Burlò et al. [10] later studies the properties of such black-box monitors in terms of soundness and completeness of violation reports. In contrast to all the works on session monitoring listed above, we believe that it may be necessary to forego monitor transparency and instead allow compensatory behaviours upon a monitor violation (for example, raising an exception or requiring the sender to send a revised message).

Session types and heterogeneity. Only very little work has attempted to address heterogeneous session typing. Jongmans and Proença [27] describe the design of a system called ST4MP that aims to support multi-lingual programming through the established API generation approach [23]; the idea is to generate multiple compatible APIs for different languages from a given global type specification. In contrast to this position paper, ST4MP supports a base multiparty session typing discipline without any advanced language features (e.g., timing or refinement types), and does not make use of any form of runtime checking. In contrast we would expect our general session IR to be able to support even untyped components.

Language and system interoperability. Our proposal to use a common Session IR to provide safety properties even when composing heterogeneous components written in different languages is similar in spirit to recent work on sound language interoperability [38, 39]. While previous work only targets languages interoperating via shared memory, our proposal specifically aims to address interoperability for typed message-passing concurrency. Gradual session types [25] provide a framework for ensuring type and communication safety for programs integrating statically-typed sessions and dynamic types. It is assumed that programs share a common internal language with casts. Our proposal aims to decouple components even further by mediating communication via Session IR proxies which may, in addition to casts, insert or reorder messages and perform other forms of monitoring. Session IR and Session IR proxies are related to IDLs used for integrating distributed components [30] as well as systems for business-to-business interactions [31] which explicitly aim to address heterogeneity.

At a more abstract level, the ideas presented in this position paper can be related to another position paper by Albert *et al.* [4] that advocates a formal language for service-level agreements (SLAs) for (virtualised and heterogeneous) distributed services, to be enforced via e.g. static verification and/or runtime monitoring, possibly aided by code generation from executable models written in ABS [26]. Our approach is focused on behavioural types (which could be seen both as a form of formalised SLA) and as a form of executable specification (usable e.g. for monitor generation via the session IR).

Jolie [1, 34] is a service-oriented programming language. Programs can either be written in Jolie, or Jolie can serve as an interface to services written in a different programming language. Jolie programs can also serve as *orchestrators* to interact with multiple other services, potentially using different transport mechanisms. In contrast, our proposal takes a more protocol-centric approach: instead of specifying the services and an orchestration-based method of allowing them to interact, our proposal instead involves concentrates on boundary monitoring and manipulation of existing communication flows. An advantage is that we can (potentially statically) verify fine-grained data and timing constraints.

6 Conclusion

Although behavioural types offer a strong foundation for lightweight, language-integrated verification of behavioural properties, a large barrier to their adoption is that at present an *entire system* must be written in a single language. In this position paper we have described a potential line of work that, if completed successfully, could allow behavioural types in *heterogeneous* software systems where components can be written in different languages, using different tools, each of which support different static guarantees. Our approach relies on an *extensible* protocol description language that can support additional language features (e.g., timing or refinement types) as plugins, and a *session IR* that can describe transformations on data (e.g., wire formatting, message reordering, and boundary monitoring). Structured support for heterogeneity can greatly expand the reach of behavioural types in real-world systems, and we hope that these initial ideas serve as a starting point for addressing this challenging research topic.

Acknowledgements

We thank the organisers of Dagstuhl Seminar 24051 and Schloss Dagstuhl — Leibniz Center for Informatics for making this work possible. We are also grateful to the anonymous reviewers for their detailed and encouraging reviews. This work was supported by EPSRC grant EP/T014628/1 (STARDUST), Horizon Europe grant 101093006 (TaRDIS), Independent Research Fund Denmark RP-1 grant “Hyben”, Digital Futures Research Pairs Consolidator Project “PORTALS”, and UKRI Future Leaders Fellowship MR/T043830/1 (EHOP), FCT grant PTDC/CCI-COM/6453/2020 (SafeSessions), and the LASIGE Research Unit.

References

[1] Jolie programming language — official website. URL `https://www.jolie-lang.org/`. Accessed on 25/03/2024.

[2] ST30: 30 years of session types — workshop website. URL `https://2023.splashcon.org/home/st-anniversary-30`. Accessed on 11/01/2024.

[3] Apache Thrift - official website. URL `https://protobuf.dev/`. Accessed on 11/01/2024.

[4] Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, NordiCloud '13, page 59–63. Association for Computing Machinery, 2013. doi: 10.1145/2513534.2513545.

[5] Assel Altayeva and Nobuko Yoshida. Service equivalence via multiparty session type isomorphisms. In *PLACES@ETAPS*, volume 291 of *EPTCS*, pages 1–11, 2019. doi: 10.4204/eptcs.291.1.

[6] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Contract-oriented programming with timed session types. *Behavioural Types: from Theory to Tools*, page 27, 2017. doi: 10.1201/9781003337331-2.

[7] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised multiparty session types with crash-stop failures. In *CONCUR*, volume 243 of *LIPIcs*, pages 35:1–35:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.CONCUR.2022.35.

[8] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017. doi: 10.1016/j.tcs.2017.02.009.

[9] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. Towards a hybrid verification methodology for communication protocols (short paper). In *FORTE*, volume 12136 of *Lecture Notes in Computer Science*, pages 227–235. Springer, 2020. doi: 10.1007/978-3-030-50086-3_13.

[10] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. On the monitorability of session types, in theory and practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 20:1–20:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPICS.ECOOP.2021.20.

[11] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 7173 of *Lecture Notes in Computer Science*, pages 25–45. Springer, 2011. doi: 10.1007/978-3-642-30065-3_2.

[12] Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in Rust with multiparty session types. In *PPoPP*, pages 246–261. ACM, 2022. doi: 10.1145/3503221.3508404.

[13] Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi: 10.1007/978-3-642-32940-1_20.

[14] Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. doi: 10.1007/978-3-642-28869-2_10.

[15] Mariangiola Dezani-Ciancaglini, Luca Padovani, and Jovanka Pantovic. Session type isomorphisms. In *PLACES*, volume 155 of *EPTCS*, pages 61–71, 2014. doi: 10.4204/eptcs.155.9.

[16] Simon Fowler. Model-view-update-communicate: Session types meet the Elm architecture. In *ECOOP*, volume 166 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ECOOP.2020.14.

[17] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi: 10.1145/3291617.

[18] Simon Gay and António Ravara. *Behavioural Types: from Theory to Tools*. River Publishers, 2017.

[19] Google. Protocol buffers documentation. URL `https://protobuf.dev/`. Accessed on 11/01/2024.

[20] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi: 10.1007/3-540-57208-2_35.

[21] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi: 10.1007/BFb0053567.

[22] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008. doi: 10.1145/1328438.1328472.

[23] Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2016. doi: 10.1007/978-3-662-49665-7_24.

[24] Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017. doi: 10.1007/978-3-662-54494-5_7.

[25] Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *J. Funct. Program.*, 29:e17, 2019. doi: 10.1017/S0956796819000169. URL `https://doi.org/10.1017/S0956796819000169`.

[26] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010. doi: 10.1007/978-3-642-25271-6_8.

[27] Sung-Shik Jongmans and José Proença. ST4MP: A blueprint of multiparty session typing for multilingual programming. In *ISoLA (1)*, volume 13701 of *Lecture Notes in Computer Science*, pages 460–478. Springer, 2022. doi: 10.1007/978-3-031-19849-6_26.

[28] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for Java. *Sci. Comput. Program.*, 155:52–75, 2018. doi: 10.1016/j.scico.2017.10.006.

[29] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay safe under panic: Affine Rust programming with multiparty session types. In *ECOOP*, volume 222 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ECOOP.2022.4.

[30] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Comput. Surv.*, 30(1):3–27, 1998. doi: 10.1145/274440.274441. URL https://doi.org/10.1145/274440.274441.

