EPTCS 314

Proceedings of the

12th International Workshop on

Programming Language Approaches to Concurrency- and Communication-cEntric Software

Dublin, Ireland, 26th April 2020

Edited by: Stephanie Balzer and Luca Padovani

Published: 3rd April 2020
DOI: 10.4204/EPTCS.314
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii
Stephanie Balzer and Luca Padovani

Session Type Systems based on Linear Logic: Classical versus Intuitionistic 1
Bas van den Heuvel and Jorge A. Pérez

Generating Interactive WebSocket Applications in TypeScript 12
Anson Miu, Francisco Ferreira, Nobuko Yoshida and Fangyi Zhou

Duality of Session Types: The Final Cut 23
Simon J. Gay, Peter Thiemann and Vasco T. Vasconcelos

Bounded verification of message-passing concurrency in Go using Promela and Spin 34
Nicolas Dilley and Julien Lange

Mixed Sessions: the Other Side of the Tape 46
Filipe Casal, Andreia Mordido and Vasco T. Vasconcelos

Fluent Session Programming in C# 61
Shunsuke Kimura and Keigo Imai

Preface

Stephanie Balzer
Carnegie Mellon University

Luca Padovani
Università di Torino

The PLACES workshop series is dedicated to the exploration and the understanding of a wide variety of foundational and practical ideas in the increasingly concurrent and parallel landscape of hardware and software infrastructures. Programming such systems, where concurrency and distribution are the norm rather than a marginal concern, poses significant challenges and demands radically different approaches to software development, maintentenance and deployment.

This volume contains the proceedings of PLACES 2020, the 12^{th} edition of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software. The workshop was scheduled to be held in Dublin, Ireland, on April 26^{th} 2020, as a satellite event of ETAPS, the European Joint Conferences on Theory and Practice of Software. However, the COVID-19 spread forced the ETAPS organizers to postpone the whole event to some later date which was unknown at the time these proceedings have been prepared for publication.

Below is the list of Programme Committee members of PLACES 2020:

- Jonathan Aldrich, Carnegie Mellon University, USA
- Massimo Bartoletti, Università di Cagliari, IT
- Ilaria Castellani, INRIA Sophia Antipolis Méditerranée, FR
- Silvia Crafa, Università di Padova, IT
- Cinzia Di Giusto, Université Nice Sophia Antipolis, FR
- Hannah Gommerstadt, Vassar College, USA
- Bart Jacobs, KU Leuven, NL
- Wen Kokke, University of Edinburgh, UK
- Hernán Melgratti, Universidad de Buenos Aires, AR
- Andreia Mordido, Universidade de Lisboa, PT
- Matthew Parkinson, Microsoft Research, UK
- Jorge A. Perez, University of Groningen, NL

The Programme Committee, after a careful and thorough reviewing process, selected 6 papers out of 9 submissions to appear in the current proceedings. Each submission was evaluated by three referees, and the accepted papers were selected after an electronic discussion. Three papers were conditionally accepted and re-checked by one PC chair or Programme Committee member before final acceptance. The submissions that were not selected for publication were judged to present interesting and valuable ideas and the authors kindly agreed to present their research at the workshop.

PLACES 2020 was made possible by the contribution and dedication of many people. We thank all the authors who submitted papers for consideration. We wish to thank Einar Broch Johnsen (University of Oslo, NO) and Zhong Shao (Yale University, USA) for accepting to give invited talks at the workshop. We are extremely grateful to the members of the Programme Committee and additional experts for their

S. Balzer, L. Padovani (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. ii–iii, doi:10.4204/EPTCS.314.0

careful reviews, and the balanced discussions during the selection process. The EasyChair system was instrumental in supporting the submission and reviewing process; the EPTCS website was similarly useful in production of these proceedings. Finally, we reserve a special thank you to Simon Gay, Vasco T. Vasconcelos and Nobuko Yoshida, who helped and guided us in the organization of this edition of the workshop.