[31] Brahim Medjahed, Boualem Benatallah, Athman Bouguettaya, Anne H. H. Ngu, and Ahmed K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. *VLDB J.*, 12(1):59–85, 2003. doi: 10.1007/S00778-003-0087-Z. URL https://doi.org/10.1007/s00778-003-0087-z.

[32] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in TypeScript with routed multiparty session types. In *CC*, pages 94–106. ACM, 2021. doi: 10.1145/3446804.3446854.

[33] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. doi: 10.1017/9781108981491.

[34] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014. doi: 10.1007/978-1-4614-7518-7_4.

[35] Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. In *Models, Languages, and Tools for Concurrent and Distributed Programming*, volume 11665 of *Lecture Notes in Computer Science*, pages 236–259. Springer, 2019. doi: 10.1007/978-3-030-21485-2_14.

[36] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.*, 29(5):877–910, 2017. doi: 10.1007/s00165-017-0420-8.

[37] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In *CC*, pages 128–138. ACM, 2018. doi: 10.1145/3178372.3179495.

[38] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. Semantic soundness for language interoperability. In *PLDI*, pages 609–624. ACM, 2022. doi: 10.1145/3519939.3523703.

[39] Daniel Patterson, Andrew Wagner, and Amal Ahmed. Semantic encapsulation using linking types. In *TyDe@ICFP*, pages 14–28. ACM, 2023. doi: 10.1145/3609027.3609405.

[40] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPICS.ECOOP.2017.24. URL https://doi.org/10.4230/LIPIcs.ECOOP.2017.24.

[41] Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. *J. Log. Algebraic Methods Program.*, 90:61–83, 2017. doi: 10.1016/j.jlamp.2016.11.005.

[42] Bas van den Heuvel, Jorge A. Pérez, and Rares A. Dobre. Monitoring blackbox implementations of multiparty session protocols. In *RV*, volume 14245 of *Lecture Notes in Computer Science*, pages 66–85. Springer, 2023. doi: 10.1007/978-3-031-44267-4_4.

[43] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009. doi: 10.1007/978-3-642-00590-9_1.

[44] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi: 10.1007/978-3-319-05119-2_3.

[45] Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In *FCT*, volume 12867 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2021. doi: 10.1007/978-3-030-86593-1_2.

[46] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020. doi: 10.1145/3428216.

Three Subtyping Algorithms for Binary Session Types and their Complexity Analyses

Thien Udomsrirungruang
University of Oxford, Oxford, UK
thien.udomsrirungruang@keble.ox.ac.uk

Nobuko Yoshida
University of Oxford, Oxford, UK
nobuko.yoshida@cs.ox.ac.uk

Session types are a type discipline for describing and specifying communication behaviours of concurrent processes. *Session subtyping*, firstly introduced by Gay and Hole [3], is widely used for enlarging typability of session programs. This paper gives the complexity analysis of three algorithms for subtyping of synchronous binary session types. First, we analyse the complexity of the algorithm from the original paper, which is based on an inductive tree search. We then introduce its optimised version, which improves the complexity, but is still exponential against the size of the two types. Finally, we propose a new quadratic algorithm based on a graph search using the concept of $\mathscr{X}\mathscr{Y}\mathscr{Z}\mathscr{W}$-simulation, recently introduced by Silva *et al.* [11].

1 Introduction

Session types [5, 12] are a type discipline for describing and specifying communication behaviours of concurrent processes. They stem from the observation that many real-world processes communicate in a highly-structured manner; these communications may include sending and receiving messages, selection and branching from a set of labels, and recursion. These session types allow programs to be validated for type safety using typechecking algorithms.

With regards to the synchronous session type system, Gay and Hole [3] proposed a subtyping scheme, and proved the soundness of the type system; Chen *et al.* [2] later proved that the type system was complete (i.e. there is no larger subtyping relation between session types that is sound). Thus this notion of subtyping is of interest as it is most general. Our presentation of session types will use this system, as described in Gay and Hole's paper [3]. For simplicity, we omit the *channel type* $\hat{}[S]$: they can be adopted into the algorithms with little issue.

As an example, consider the following type, which represents an interface to a server which is able to respond to pings, as well as terminate itself:

$$T_1^{\mathsf{interface}} = \mu X.\, \oplus\langle \mathsf{respond}\colon ?[\mathsf{end}]; X, \mathsf{exit}\colon \mathsf{end}\rangle \tag{1}$$

Here, the channel offers two choices of label to the process using it: (1) choosing respond then receiving a message of unit type end, or (2) choosing exit and terminating the channel. After performing (1), the μ-recursion means that the channel reverts to its original state.

Next, consider the scenario where we upgrade the interface to the server such that it can be replicated.

$$T_2^{\mathsf{interface}} = \mu X.\, \oplus\langle \mathsf{respond}\colon ?[\mathsf{end}]; X, \mathsf{exit}\colon \mathsf{end}, \mathsf{replicate}\colon ?[X]; X\rangle \tag{2}$$

The interface offers another choice: (3) choosing replicate and receiving a new channel with the same type $T_2^{\mathsf{interface}}$. We can immediately see that any program that expects a channel of type $T_1^{\mathsf{interface}}$ will also work on a channel of type $T_2^{\mathsf{interface}}$. This is because the functionality of the new channel type

D. Costa, R. Hu (Eds.): Programming Language Approaches to
Concurrency and Communication-cEntric Software 2024 (PLACES'24)
EPTCS 401, 2024, pp. 49–60, doi:10.4204/EPTCS.401.5

is identical to the old type, as long as the replicate label is never chosen. Thus, it would make sense to accept $T_2^{\mathsf{interface}}$ in type checking. It turns out that $T_2^{\mathsf{interface}}$ is a subtype of $T_1^{\mathsf{interface}}$ (denoted $T_2^{\mathsf{interface}} \leq_c T_1^{\mathsf{interface}}$), which characterises this property.

This subtyping relation is easy to see with a *syntactic* approach: the shapes of the types are almost identical, except for one additional branch in $T_2^{\mathsf{interface}}$; we could then argue inductively that the subtyping relation holds. However, for some types, this is not possible:

$$T_3^{\mathsf{interface}} = \mu Y.\, \oplus\langle \mathsf{respond} : ?[\mathsf{end}]; Y, \mathsf{exit} : \mathsf{end}, \mathsf{replicate} : ?[T_1^{\mathsf{interface}}]; Y \rangle \tag{3}$$

In essence, $T_3^{\mathsf{interface}}$ is an interface that can be replicated to get copies, but those copies cannot themselves be replicated. We have that $T_2^{\mathsf{interface}} \leq_c T_3^{\mathsf{interface}}$. Syntactic subtyping does not work because the types do not have the same shapes (namely, the messages sent in the replicate branch are X and $T_1^{\mathsf{interface}}$ respectively). The difficulty of checking subtyping comes from the ability to "branch off" into multiple subterms: in the above example, checking the subterm $?[T_1^{\mathsf{interface}}]; Y$ would require us to check both $T_1^{\mathsf{interface}}$ and Y. In certain cases this can lead to exponential complexity in the standard inductive algorithm given in [3] (see Example 3.8).

Contributions. We first analyse the algorithm given in Gay and Hole's original paper on session type subtyping [3, §5.1], improving the bound given in [8, §5.1]. Second, we propose its optimised version, in the style of [10, Fig. 21-4]. Both of these algorithms have worst-case exponential complexity, with the second algorithm having better complexity than the first. Third, we represent types as labeled transition systems, and formulate the subtyping problem as checking an $\mathscr{X}\mathscr{Y}\mathscr{Z}\mathscr{W}$-simulation, similarly to [11]; we then give a quadratic algorithm for subtyping by checking the validity of the simulation. A full version of this paper is available [13], which contains full versions of definitions and examples.

2 Preliminaries

We restate the definitions given in [3], which are used in this paper.

Definition 2.1 (Session Types)**.** Session types (denoted $S, S', T, U, V, W, \dots$) are defined by the following grammar:

$S ::= \mathsf{end}$	(inaction)	$\mid \oplus\langle l_i : S_i \rangle_{1 \leq i \leq n}$	(selection)	$\mid \mu X.S$	(recursive type)
$\mid ?[S_1, \dots, S_n]; S$	(input)	$\mid \&\langle l_i : S_i \rangle_{1 \leq i \leq n}$	(branching)	$\mid X$	(type variable)
$\mid ![S_1, \dots, S_n]; S$	(output)				

$X, X_i, Y, Z, \dots$ range over a countable set of type variables. We require that all terms are contractive, i.e. $\mu X_1.\mu X_2. \dots \mu X_n.X_1$ is not allowed as a subterm for any $n \geq 1$. As shorthand, we use $?[\tilde{T}]; S$ instead of $?[T_1, \dots, T_n]; S$ when there is no ambiguity on n, and similarly for $![\tilde{T}]; S$.

Definition 2.2 (Unfolding)**.** $\mathrm{unfold}(\mu X.T) = \mathrm{unfold}(T[\mu X.T/X])$, and $\mathrm{unfold}(T) = T$ otherwise.

Note that as all terms are contractive, this is well-defined.

Definition 2.3 (Coinductive subtyping)**.** A relation $\mathscr{R}$ is a subtyping relation if the following rules hold, for all $T \mathscr{R} U$:

- If $\mathrm{unfold}(T) = ?[\tilde{T}']; S_1$ then $\mathrm{unfold}(U) = ?[\tilde{U}']; S_2$, and $\tilde{T}' \mathscr{R} \tilde{U}'$ and $S_1 \mathscr{R} S_2$.

- If unfold$(T) = ![\tilde{T}'];S_1$ then unfold$(U) = ![\tilde{U}'];S_2$, and $\tilde{U}'\mathscr{R}\tilde{T}'$ and $S_1\mathscr{R}S_2$.
- If unfold$(T) = \&\langle l_i : T_i\rangle_{1\le i\le m}$ then unfold$(U) = \&\langle l_i : U_i\rangle_{1\le i\le n}$, and $m \le n$ and $\forall i \in \{1,\dots,m\}.\, T_i\mathscr{R}U_i$.
- If unfold$(T) = \oplus\langle l_i : T_i\rangle_{1\le i\le m}$ then unfold$(U) = \oplus\langle l_i : U_i\rangle_{1\le i\le n}$, and $n \le m$ and $\forall i \in \{1,\dots,n\}.\, T_i\mathscr{R}U_i$.
- If unfold$(T) = \mathsf{end}$ then unfold$(U) = \mathsf{end}$.

Subtyping $\le_c$ is defined by $S \le_c T$ if $(S,T) \in \mathscr{R}$ in some type simulation $\mathscr{R}$. It immediately follows that subtyping is the largest type simulation.

Definition 2.4 (Coinductive equality)**.** $T =_c T'$ if $T \le_c T'$ and $T' \le_c T$.

Example 2.5 (Interface)**.** We can now formally prove the subtyping relations mentioned in Section 1. Let us define $\mathscr{R} = \{(T_2^{\mathsf{interface}}, T_3^{\mathsf{interface}}), (\,?[\mathsf{end}];T_2^{\mathsf{interface}},\, ?[\mathsf{end}];T_3^{\mathsf{interface}}), (\mathsf{end},\mathsf{end}), (T_2^{\mathsf{interface}}, T_1^{\mathsf{interface}}),$ $(\,?[T_2^{\mathsf{interface}}];T_2^{\mathsf{interface}},\, ?[T_1^{\mathsf{interface}}];, T_3^{\mathsf{interface}}), (\,?[\mathsf{end}];T_2^{\mathsf{interface}},\, ?[\mathsf{end}];T_1^{\mathsf{interface}})\}$. We verify that the rules in Definition 2.3 hold. Thus $T\mathscr{R}U$ implies $T \le_c U$. In particular, $T_2^{\mathsf{interface}} \le_c T_3^{\mathsf{interface}}$ and $T_2^{\mathsf{interface}} \le_c T_1^{\mathsf{interface}}$.

To reason about the running time of the subtyping algorithms, we define the size of a session type.

Definition 2.6 (Size)**.** $|\mathsf{end}| = |X| = 1$, $|\mu X.T| = |T| + 1$; $|\,![T_1,\dots,T_n];U| = |\,?[T_1,\dots,T_n];U| = \sum_{i=1}^n |T_i| + |U| + 1$; $|\oplus\langle l_1 : T_1,\dots,l_n : T_n\rangle| = |\&\langle l_1 : T_1,\dots,l_n : T_n\rangle| = \sum_{i=1}^n |T_i| + 1$.

We will also need a notion of subterms. The following function is defined in [3].

Definition 2.7 (Bottom-up subterms)**.** The set of *bottom-up subterms* of T is defined inductively as:
$\mathrm{Sub}(\mathsf{end}) = \{\mathsf{end}\}$; $\mathrm{Sub}(X) = \{X\}$; $\mathrm{Sub}(\mu X.T) = \{\mu X.T\} \cup \{S[\mu X.T/X] \mid S \in \mathrm{Sub}(T)\}$;
$\mathrm{Sub}(\,![T_1,\dots,T_n];S) = \{\,![T_1,\dots,T_n];S\} \cup \mathrm{Sub}(S) \cup \bigcup_{i=1}^n \mathrm{Sub}(T_i)$;
$\mathrm{Sub}(\,?[T_1,\dots,T_n];S) = \{\,?[T_1,\dots,T_n];S\} \cup \mathrm{Sub}(S) \cup \bigcup_{i=1}^n \mathrm{Sub}(T_i)$;
$\mathrm{Sub}(\oplus\langle l_1 : T_1,\dots,l_n : T_n\rangle) = \{\oplus\langle l_1 : T_1,\dots,l_n : T_n\rangle S\} \cup \bigcup_{i=1}^n \mathrm{Sub}(T_i)$; and
$\mathrm{Sub}(\&\langle l_1 : T_1,\dots,l_n : T_n\rangle) = \{\&\langle l_1 : T_1,\dots,l_n : T_n\rangle S\} \cup \bigcup_{i=1}^n \mathrm{Sub}(T_i)$.
Then we define $\mathrm{Sub}(T,U) = \mathrm{Sub}(T) \cup \mathrm{Sub}(U)$.

Due the use of unfolds (Definition 2.2) in our proof rules, it will be easier to reason with a definition of subterms that use an operation similar to unfolding in the treatment of recursive types. Thus, we give the analogue of [10, Definition 21.9.1] for session types. The full definition is in the full version of this paper [13, Definition A.1].

Definition 2.8 (Top-down subterms)**.** The set of *top-down subterms* of T is defined: $\mathrm{Sub}_{\mathrm{TD}}(\mu X.T) = \{\mu X.T\} \cup \mathrm{Sub}_{\mathrm{TD}}(T[\mu X.T/X])$, with all other rules from Definition 2.7 with Sub replaced by $\mathrm{Sub}_{\mathrm{TD}}$.

3 Inductive subtyping algorithms

Number of subterms. In this section we show that the number of top-down subterms of a binary session type is linear. Proving this directly by induction is difficult as the definition of top-down subterm of $\mu X.T$ relies on the definition of a potentially larger term $T[\mu X.T/X]$.

We adapt the proofs in [10, Chapter 21.9], which dealt with μ-types in the lambda calculus. We will use bottom-up subterms as a proxy for top-down subterms: we will show that all top-down subterms are bottom-up subterms, and there are linearly many bottom-up subterms. Note that in this section we assume all substitutions are capture-avoiding. This can be done by alpha-conversion without changing the size of the term.

Lemma 3.1. $|\mathrm{Sub}(T)| \le |T|$.

Proof. By induction on the structure of T.

Case. $T = \mathsf{end}$, or $T = X$. Trivial.