March 23rd, 2020
Stephanie Balzer and Luca Padovani
PLACES 2020 Programme Chairs

Session Type Systems based on Linear Logic: Classical versus Intuitionistic*

Bas van den Heuvel Jorge A. Pérez
University of Groningen, The Netherlands

Session type systems have been given logical foundations via Curry-Howard correspondences based on both intuitionistic and classical linear logic. The type systems derived from the two logics enforce communication correctness on the same class of π-calculus processes, but they are significantly different. Caires, Pfenning, and Toninho informally observed that, unlike the classical type system, the intuitionistic type system enforces locality for shared channels, i.e. received channels cannot be used for replicated input. In this paper, we revisit this observation from a formal standpoint. We develop United Linear Logic (ULL), a logic encompassing both classical and intuitionistic linear logic. Then, following the Curry-Howard correspondences for session types, we define πULL, a session type system for the π-calculus based on ULL. Using πULL we can formally assess the difference between the intuitionistic and classical type systems, and justify the role of locality and symmetry therein.

1 Introduction

Session types are a popular approach to typing message-passing concurrency [10, 11, 17]. They describe communication over channels as sequences of communication actions. This way, e.g., the session type !int.?bool.end types a channel as follows: send an integer, receive a boolean, and close the channel. Due to its simplicity and expressiveness, the π-calculus [15, 16]—the paradigmatic model of concurrency and interaction—is a widely used setting for studying session types.

In a line of work developed by Caires, Pfenning, Wadler, and several others, the theory of session types has been given strong logical foundations. Caires and Pfenning discovered a Curry-Howard correspondence between a form of session types for the π-calculus and Girard's *linear logic* [8]: session types correspond to linear logic propositions, type inference rules to sequent calculus, and communication to cut reduction [2]. The resulting session type system ensures important correctness properties for communicating processes: protocol fidelity, communication safety, deadlock freedom, and strong normalization.

As in standard logic, there are two "schools" of linear logic: *classical* [8] and *intuitionistic* [1]. The differences between classical and intuitionistic linear logic are known—see, e.g., [5, 12]. This dichotomy also appears in the logical foundations of session types: while Caires and Pfenning's correspondence relies on intuitionistic linear logic [1], Wadler developed a correspondence based on classical linear logic [18]. Superficial differences between the resulting type systems include the number of typing rules (the intuitionistic system has roughly twice as many rules as the classical system) and the shape/meaning of typing judgments (in the intuitionistic system, judgments have a rely-guarantee reading not present in the classical system). In turn, these differences follow from the way in which each system internalizes duality: the classical system provides a more explicit account of duality than the intuitionistic system.

*Work partially supported by the Netherlands Organization for Scientific Research (NWO) under the VIDI Project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

S. Balzer, L. Padovani (Eds.): Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 1–11, doi:10.4204/EPTCS.314.1

We are interested in going beyond these superficial differences, so as to establish a formal comparison between the two type systems. This seems to us an indispensable step in consolidating the logical foundations of message-passing concurrency. To our knowledge, the only available comparison is informal: Caires, Pfenning, and Toninho [4] observed that a more fundamental difference concerns the *locality principle* for shared channels. The principle states that received channels cannot be used for further reception, i.e., only the output capability of channels can be sent [13]. In session-based concurrency, shared channels define services; clients connect to services by sending a linear channel. Locality of shared channels therefore means that received channels cannot be used to provide a service. Well-known from a foundational perspective, locality has been promoted as a sensible principle for distributed implementations of (object-oriented) languages based on the π-calculus [14]. The observation in [4] is that Caires and Pfenning's intuitionistic interpretation of session types enforces locality of shared channels [4], whereas Wadler's classical interpretation does not: processes that break locality are well-typed in [18].

The existence of a class of processes that is typable in one system but not in the other immediately frames the desired formal comparison as an expressiveness question: the type system in [18] can be considered to be *more expressive* than the one in [2]. To formally examine this question, the first step is defining a basic framework of reference in which both type systems can be objectively compared. To this end, we build upon Girard's Logic of Unity [9], which subsumes classical, intuitionistic and linear logic in one system. In the same spirit, we develop *United Linear Logic* (ULL): a logic that subsumes classical and intuitionistic linear logic. Following the Curry-Howard correspondence by Caires and Pfenning, we interpret ULL as a session type system for the π-calculus, dubbed πULL. The class of πULL-typable processes therefore contains processes induced by type systems derived from both intuitionistic and classical interpretations of linear logic. Using πULL, we corroborate and formalize Caires, Pfenning, and Toninho's observation as inclusions between classes of typable processes: our technical results are that (i) πULL precisely captures the class of processes typable under the classical interpretation and that (ii) the class of processes typable under the intuitionistic interpretation is strictly included in πULL.