Case. $T = \mu X.T'$. Assuming the inductive hypothesis, we have $|\mathrm{Sub}(T)| = |\{T\} \cup \{S[\mu X.T/X] \mid S \in \mathrm{Sub}(T)\}| \leq |\{T\}| + |\mathrm{Sub}(T)| = 1 + |T'| = |T|$.

Case. $T = \oplus\langle T_1, \ldots, T_n\rangle$, or $T = \&\langle T_1, \ldots, T_n\rangle$. Assuming the inductive hypothesis, we have $|\mathrm{Sub}(T)| \leq |\{T\}| + \sum_{i=1}^{n} |\mathrm{Sub}(T_i)| = 1 + \sum_{i=1}^{n} |T_i| = |T|$.

Case. $T = ![T_1, \ldots, T_n];S$ or $T = ?[T_1, \ldots, T_n];S$. Similar to the above. □

Lemma 3.2. If $S \in \mathrm{Sub}(T[Q/X])$ then $S = S'[Q/X]$ for some $S' \in \mathrm{Sub}(T)$, or $S \in \mathrm{Sub}(Q)$.

Proof. By induction on the structure of T.

Case. $T = \mathsf{end}$. Then $S = \mathsf{end}$, so take $S' = \mathsf{end}$.

Case. $T = Y$. If $Y = X$, then $S \in \mathrm{Sub}(Q)$. Otherwise $S = Y$, so take $S' = Y$.

Case. $T = ![T_1, \ldots, T_n];U$. Then either:

- $S = T[Q/X]$. Then $S' = T$.
- $S \in \mathrm{Sub}(U[Q/X])$. Then, by the inductive hypothesis, either $S \in \mathrm{Sub}(Q)$, or $S = S'[Q/X]$ for some $S' \in \mathrm{Sub}(U)$. In the latter case, $S' \in \mathrm{Sub}(T)$ by definition.
- $S \in \mathrm{Sub}(T_i[Q/X])$. Similar.

Case. $T = ?[T_1, \ldots, T_n];U$, $T = \oplus\langle T_1, \ldots, T_n\rangle$ or $T = \&\langle T_1, \ldots, T_n\rangle$. Similar to the above case.

Case. $T = \mu Y.T'$. We have $S \in \mathrm{Sub}(\mu Y.T'[Q/X])$. By definition, either:

- $S = \mu Y.T'[Q/X]$. Take $S' = T = \mu Y.T'$.
- $S = S_1[(\mu Y.T'[Q/X])/Y]$ for some $S_1 \in \mathrm{Sub}(T'[Q/X])$. Then by the inductive hypothesis, either:
  - $S_1 \in \mathrm{Sub}(Q)$. Then because our substitutions are capture-avoiding, $Y \notin \mathrm{fv}(Q)$, so $S_1[(\mu Y.T'[Q/X])/Y] = S_1$. Therefore $S = S_1 \in \mathrm{Sub}(Q)$.
  - $S_1 = S_2[Q/X]$ for some $S_2 \in \mathrm{Sub}(T')$.
    Then $S = S_2[Q/X][(\mu Y.T'[Q/X])/Y] = S_2[\mu Y.T'/Y][Q/X]$. Take $S' = S_2[\mu Y.T'/Y]$. (By definition $S' \in \mathrm{Sub}(T)$.)

□

Lemma 3.3. $\mathrm{Sub}_{\mathrm{TD}}(S) \subseteq \mathrm{Sub}(S)$.

Proof. Similar to [10, Prop. 21.9.10]. We need to show that each rule of $\mathrm{Sub}_{\mathrm{TD}}$ can be matched by Sub. All rules except for $\mu X.T$ are identical. For $\mu X.T$, we use Lemma 3.2: if $S \in \mathrm{Sub}(T[\mu X.T/X])$ then either $S \in \mathrm{Sub}(T)$, or $S = S'[\mu X.T/X]$ for some $S' \in \mathrm{Sub}(T)$. The former is what we want; the latter is part of the rule for μ in Sub. □

Corollary 3.3.1. $|\mathrm{Sub}_{\mathrm{TD}}(T)| \leq |T|$. Follows from Lemmas 3.3 and 3.1.

$$\frac{T \leq U \in \Sigma}{\Sigma \vdash T \leq U}\ [\text{AS-ASSUMP}] \qquad \frac{}{\Sigma \vdash \mathsf{end} \leq \mathsf{end}}\ [\text{AS-END}]$$

$$\frac{\Sigma, \mu X.T \leq U \vdash T[\mu X.T/X] \leq U}{\Sigma \vdash \mu X.T \leq U}\ [\text{AS-RECL}] \qquad \frac{\Sigma, T \leq \mu X.U \vdash T \leq U[\mu X.U/X]}{\Sigma \vdash T \leq \mu X.U}\ [\text{AS-RECR}]$$

$$\frac{\Sigma \vdash \tilde{T} \leq \tilde{U} \qquad \Sigma \vdash V \leq W}{\Sigma \vdash ?[\tilde{T}];V \leq ?[\tilde{U}];W}\ [\text{AS-IN}] \qquad \frac{\Sigma \vdash \tilde{U} \leq \tilde{T} \qquad \Sigma \vdash V \leq W}{\Sigma \vdash ![\tilde{T}];V \leq ![\tilde{U}];W}\ [\text{AS-OUT}]$$

$$\frac{m \leq n \qquad \forall i \in 1,\ldots,m.\Sigma \vdash S_i \leq T_i}{\Sigma \vdash \&\langle l_i : S_i\rangle_{1\leq i\leq m} \leq \&\langle l_i : T_i\rangle_{1\leq i\leq n}}\ [\text{AS-BRA}] \qquad \frac{m \leq n \qquad \forall i \in 1,\ldots,m.\Sigma \vdash S_i \leq T_i}{\Sigma \vdash \oplus\langle l_i : S_i\rangle_{1\leq i\leq n} \leq \oplus\langle l_i : T_i\rangle_{1\leq i\leq m}}\ [\text{AS-SEL}]$$

Figure 1: Algorithmic rules for subtyping, taken from [3].

3.1 An inductive algorithm [3]

First, we introduce the algorithm for checking subtyping in the original paper by Gay and Hole [3]. The paper introduces the *algorithmic rules for subtyping* shown in Figure 1. These rules prove judgements of the form $\Sigma \vdash T \leq U$, which is intuitively read: "assuming the relations in Σ, we can deduce that T is a subtype of U". The paper then formalises this by proving soundness and completeness of the rules in Figure 1, i.e. $T \leq_c U$ iff there is a proof tree deriving $\emptyset \vdash T \leq U$.

Thus, in the algorithm, the objective is to infer this rule. To do this, it builds the proof tree bottom-up, using rules in Figure 1 with [AS-ASSUMP] used with highest priority, and ties between [AS-RECL] and [AS-RECR] broken arbitrarily (we will assume that [AS-RECL] takes priority). All other rules are applicable on disjoint sets of judgements, so there is no further ambiguity.

Gay and Hole's proof of termination [3, Lemma 10] contains the following fact, using Sub instead of Sub_{TD}. However, with our definition of Sub_{TD}, the proof is clearer:

Lemma 3.4. If $\Gamma \vdash T' \leq U'$ is produced from $\emptyset \vdash T \leq U$, then $T' \in \text{Sub}_{\text{TD}}(T,U)$ and $U' \in \text{Sub}_{\text{TD}}(T,U)$, and for all $V \leq W \in \Gamma$, $V \in \text{Sub}_{\text{TD}}(T,U)$ and $W \in \text{Sub}_{\text{TD}}(T,U)$.

Proof. Verify this for each rule in Figure 1. The result follows from transitivity of the subterm relation. □

Definition 3.5 (Nesting depth)**.** $\text{nd}(\mathsf{end}) = \text{nd}(X) = 1$; $\text{nd}(\mu X.T) = \text{nd}(T) + 1$; $\text{nd}(\,?[T_1,\ldots,T_n];U) = \text{nd}(\,![T_1,\ldots,T_n];U) = \max(\{\text{nd}(T_i) \mid 1 \leq i \leq n\} \cup \{\text{nd}(U)\}) + 1$; and $\text{nd}(\oplus\langle l_1 : T_1,\ldots,l_n : T_n\rangle) = \text{nd}(\&\langle l_1 : T_1,\ldots,l_n : T_n\rangle) = \max(\{\text{nd}(T_i) \mid 1 \leq i \leq n\}) + 1$.