This paper is structured as follows. In Section 2 we introduce ULL, explain the Curry-Howard interpretation as the session type system πULL, and detail the correctness properties for processes derived by typing. Section 3 formally establishes the differences between the classical and intuitionistic interpretations of linear logic as session type systems. Section 4 concludes the paper.

2 United Linear Logic as a Session Type System

In this section, we introduce United Linear Logic (ULL), a logic based on the linear fragment of Girard's Logic of Unity [9]. We present ULL as a session type system for the π-calculus [15, 16], dubbed πULL, following the Curry-Howard correspondences established by Caires and Pfenning [2] and by Wadler [18].

Propositions / Types. Propositions in ULL correspond to session types in πULL; they are defined as follows:

Definition 2.1. ULL *propositions /* πULL *types are generated by the following grammar:*

$$A,B ::= \mathbf{1} \mid \bot \mid A \otimes B \mid A \parr B \mid A \multimap B \mid !A \mid ?A$$

Session types represent sequences of communication actions that should be performed along channels. Table 1 gives the intuitive reading of the interpretation of propositions as session types. Note that there are two types for reception: $\parr$ from classical and $\multimap$ from intuitionistic linear logic.

$\mathbf{1}$ and $\bot$	Close the channel
$A \otimes B$	Send a channel of type A and continue as B
$A \parr B$ and $A \multimap B$	Receive a channel of type A and continue as B
$!A$	Repeatedly provide a service of type A
$?A$	Connect to a service of type A

Table 1: Interpretation of ULL propositions as session types

ULL does not include the additives $A \oplus B$ and $A \,\&\, B$ of linear logic. Although the Logic of Unity does include these connectives, we leave them out from ULL (and πULL), because their interpretation as session types—internal and external choice, respectively—largely coincides in many presentations of logic-based session types. Therefore they are not particularly insightful in our formal comparison. They can be easily accommodated in ULL, with the possibility of choosing between binary choice (as in e.g. [2,4]) and n-ary choice (as in e.g. [3,18]).

Duality. The duality of ULL propositions is given in Definition 2.2. In πULL duality is reflected by the intended reciprocity of protocols between two parties: when a process on one side of a channel sends, the process on the opposite side must receive, and vice versa.

Definition 2.2. *Duality ($A^\bot$) is given by the following set of equations:*

$$\begin{array}{lll} \mathbf{1}^\bot := \bot & (A \otimes B)^\bot := A^\bot \parr B^\bot & (!A)^\bot := ?A^\bot \\ \bot^\bot := \mathbf{1} & (A \parr B)^\bot := A^\bot \otimes B^\bot & (?A)^\bot := !A^\bot \end{array}$$

It is easy to see that duality is an involution: $(A^\bot)^\bot = A$. As usual, we decree that $A \multimap B = A^\bot \parr B$. From this, we can derive the relation between $\multimap$ and $\otimes$ by means of their duals:

$$\begin{array}{l} (A \multimap B)^\bot = (A^\bot \parr B)^\bot = (A^\bot)^\bot \otimes B^\bot = A \otimes B^\bot \\ (A \otimes B)^\bot = A^\bot \parr B^\bot = A \multimap B^\bot \end{array}$$

Processes. πULL is a type system for the π-calculus processes defined as follows:

Definition 2.3. *Process terms are generated by the following grammar:*

$$P, Q := \mathbf{0} \mid [x \leftrightarrow y] \mid (\nu x)P \mid P \mid Q \mid x\langle y\rangle.P \mid x(y).P \mid !x(y).P \mid x\langle\rangle.\mathbf{0} \mid x().P$$