Using the notion of nesting depth, Gay and Hole proceed to prove termination, as follows. Observe that when generating the premise $\Gamma' \vdash V' \leq W'$ above $\Gamma \vdash V \leq W$, either $|\Gamma'| > |\Gamma|$; or $|\Gamma'| = |\Gamma|$ and $\text{nd}(V') < \text{nd}(V)$.
Thus pairs $(\Gamma, \text{nd}(V))$ for judgements $\Gamma \vdash V \leq W$ are distinct along any path from the root to any leaf of the proof tree. Termination follows by observing that both $|\Gamma|$ and $\text{nd}(V)$ are bounded. We may extend this to a complexity bound as follows.

Theorem 3.6. The worst-case complexity of Gay and Hole's subtyping algorithm is $O\left(n^{n^3}\right)$, where n is the sum of the sizes of the two inputs.

Proof. When $\Gamma \vdash V \leq W$ is generated from the rule $\emptyset \vdash T \leq U$, we have:

- The number of possible judgements in Γ is $|\text{Sub}_{\text{TD}}(T,U)|^2$ by Lemma 3.4.

```
1: function SUBTYPE(Δ,Σ,T,U)
2:     if Δ = false then
3:         return false
4:     end if
5:     if Σ ⊢ T ≤ U ∈ Δ then  ▷ Memoization of inferences
6:         return Δ
7:     end if
8:     Δ ← Δ ∪ Σ ⊢ T ≤ U ∈ Δ  ▷ Add to the memoized set
9:     if T ≤ U ∈ Σ then
10:        return Δ
11:    else if T = end and U = end then
12:        return Δ
13:    else if T = μX.T′ then
14:        return SUBTYPE(Δ,Σ∪{T ≤ U},T′[T/X],U)
15:    else if U = μX.U′ then
16:        return SUBTYPE(Δ,Σ∪{T ≤ U},T,U′[U/X])
17:    else if T = ?[T̃];V and U = ?[Ũ];W then
18:        for (T_i,U_i) ← (T̃,Ũ) do
19:            Δ ← SUBTYPE(Δ,Σ,T_i,U_i)
20:        end for
21:        return SUBTYPE(Δ,Σ,V,W)
22:    else if T = ![T̃];V and U = ![Ũ];W then
23:        for (T_i,U_i) ← (T̃,Ũ) do
24:            Δ ← SUBTYPE(Δ,Σ,U_i,T_i)
25:        end for
26:        return SUBTYPE(Δ,Σ,V,W)
27:    else if T = &⟨l_i : T_i⟩_{1≤i≤m} and U = &⟨l_i : U_i⟩_{1≤i≤n} and m ≤ n then
28:        for i ← 1..m do
29:            Δ ← SUBTYPE(Δ,Σ,T_i,U_i)
30:        end for
31:        return Δ
32:    else if T = ⊕⟨l_i : T_i⟩_{1≤i≤n} and U = ⊕⟨l_i : U_i⟩_{1≤i≤m} and m ≤ n then
33:        for i ← 1..m do
34:            Δ ← SUBTYPE(Δ,Σ,T_i,U_i)
35:        end for
36:        return Δ
37:    else
38:        return false
39:    end if
40: end function
```

Figure 2: A memoized subtyping algorithm.

- The number of possible values of V is $|\mathrm{Sub}_{\mathrm{TD}}(T,U)|$, thus there are only $|\mathrm{Sub}_{\mathrm{TD}}(T,U)|$ possible values of $\mathrm{nd}(V)$.

Therefore the height of the tree is bounded by $(|T|+|U|)^3$, using Corollary 3.3.1, and the branching factor is $O(|T|+|U|)$, so the worst-case complexity is $O\left(n^{n^3}\right)$, taking $n = |T|+|U|$. □

3.2 An algorithm with memoization

A way to optimise the first algorithm is to treat the proof tree like a proof DAG: as identical nodes will have the same subtrees, we can search for their proofs only once. The algorithm in Figure 2 performs a depth-first search of the proof tree, ignoring nodes that have been seen before. This is done by keeping a set Δ of visited nodes.

Theorem 3.7. The algorithm in Fig. 2 has worst-case time complexity $2^{O(n^2)}$.

Proof. Observe that the runtime is proportional to $|\Delta|$ at the end of the program. As an upper bound, by Lemma 3.4, there are $|\mathrm{Sub}_{\mathrm{TD}}(T,U)|^2$ possible judgements in Σ, and $|\mathrm{Sub}_{\mathrm{TD}}(T,U)|$ possible terms for T and U, so Δ has size at most $2^{|\mathrm{Sub}_{\mathrm{TD}}(T,U)|^2} \cdot |\mathrm{Sub}_{\mathrm{TD}}(T,U)|^2 = 2^{O(n^2)}$, again by Corollary 3.3.1. □

Example 3.8. To show for certain that the algorithms so far are exponential in complexity, we will show that the following construction takes exponential time for the two subtyping algorithms presented:

$$T_k \leq_c T_{k+1} \quad \text{where} \quad T_k := \mu X.\, ?[\mu Y_{k-1}.V_{k-1}^k];\, ?[\mu Y_{k-2}.V_{k-2}^k];\ldots ?[\mu Y_1.V_1^k];\, ?[\mu Y_0.V_0^k];X$$
$$\text{and} \quad V_l^k := \underbrace{?[\mu Z.\, ?[Z];Z];\ldots.\, ?[\mu Z.\, ?[Z];Z];}_{l \text{ times}}X$$

We first show that the subtyping relation holds, by proving the stronger notion of coinductive equality.

Lemma 3.9. In Example 3.8, $T_k =_c \mu X.\, ![X];X$.

Proof. Let $U = \mu X.![X];X$, and take $\mathscr{R} = \{(T,U) \mid T \in \mathrm{Sub_{TD}}(T_k)\}$ and $\mathscr{R}' = \{(U,T) \mid T \in \mathrm{Sub_{TD}}(T_k)\}$. It is easy to see that for all $T \in \mathrm{Sub_{TD}}(T_k)$, $\mathrm{unfold}(T) = ![S_1];S_2$ for some $S_1, S_2 \in \mathrm{Sub_{TD}}(T_k)$. Also we have $\mathrm{unfold}(U) = ![U];U$. Hence $(S_1,U),(S_2,U) \in \mathscr{R}$ and $(U,S_1),(U,S_2) \in \mathscr{R}'$. Thus $\mathscr{R}$ and $\mathscr{R}'$ are type simulations. □

Corollary 3.9.1. $T_k \leq_c T_{k+1}$.

Proof. Follows from transitivity of $=_c$. □

Therefore, by completeness of the inductive rules [3] it follows that the algorithm will construct a valid derivation of $\emptyset \vdash T_k \leq T_{k+1}$. As both algorithms presented so far will need to traverse every node in the tree at least once, we will now show that this proof tree has an exponential amount of nodes.

Lemma 3.10. Define: $W_r^k = V_r^k[T_k/X]$ and $S_r^k = ?[\mu Y_{r-1}^k.W_{r-1}^k];\dots ?[\mu Y_0^k.W_0^k];T_k$. Then for every sequence $\alpha_1,\dots,\alpha_l (0 \leq l < k)$ such that $0 \leq \alpha_i < k-i$:

$$\left.\begin{array}{l} \{T_k \leq T_{k+1}, S_k^k \leq T_{k+1}\} \cup \{\mu Y_{\alpha_i}.W_{\alpha_i}^k \leq \mu Y_{\alpha_i+i}.W_{\alpha_i+i}^{k+1} \mid 1 \leq i \leq l\} \\ \cup \{W_{\alpha_i}^k \leq \mu Y_{\alpha_i+i}.W_{\alpha_i+i}^{k+1} \mid 1 \leq i \leq l\} \cup \{T_k \leq W_i^{k+1} \mid 1 \leq i \leq l\} \\ \cup \{S_{k-i}^k \leq T_{k+1} \mid 1 \leq i \leq l\} \end{array}\right\} \vdash S_{k-l}^k \leq S_{k+1}^{k+1} \tag{4}$$

is derivable from $\emptyset \vdash T_k \leq T_{k+1}$ as the root.

Proof. By induction on l. For some $\alpha_1,\dots,\alpha_{k-1}$, let $\mathscr{S}_l$ be the set of inferences on the left side of (4).

The base case $l = 0$ is simple: $\mathscr{S}_0 = \{T_k \leq T_{k+1}, S_k^k \leq T_{k+1}\}$. The corresponding proof tree is

$$\dfrac{\dfrac{\mathscr{S}_0 \vdash S_k^k \leq S_{k+1}^{k+1}}{T_k \leq T_{k+1} \vdash S_k^k \leq T_{k+1}}\,[\text{AS-RecR}]}{\emptyset \vdash T_k \leq T_{k+1}}\,[\text{AS-RecL}]$$

For the inductive step, we will build the tree starting from $\mathscr{S}_{l-1} \vdash S_{k-(l-1)}^k \leq S_{k+1}^k$, for $l > 0$, using the inductive hypothesis. The following proof tree works, taking
$\mathscr{C}_1 = \mu Y_{\alpha_l}.W_{\alpha_l}^k \leq \mu Y_{\alpha_l+l}^{k+1}.W_{\alpha_l+l}^{k+1}$, $\mathscr{C}_2 = W_{\alpha_l}^k \leq \mu Y_{\alpha_l+l}^{k+1}.W_{\alpha_l+l}^{k+1}$, $\mathscr{C}_3 = T_k \leq W_l^{k+1}$, $\mathscr{C}_4 = S_{k-l}^k \leq T_{k+1}$:

$$\dfrac{\dfrac{\dfrac{\cdots \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathscr{S}_{l-1} \cup \{\mathscr{C}_1,\mathscr{C}_2,\mathscr{C}_3,\mathscr{C}_4\} \vdash S_{k-l}^k \leq S_{k+1}^{k+1}}{\mathscr{S}_{l-1} \cup \{\mathscr{C}_1,\mathscr{C}_2,\mathscr{C}_3\} \vdash S_{k-l}^k \leq T_{k+1} = W_0^{k+1}}\,[\text{AS-RecR}] \quad \cdots}{\vdots}\,[\text{AS-In}]}{\mathscr{S}_{l-1} \cup \{\mathscr{C}_1,\mathscr{C}_2,\mathscr{C}_3\} \vdash S_k^k \leq W_l^{k+1}}\,[\text{AS-In}]}{\mathscr{S}_{l-1} \cup \{\mathscr{C}_1,\mathscr{C}_2\} \vdash W_0^k = T_k \leq W_l^{k+1}}\,[\text{AS-RecL}] \quad \cdots}{\vdots}\,[\text{AS-In}]}{\mathscr{S}_{l-1} \cup \{\mathscr{C}_1,\mathscr{C}_2\} \vdash W_{\alpha_l}^k \leq W_{\alpha_l+l}^{k+1}}\,[\text{AS-In}]}{\mathscr{S}_{l-1} \cup \{\mathscr{C}_1\} \vdash W_{\alpha_l}^k \leq \mu Y_{\alpha_l}^{k+1}.W_{\alpha_l+1}^{k+1}}\,[\text{AS-RecR}]}{\mathscr{S}_{l-1} \vdash \mu Y_{\alpha_l}.W_{\alpha_l}^k \leq \mu Y_{\alpha_l+l}^{k+1}.W_{\alpha_l+l}^{k+1}}\,[\text{AS-RecL}]}{\mathscr{S}_{l-1} \vdash S_{\alpha_l+1}^k \leq S_{\alpha_l+l+1}^{k+1}}\,[\text{AS-In}] \quad \cdots}{\vdots}\,[\text{AS-In}]}{\mathscr{S}_{l-1} \vdash S_{k-(l-1)}^k \leq S_{k+1}^{k+1}}\,[\text{AS-In}]$$

Observing that $\mathscr{S}_l = \mathscr{S}_{l-1} \cup \{\mathscr{C}_1,\mathscr{C}_2,\mathscr{C}_3,\mathscr{C}_4\}$ finishes the proof. □

$$\frac{\text{unfold}(T) = \mathsf{end}}{T \xrightarrow{\mathsf{end}} \mathsf{Skip}}\ [\text{G-E}]\quad \frac{\text{unfold}(T) = ?[T_1,\ldots,T_n];U}{T \xrightarrow{?\mathsf{c}} U}\ [\text{G-IC}]\quad \frac{\text{unfold}(T) = ?[T_1,\ldots,T_n];U \qquad 1 \le i \le n}{T \xrightarrow{?\mathsf{p}_i} T_i}\ [\text{G-IP}]$$

$$\frac{\text{unfold}(T) = ![T_1,\ldots,T_n];U}{T \xrightarrow{!\mathsf{c}} U}\ [\text{G-OC}]\quad \frac{\text{unfold}(T) = ![T_1,\ldots,T_n];U \qquad 1 \le i \le n}{T \xrightarrow{!\mathsf{p}_i} T_i}\ [\text{G-OP}]$$

$$\frac{\text{unfold}(T) = \&\langle l_i : T_i\rangle_{1\le i\le m} \qquad 1 \le j \le m}{T \xrightarrow{\oplus l_j} T_j}\ [\text{G-B}]\quad \frac{\text{unfold}(T) = \oplus\langle l_i : T_i\rangle_{1\le i\le m} \qquad 1 \le j \le m}{T \xrightarrow{\&l_j} T_j}\ [\text{G-S}]$$

Figure 3: Rules for the type LTS.

Theorem 3.11. The lower bound of the worst-case complexity of both inductive algorithms in this section is $\Omega((\sqrt{n})!)$.

Proof. Consider Example 3.8. The complexity is at least the number of distinct nodes in the proof tree. Lemma 3.10 shows that there are $\Omega(k!)$ such nodes, by observing that each sequence $\alpha_1,\ldots,\alpha_l$ yields a distinct set $\mathscr{S}_l$. As $|T_k| + |T_{k+1}| = \Theta(k^2)$, we conclude that both algorithms on inductive trees run in worst-case exponential time, i.e. $\Omega((\sqrt{n})!)$. □

4 Quadratic subtyping algorithm

We exploit the coinductive nature of subtyping to yield a quadratic algorithm. Firstly, we translate the constructs from Definition 2.3 into a labelled transition system (LTS), so that subtyping is defined as a simulation-like relation.

Definition 4.1. The type LTS is defined as in Figure 3. For a type T, the type LTS for T is the part of the LTS that is reachable from T.

The above definition gives us a graphical representation of types, which will be easier to work with. We show that the size of this LTS is linear:

Lemma 4.2. The number of nodes in the type LTS for T is $O(|T|)$.

Proof. Note that $T \xrightarrow{\alpha} T'$ implies $T' \in \text{Sub}_{\text{TD}}(T)$ or $T' = \mathsf{Skip}$, thus all nodes reachable from T are elements of $\text{Sub}_{\text{TD}}(T) \cup \{\mathsf{Skip}\}$. The result follows from Corollary 3.3.1. □

Lemma 4.3. The number of edges in the type LTS for T is $O(|T|)$.