Process constructs for inaction $\mathbf{0}$, channel restriction $(\nu x)P$, and parallel composition $P \mid Q$ have standard readings. The same applies to constructs for output, input, and replicated input prefixes, which are denoted $x\langle y\rangle.P$, $x(y).P$, and $!x(y).P$, respectively. Process $[x \leftrightarrow y]$ denotes a forwarder that "fuses" channels x and y. We consider also constructs $x\langle\rangle.\mathbf{0}$ and $x().P$, which specify the explicit closing of channels: their synchronization represents the explicit de-allocation of linear resources. These constructs result from the non-silent interpretation of $\mathbf{1}$, which, as explained in [3], leads to a Curry-Howard correspondence that is stronger than correspondences with silent interpretations of $\mathbf{1}$ (such as those in [2,4]).

Structural congruence. Processes can be syntactically different, but still exhibit the same behavior. Such processes are *structurally congruent*, in the sense of the following definition:

Definition 2.4. *Structural congruence ($\equiv$) is given by the following set of equations, where $\equiv_\alpha$ denotes equality up to capture-avoiding α-conversion, and $fn(P)$ gives the set of free names in P, i.e. the complement of $bn(P)$: the names in P bound by restriction (νx) and (replicated) input $x(y)$ and $!x(y)$:*

$$P \mid \mathbf{0} \equiv P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$
$$P \mid Q \equiv Q \mid P \qquad [x \leftrightarrow y] \equiv [y \leftrightarrow x]$$
$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad P \equiv_\alpha Q \implies P \equiv Q$$
$$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad x \notin fn(P) \implies P \mid (\nu x)Q \equiv (\nu x)(P \mid Q)$$

Computation. In a Curry-Howard correspondence, computation is related to cut reduction in the logic. Cut reduction removes cuts from an inference tree, which reduces the size of the tree without changing the result of the inference. In the correspondence between linear logic and the π-calculus, cut reduction is related to communication, defined by the following reduction relation:

Definition 2.5. *Reduction of process terms ($\to$) is given by the following relation:*

$$x\langle y\rangle.P \mid x(z).Q \to P \mid Q\{y/z\} \qquad Q \to Q' \implies P \mid Q \to P \mid Q'$$
$$x\langle y\rangle.P \mid !x(z).Q \to P \mid Q\{y/z\} \mid !x(z).Q \qquad P \to Q \implies (\nu y)P \to (\nu y)Q$$
$$x\langle\rangle.\mathbf{0} \mid x().Q \to Q \qquad P \equiv P' \wedge P' \to Q' \wedge Q' \equiv Q \implies P \to Q$$
$$P \mid [x \leftrightarrow y] \to P\{y/x\}$$

Typing inference. The inference system of ULL is a sequent calculus with sequents of the form $\Gamma;\Delta \vdash \Lambda$ in which Γ is a collection of propositions that can be indefinitely used, and Δ and Λ collect propositions that must be used linearly (exactly once). With respect to the Logic of Unity, we added a left rule for $\mathbf{1}$ and a right rule for $\bot$, and removed rules that allow propositions to switch sides in a sequent.

πULL's typing inference system annotates sequents with process terms and channel names to form typing judgments of the following form:

$$\Gamma;\Delta \vdash P :: \Lambda$$

In this interpretation, Γ (resp. Δ and Λ), the unrestricted (resp. linear) context of P, consists of assignments of the form $x : A$, where x is a channel and A is a proposition/type. In a correct inference, these contexts together contain exactly the free channel names of the process term P. We write $\cdot$ to denote an empty context. πULL's inference rules are given in Figure 1. Note that some rules are labeled with an '$*$', which we use to distinguish a class of rules to be used in the formal comparison in the next section.

Cut reduction and identity expansion. Caires, Pfenning, and Toninho [3] showed that the validity of session type interpretations of linear logic propositions can be demonstrated by checking that cut reductions in typing inferences do correspond to reductions of processes, as well as by showing that the identity axiom of any type can be expanded to a larger process term with forwarding of a smaller type. Following this approach, πULL can be shown valid for all reductions, using CutR as well as CutL, and expansions, using IdR as well as IdL.