Proof. Define the following function, which is the out-degree of an unfolded type: $\text{od}(X) = 0$; $\text{od}(\mathsf{end}) = 1$; $\text{od}(\,![T_1,\ldots,T_n];U) = \text{od}(\,?[T_1,\ldots,T_n];U) = n+1$; and $\text{od}(\oplus\langle l_1 : T_1,\ldots,l_n : T_n\rangle) = \text{od}(\&\langle l_1 : T_1,\ldots,l_n : T_n\rangle) = n$. Also, $\text{od}(\mu X.T) = 0$ as it is not an unfolded type.

Then, the number of edges in the LTS is $\sum_{U \text{ reachable from } T} \text{od}(\text{unfold}(U)) \le \sum_{U\in\text{Sub}_{\text{TD}}(T)} \text{od}(U) \le \sum_{U\in\text{Sub}(T)} \text{od}(U)$.

Let $f(T) = \sum_{U\in\text{Sub}(V)} \text{od}(T)$.

We prove that $f(T) \le 2|T| - 1$, by structural induction on T.

Case. $T = \mathsf{end}$, or $T = X$. Then $\text{Sub}(T) = \{T\}$, so $f(T) \le 2|T| - 1$.

Case. $T = \mu X.T'$. Then $f(T) = \text{od}(T) + f(T')$. By the inductive hypothesis, $f(T') \le 2|T'| - 1$. By definition, $\text{od}(T) = 0$, so $\sum_{U\in\text{Sub}(T)} \text{od}(U) \le 2|T'| - 1 \le 2|T| - 1$.

Case. $T = ![T_1,\ldots,T_n];W$, $T = ?[T_1,\ldots,T_n];W$. Then $f(T) = \mathrm{od}(T) + f(W) + \sum_{i=1}^{n} f(T_i)$. By the inductive hypothesis, $f(W) \leq 2|W| - 1$ and $f(T_i) \leq 2|T_i| - 1$. By definition, $\mathrm{od}(T) = n+1$, so $f(T) \leq n+1+2|W|-1+\sum_{i=1}^{n}(2|T_i|-1) = 2(1+|W|+\sum_{i=1}^{n}|T_i|) - 2 = 2|T| - 2 \leq 2|T| - 1$.

Case. $T = \oplus\langle T_1,\ldots,T_n\rangle$, $T = \&\langle T_1,\ldots,T_n\rangle$. Then $f(T) = \mathrm{od}(T) + \sum_{i=1}^{n} f(T_i)$. By the inductive hypothesis, $f(T_i) \leq 2|T_i| - 1$. By definition, $\mathrm{od}(T) = n$, so $f(T) \leq n + \sum_{i=1}^{n}(2|T_i|-1) = 2\sum_{i=1}^{n}|T_i| - 1 \leq 2|T| - 1$. □

In line with the treatment of context-free session types in Silva *et al.* [11], we can then rewrite Definition 2.3 in terms of this representation. In the language of the paper, this is a $\mathscr{X}\mathscr{Y}\mathscr{Z}\mathscr{W}$-simulation, with $\mathscr{X} = \{?\mathsf{c}, !\mathsf{c}, ?\mathsf{p}_i, \&l, \mathsf{end}\}, \mathscr{Y} = \{?\mathsf{p}_i, \oplus l, \mathsf{end}\}, \mathscr{Z} = \alpha \in \{!\mathsf{p}_i\}, \mathscr{W} = \alpha \in \{!\mathsf{p}_i\}$. We will then demonstrate how to check this $\mathscr{X}\mathscr{Y}\mathscr{Z}\mathscr{W}$-simulation relation in quadratic time on a type LTS.

Definition 4.4. $\mathscr{R}$ is a subtyping relation if, for all $T\mathscr{R}U$:

- If $T \xrightarrow{\alpha} T'$ then $U \xrightarrow{\alpha} U'$, and $T'\mathscr{R}U'$, for $\alpha \in \{?\mathsf{c}, !\mathsf{c}, ?\mathsf{p}_i, \&l, \mathsf{end}\}$.
- If $U \xrightarrow{\alpha} U'$ then $T \xrightarrow{\alpha} T'$, and $T'\mathscr{R}U'$, for $\alpha \in \{?\mathsf{p}_i, \oplus l, \mathsf{end}\}$.
- If $T \xrightarrow{\alpha} T'$ then $U \xrightarrow{\alpha} U'$, and $U'\mathscr{R}T'$, for $\alpha \in \{!\mathsf{p}_i\}$.
- If $U \xrightarrow{\alpha} U'$ then $T \xrightarrow{\alpha} T'$, and $U'\mathscr{R}T'$, for $\alpha \in \{!\mathsf{p}_i\}$.

$S \leq_c T$ if $(S,T) \in \mathscr{R}$ in some type simulation $\mathscr{R}$.

It follows that Definitions 2.3 and 4.4 are equivalent.

Definition 4.5. Call a pair (T,U) *inconsistent* if at least one of the following hold: (1) $T \xrightarrow{\alpha} T'$ and $U \not\xrightarrow{\alpha}$, for some $\alpha \in \{?\mathsf{c}, !\mathsf{c}, ?\mathsf{p}_i, \&l, \mathsf{end}\}$; (2) $U \xrightarrow{\alpha} U'$ and $T \not\xrightarrow{\alpha}$, for some $\alpha \in \{?\mathsf{p}_i, \oplus l, \mathsf{end}\}$; (3) $T \xrightarrow{\alpha} T'$ and $U \not\xrightarrow{\alpha}$, for some $\alpha \in \{!\mathsf{p}_i\}$; or (4) $U \xrightarrow{\alpha} U'$ and $T \not\xrightarrow{\alpha}$, for some $\alpha \in \{!\mathsf{p}_i\}$.

The LTS presentation gives rise to the following algorithm for checking $T \leq_c U$, where we want to find a consistent relation $\mathscr{R}$ containing (T,U); this can be extended to an algorithm that checks for any $\mathscr{X}\mathscr{Y}\mathscr{Z}\mathscr{W}$-simulation on finite structures.

Theorem 4.6. $T \leq_c U$ can be checked in $O(n^2)$ time (where $n = |T| + |U|$).

Proof. Our algorithm is as follows: construct a graph on nodes $(T',U') \in (\mathrm{Sub}_{\mathrm{TD}}(T,U) \cup \{\mathsf{Skip}\}) \times (\mathrm{Sub}_{\mathrm{TD}}(T,U) \cup \{\mathsf{Skip}\})$. For each node, check whether it is inconsistent (Definition 4.5). Then, add an edge $(V,W) \to (V',W')$ if $V\mathscr{R}W$ directly implies $V'\mathscr{R}W'$ under Definiton 4.4. If any inconsistent nodes are reachable from (T,U), then $T \not\leq_c U$, otherwise $T \leq_c U$.

To show correctness, observe that any set of consistent vertices closed under reachability is a type simulation, directly from Definition 4.4. The minimal such set containing (T,U), if it exists, must be the set of reachable nodes from (T,U). Thus we can check if any inconsistent nodes are contained in this set to solve the problem.

Note that there are $O((|T|+|U|)^2)$ nodes and edges in the graph, by Lemmas 3.3.1 and 4.3. Thus we can check $T \leq_c U$ in $O(n^2)$ time with a simple reachability search. □

Corollary 4.6.1. $T' \leq_c U'$ for all subterms $T', U' \in \mathrm{Sub}_{\mathrm{TD}}(T,U)$ can be checked in $O(n^2)$ time.

Proof. By finding the set of nodes in the above graph for which no inconsistent node is reachable, which can be done with a graph search from the inconsistent nodes in time linear in the size of the graph. □

The next examples use the above algorithm to decide subtyping for examples from Section 1.