Correctness properties. As is usual for Curry-Howard correspondences for concurrency, the cut elimination property of linear logic means that πULL has the *soundness/subject reduction* (Theorem 2.1) and *progress* (Theorem 2.2) properties.

$$\frac{}{\Gamma; x : A \vdash [x \leftrightarrow y] :: y : A}\ (^*\text{IDR}) \qquad \frac{}{\Gamma; x : A, y : A^\perp \vdash [x \leftrightarrow y] :: \cdot}\ (\text{IDL})$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, x : A \qquad \Gamma; \Delta', x : A \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: \Lambda, \Lambda'}\ (^*\text{CUTR})$$

$$\frac{\Gamma; \Delta, x : A \vdash P :: \Lambda \qquad \Gamma; \Delta', x : A^\perp \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta' \vdash (\nu x)(P \mid Q) :: \Lambda, \Lambda'}\ (\text{CUTL})$$

$$\frac{\Gamma; x : A^\perp \vdash P :: \cdot \qquad \Gamma, u : A; \Delta \vdash Q :: \Lambda}{\Gamma; \Delta \vdash (\nu u)(!u(x).P \mid Q) :: \Lambda}\ (\text{CUT}^?)$$

$$\frac{\Gamma; \cdot \vdash P :: x : A \qquad \Gamma, u : A; \Delta \vdash Q :: \Lambda}{\Gamma; \Delta \vdash (\nu u)(!u(x).P \mid Q) :: \Lambda}\ (^*\text{CUT}^!)$$

$$\frac{\Gamma, u : A; \Delta \vdash P :: \Lambda, x : A^\perp}{\Gamma, u : A; \Delta \vdash (\nu x)u\langle x\rangle.P :: \Lambda}\ (\text{COPYR}) \qquad \frac{\Gamma, u : A; \Delta, x : A \vdash P :: \Lambda}{\Gamma, u : A; \Delta \vdash (\nu x)u\langle x\rangle.P :: \Lambda}\ (^*\text{COPYL})$$

$$\frac{}{\Gamma; \cdot \vdash x\langle\rangle.\mathbf{0} :: x : \mathbf{1}}\ (^*\mathbf{1}\text{R}) \qquad \frac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : \mathbf{1} \vdash x().P :: \Lambda}\ (^*\mathbf{1}\text{L})$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda}{\Gamma; \Delta \vdash x().P :: \Lambda, x : \perp}\ (\perp\text{R}) \qquad \frac{}{\Gamma; x : \perp \vdash x\langle\rangle.\mathbf{0} :: \cdot}\ (\perp\text{L})$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, y : A \qquad \Gamma; \Delta' \vdash Q :: \Lambda', x : B}{\Gamma; \Delta, \Delta' \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: \Lambda, \Lambda', x : A \otimes B}\ (^*\otimes\text{R})$$

$$\frac{\Gamma; \Delta, y : A, x : B \vdash P :: \Lambda}{\Gamma; \Delta, x : A \otimes B \vdash x(y).P :: \Lambda}\ (^*\otimes\text{L})$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, x : B, y : A}{\Gamma; \Delta \vdash x(y).P :: \Lambda, x : A \parr B}\ (\parr\text{R})$$

$$\frac{\Gamma; \Delta, y : A \vdash P :: \Lambda \qquad \Gamma; \Delta', x : B \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta', x : A \parr B \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: \Lambda, \Lambda'}\ (\parr\text{L})$$

$$\frac{\Gamma; \Delta, y : A \vdash P :: \Lambda, x : B}{\Gamma; \Delta \vdash x(y).P :: \Lambda, x : A \multimap B}\ (^*\multimap\text{R})$$

$$\frac{\Gamma; \Delta \vdash P :: \Lambda, y : A \qquad \Gamma; \Delta', x : B \vdash Q :: \Lambda'}{\Gamma; \Delta, \Delta', x : A \multimap B \vdash (\nu y)x\langle y\rangle.(P \mid Q) :: \Lambda, \Lambda'}\ (^*\multimap\text{L})$$