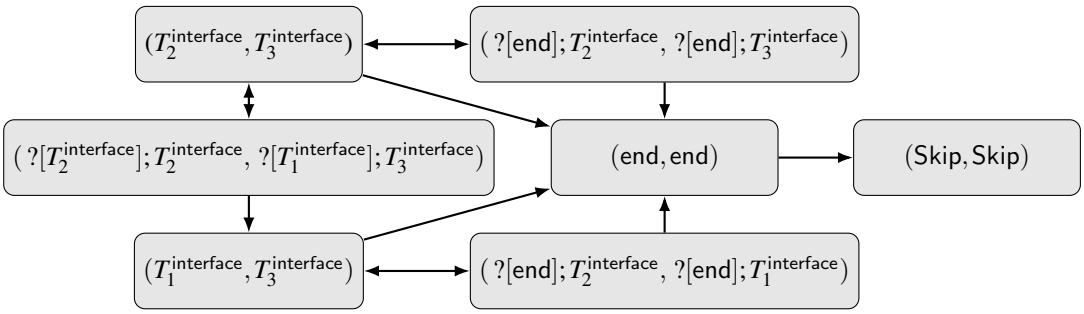


Figure 4: Graph for $(T_2^{\mathsf{interface}}, T_3^{\mathsf{interface}})$.

Example 4.7. The graph for $(T_2^{\mathsf{interface}}, T_3^{\mathsf{interface}})$ is drawn in Figure 4. There are no inconsistent nodes, which shows that $T_2^{\mathsf{interface}} \leq_c T_3^{\mathsf{interface}}$. Note the similarity to Example 2.5.

Example 4.8. Node $(T_1^{\mathsf{interface}}, T_2^{\mathsf{interface}})$ is inconsistent because $T_2^{\mathsf{interface}} \xrightarrow{\oplus\mathsf{replicate}} ?[T_2^{\mathsf{interface}}]; T_2^{\mathsf{interface}}$ but $T_1^{\mathsf{interface}} \not\xrightarrow{\oplus\mathsf{replicate}}$. Thus $T_1^{\mathsf{interface}} \not\leq_c T_2^{\mathsf{interface}}$.

We could also represent this algorithm procedually, similarly to Figure 2, differing only in what we keep track of to avoid repetition. In the inductive algorithm we store entire judgements of the form $\Gamma \vdash T \leq U$ (lines 1-8), but in this quadratic algorithm we only store visited nodes of the form $T \leq U$. The main difference is the number of possible values we could store: exponential in the former case and quadratic in the latter. The full version of this paper [13, Figure A.1] contains this presentation of the algorithm. This procedural form is also similar to the subtyping algorithm for μ-lambda-terms in [10, Fig. 21-4].

However, the LTS presentation of the algorithm does have its benefits; by representing the types as a LTS (as in Theorem 4.6), we obtain a representation of types that eliminates the overhead of manipulating the types, to yield a truly quadratic algorithm.

5 Related work

The first algorithm for subtyping of recursive function types dates back to Amadio and Cardelli [1], where they give an exponential algorithm for recursive function type subtyping. Their algorithm inductively checks for $\alpha \to \beta \leq \gamma \to \delta$ by recursively checking that $\alpha \leq \gamma$ and $\beta \leq \delta$, unwrapping μ-recursions, and keeping track of which pairs have been checked before.

Pierce [10, Chapter 21.12] surveys the development of quadratic subtyping algorithms for these types. Notably, Kozen *et al.* [7] represent types as automata, then does a linear-time check on the product automaton of two types to check for subtyping, a method similar to our third algorithm (§4).

The subtyping relation for synchronous session types was introduced by Gay and Hole [3], in which they showed that subtyping is sound and decidable. Their algorithm for subtyping (as presented here in §3.1), is similar to Amadio and Cardelli's, adapted for session types. Later, Chen *et al.*[2] proved that this relation is *precise*: no strictly larger relation respects type safety.

Lange and Yoshida [8] investigate subtyping for session types by converting terms to a modal μ-calculus formula which represents its subtypes, then using a model checker to check for subtyping. A short analysis of two other subtyping algorithms is also provided, along with an empirical evaluation. The first is Gay and Hole's algorithm [3], in which a doubly-exponential upper bound is given, which we improve to a singly-exponential bound. The second is an adaptation of Kozen's algorithm [7], which

is similar to our LTS-based construction (§4) in that it builds a product automaton and checks for reachability. A quadratic algorithm is given; however, the type system provided in [8] does not allow sending sessions as messages; instead, sorts are used as payloads. Our algorithm is adapted to remove this restriction. A comparison of complexity bounds in [8] and this paper is in Table 1.

Subtyping for non-regular session types, in which there are infinitely many subterms, are explored by Silva *et al.* [11], in which the concept of $\mathscr{X}\mathscr{Y}\mathscr{Z}\mathscr{W}$-simulation for session type subtyping is introduced; it is also used in this paper. A sound algorithm is introduced to semi-decide the subtyping problem for context-free session types, which is accompanied by an empirical evaluation. In general, subtyping for context-free session types is undecidable, as shown by Padovani [9].

Algorithm	[8]	This paper
Gay and Hole [3]	$O(n^{2^n})$	$O\left(n^{n^3}\right), \Omega((\sqrt{n})!)$
[3] with memoization	–	$2^{O(n^2)}, \Omega((\sqrt{n})!)$
Coinductive subtyping	$O(n^2)$	$O(n^2)$

Table 1: Comparison between upper and lower bounds for the worst-case complexity given in [8] and this paper to check the subtyping relation $T \leq_c U$ where $n = |T| + |U|$.

We believe that the results from this paper could be applied to other session typing schemes with little difficulty: for example, the subtyping relation for local synchronous multiparty session types [6], which is also sound and complete [4].

Acknowledgements. The authors thank PLACES'24 reviewers for their careful reading and comments. The first author is supported by the Keble Association Grant. The second author is supported by EPSRC EP/T006544/2, EP/Y005244/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/T014709/2, EP/V000462/1, EP/X015955/1 and Horizon EU TaRDIS 101093006.

References

[1] Roberto Amadio & Luca Cardelli (1993): *Subtyping Recursive Types*. *ACM transactions on programming languages and systems* 15(4), pp. 575–631, doi:10.1145/155183.155231.

[2] Tzu Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas & Nobuko Yoshida (2017): *On the Preciseness of Subtyping in Session Types*. *Logical Methods in Computer Science* 13(2), doi:10.23638/LMCS-13(2:12)2017.

[3] Simon Gay & Malcolm Hole (2005): *Subtyping for Session Types in the Pi Calculus*. *Acta Informatica* 42(2-3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[4] Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas & Nobuko Yoshida (2019): *Precise Subtyping for Synchronous Multiparty Sessions*. *Journal of Logical and Algebraic Methods in Programming* 104, pp. 127–173, doi:10.1016/j.jlamp.2018.12.002.

[5] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFB0053567.

[6] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, Association for Computing Machinery, New York, NY, USA, pp. 273–284, doi:10.1145/1328438.1328472.

[7] Dexter Kozen, Jens Palsberg & Michael I. Schwartzbach (1993): *Efficient Recursive Subtyping*. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, Association for Computing Machinery, New York, NY, USA, pp. 419–428, doi:10.1145/158511.158700.

[8] Julien Lange & Nobuko Yoshida (2016): *Characteristic Formulae for Session Types*. In Marsha Chechik & Jean-François Raskin, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 833–850, doi:10.1007/978-3-662-49674-9_52.

[9] Luca Padovani (2019): *Context-Free Session Type Inference*. *ACM Transactions on Programming Languages and Systems* 41(2), pp. 9:1–9:37, doi:10.1145/3229062.

[10] Benjamin C. Pierce (2002): *Types and Programming Languages*, 1st edition. The MIT Press.

[11] Gil Silva, Andreia Mordido & Vasco T. Vasconcelos (2023): *Subtyping Context-Free Session Types*. 279, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 11:1–11:19, doi:10.4230/LIPICS.CONCUR.2023.11.

[12] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and Its Typing System*. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou & Sergios Theodoridis, editors: *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, *Lecture Notes in Computer Science* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[13] Thien Udomsrirungruang & Nobuko Yoshida (2024): *Three Subtyping Algorithms for Binary Session Types and Their Complexity Analyses (full version)*, doi:10.48550/arXiv.2402.06988. arXiv:2402.06988.