$$\frac{\Gamma; \cdot \vdash P :: y : A}{\Gamma; \cdot \vdash !x(y).P :: x : !A}\ (^*!\text{R}) \qquad \frac{\Gamma, u : A; \Delta \vdash P :: \Lambda}{\Gamma; \Delta, x : !A \vdash P\{x/u\} :: \Lambda}\ (^*!\text{L})$$

$$\frac{\Gamma, u : A; \Delta \vdash P :: \Lambda}{\Gamma; \Delta \vdash P\{x/u\} :: \Lambda, x : ?A^\perp}\ (?\text{R}) \qquad \frac{\Gamma; y : A \vdash P :: \cdot}{\Gamma; x : ?A \vdash !x(y).P :: \cdot}\ (?\text{L})$$

Figure 1: The ULL inference rules / type system

	Explicit closing	Separate unrestricted context	Identity as forwarding
πCLL [4]	No	Yes	No
CP [18]	Yes	No	Yes
$\pi\mathsf{CLL}^{\mathsf{CP}}$	Yes	Yes	Yes

Table 2: Feature comparison of three session type interpretations of classical linear logic

Theorem 2.1. *If* $\Gamma;\Delta \vdash P :: \Lambda$ *and* $P \to Q$*, then* $\Gamma;\Delta \vdash Q :: \Lambda$.

Proof (sketch). By induction on the structure of the proof of P, using cut reduction. □

Definition 2.6. *For any process P, live*(P) *if and only if there are processes Q and R and channels* $\tilde{n}$ *such that* $P \equiv (\nu\tilde{n})(\pi.Q \mid R)$*, where* $\pi \in \{x\langle y\rangle, x(y), x\langle\rangle, x()\}$.

Theorem 2.2. *If* $\cdot;\cdot \vdash P :: \cdot$ *and live*(P)*, then there exists Q such that* $P \to Q$.

Proof (sketch). The liveness assumption tells us that P is a parallel composition of a process Q that is guarded by some non-persistent communication π and some other process R. The fact that P's proof has an empty context allows us to infer that $\pi.Q$ and R must have been composed using a CUT rule and that R must be guarded by a prefix that is dual to π. Therefore, a reduction can take place. □

3 Comparing Expressivity through United Linear Logic

πULL is a suitable framework for a rigorous comparison of the expressivity of session type systems based on linear logic. In this section, we compare the class of processes typable in πULL to the classes of processes typable in a classical and intuitionistic interpretation of linear logic. For this comparison to be fair, the differences between these classes need to come only from typing, so the process languages need to be the same. This means that our classical and intuitionistic interpretations need to type process terms as given in the previous section, with the same features as those in πULL: explicit closing, a separate unrestricted context, and identity as forwarding.

Our intuitionistic and classical session type interpretations of linear logic are denoted πILL and $\pi\mathsf{CLL}^{\mathsf{CP}}$, respectively. Their respective rules are given in Figure 2 and Figure 3. πILL is based on the work by Caires, Pfenning and Toninho [3].

It is worth noticing that, because of the features we require for our type systems, we could not directly base our classical interpretation on πCLL by Caires, Pfenning and Toninho [4] nor on Wadler's CP [18]. Therefore, we have designed $\pi\mathsf{CLL}^{\mathsf{CP}}$ as a combination of features from πCLL and CP. Table 2 compares these features in πCLL, CP and $\pi\mathsf{CLL}^{\mathsf{CP}}$; the differences are merely superficial:

- Explicit closing of sessions concerns a non-silent interpretation of $\mathbf{1}$ and $\bot$ in the logic that entails a reduction on processes (which, in turn, corresponds to cut reduction). In contrast, implicit closing is due to a silent interpretation and corresponds to (structural) congruences in processes.
- Sequents with a separate unrestricted context are of the form $P \vdash \Gamma;\Delta$, which can also be written as $P \vdash \Delta, \Gamma'$ where Γ' contains only types of the form $!A$.
- The identity axiom can be interpreted as the forwarding process, which enables to account for polymorphism. The forwarding process, however, is not usually present in session π-calculi.

$$\frac{}{\Gamma; x : A \vdash_{\mathsf{ILL}} [x \leftrightarrow y] :: y : A}\ (\textsc{Id}) \qquad \frac{\Gamma, u : A; \Delta, x : A \vdash_{\mathsf{ILL}} P :: z : C}{\Gamma, u : A; \Delta \vdash_{\mathsf{ILL}} (\nu x) u\langle x \rangle.P :: z : C}\ (\textsc{Copy})$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{ILL}} P :: x : A \qquad \Gamma; \Delta', x : A \vdash_{\mathsf{ILL}} Q :: z : C}{\Gamma; \Delta, \Delta' \vdash_{\mathsf{ILL}} (\nu x)(P \mid Q) :: z : C}\ (\textsc{Cut})$$

$$\frac{\Gamma; \cdot \vdash_{\mathsf{ILL}} P :: x : A \qquad \Gamma, u : A; \Delta \vdash_{\mathsf{ILL}} Q :: z : C}{\Gamma; \Delta \vdash_{\mathsf{ILL}} (\nu u)(!u(x).P \mid Q) :: z : C}\ (\textsc{Cut}^!)$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{ILL}} P :: z : C}{\Gamma; \Delta, x : \mathbf{1} \vdash_{\mathsf{ILL}} x().P :: z : Z}\ (\mathbf{1}\mathrm{L}) \qquad \frac{}{\Gamma; \cdot \vdash_{\mathsf{ILL}} x\langle\rangle.\mathbf{0} :: x : \mathbf{1}}\ (\mathbf{1}\mathrm{R})$$

$$\frac{\Gamma; \Delta, y : A, x : B \vdash_{\mathsf{ILL}} P :: z : C}{\Gamma; \Delta, x : A \otimes B \vdash_{\mathsf{ILL}} x(y).P :: z : C}\ (\otimes\mathrm{L}) \qquad \frac{\Gamma; \Delta \vdash_{\mathsf{ILL}} P :: y : A \qquad \Gamma; \Delta' \vdash_{\mathsf{ILL}} Q :: x : B}{\Gamma; \Delta, \Delta' \vdash_{\mathsf{ILL}} (\nu y) x\langle y \rangle.(P \mid Q) :: x : A \otimes B}\ (\otimes\mathrm{R})$$

$$\frac{\Gamma; \Delta \vdash_{\mathsf{ILL}} P :: y : A \qquad \Gamma; \Delta', x : B \vdash_{\mathsf{ILL}} Q :: z : C}{\Gamma; \Delta, \Delta', x : A \multimap B \vdash_{\mathsf{ILL}} (\nu y) x\langle y \rangle.(P \mid Q) :: z : C}\ (\multimap\mathrm{L}) \qquad \frac{\Gamma; \Delta, y : A \vdash_{\mathsf{ILL}} P :: x : B}{\Gamma; \Delta \vdash_{\mathsf{ILL}} x(y).P :: x : A \multimap B}\ (\multimap\mathrm{R})$$

$$\frac{\Gamma, u : A; \Delta \vdash_{\mathsf{ILL}} P :: z : C}{\Gamma; \Delta, x : !A \vdash_{\mathsf{ILL}} P\{x/u\} :: z : C}\ (!\mathrm{L}) \qquad \frac{\Gamma; \cdot \vdash_{\mathsf{ILL}} P :: y : A}{\Gamma; \cdot \vdash_{\mathsf{ILL}} !x(y).P :: x : !A}\ (!\mathrm{R})$$

Figure 2: The $\pi\mathsf{ILL}$ type system

$$\frac{}{[x \leftrightarrow y] \vdash_{\mathsf{CLL}} \Gamma; x : A, y : A^{\perp}}\ (\textsc{Id}) \qquad \frac{P \vdash_{\mathsf{CLL}} \Gamma, u : A; \Delta, y : A}{(\nu y) u\langle y \rangle.P \vdash_{\mathsf{CLL}} \Gamma, u : A; \Delta}\ (\textsc{Copy})$$

$$\frac{P \vdash_{\mathsf{CLL}} \Gamma; \Delta, x : A \qquad Q \vdash_{\mathsf{CLL}} \Gamma; \Delta', x : A^{\perp}}{(\nu x)(P \mid Q)}\ (\textsc{Cut})$$

$$\frac{P \vdash_{\mathsf{CLL}} \Gamma; x : A^{\perp} \qquad Q \vdash_{\mathsf{CLL}} \Gamma, u : A; \Delta}{(\nu u)(!u(x).P \mid Q) \vdash_{\mathsf{CLL}} \Gamma; \Delta}\ (\textsc{Cut}^?)$$

$$\frac{P \vdash_{\mathsf{CLL}} \Gamma; \Delta}{x().P \vdash_{\mathsf{CLL}} \Gamma; \Delta, x : \perp}\ (\perp) \qquad \frac{}{x\langle\rangle.\mathbf{0} \vdash_{\mathsf{CLL}} \Gamma; x : \mathbf{1}}\ (\mathbf{1})$$

$$\frac{P \vdash_{\mathsf{CLL}} \Gamma; \Delta, y : A \qquad Q \vdash_{\mathsf{CLL}} \Gamma; \Delta', x : B}{(\nu y) x\langle y \rangle.(P \mid Q) \vdash_{\mathsf{CLL}} \Gamma; \Delta, \Delta', x : A \otimes B}\ (\otimes) \qquad \frac{P \vdash_{\mathsf{CLL}} \Gamma; \Delta, y : A, x : B}{x(y).P \vdash_{\mathsf{CLL}} \Gamma; \Delta, x : A \parr B}\ (\parr)$$

$$\frac{P \vdash_{\mathsf{CLL}} \Gamma, u : A; \Delta}{P\{x/u\} \vdash_{\mathsf{CLL}} \Gamma; \Delta, x : ?A}\ (?) \qquad \frac{P \vdash_{\mathsf{CLL}} \Gamma; y : A}{!x(y).P \vdash_{\mathsf{CLL}} \Gamma; x : !A}\ (!)$$

Figure 3: The $\pi\mathsf{CLL}^{\mathsf{CP}}$ type system

Judgments. Before we study the expressivity of these three systems, it is important to take note of the difference in the forms of their typing judgments. For $\pi\mathsf{ULL}$, $\pi\mathsf{CLL}^{\mathsf{CP}}$, and $\pi\mathsf{ILL}$, respectively, they are as follows:

$$\Gamma;\Delta \vdash P :: \Lambda \qquad\qquad P \vdash_{\mathsf{CLL}} \Gamma;\Delta \qquad\qquad \Gamma;\Delta \vdash_{\mathsf{ILL}} x : A$$

$\pi\mathsf{CLL}^{\mathsf{CP}}$ has one-sided sequents, whereas $\pi\mathsf{ILL}$ has two-sided sequents (like $\pi\mathsf{ULL}$), but with exactly one channel/type pair on the right of the turnstile. Compare, for example, $\pi\mathsf{CLL}^{\mathsf{CP}}$'s inference rule for $\otimes$ and the $\otimes$R rules for $\pi\mathsf{ILL}$ and $\pi\mathsf{ULL}$ (see Figure 1):

$$\frac{P \vdash_{\mathsf{CLL}} \Gamma;\Delta,y:A \qquad Q \vdash_{\mathsf{CLL}} \Gamma;\Delta',x:B}{(\nu y)x\langle y\rangle.(P \mid Q) \vdash_{\mathsf{CLL}} \Gamma;\Delta,\Delta',x:A\otimes B}\,(\otimes) \qquad \frac{\Gamma;\Delta \vdash_{\mathsf{ILL}} P :: y:A \qquad \Gamma;\Delta' \vdash_{\mathsf{ILL}} Q :: x:B}{\Gamma;\Delta,\Delta' \vdash_{\mathsf{ILL}} (\nu y)x\langle y\rangle.(P \mid Q) :: x : A\otimes B}\,(\otimes\mathrm{R})$$

Locality. Locality is a well-known principle in concurrency research [13]. The idea is that freshly created channels are *local*. Local channels are *modifiable*, in the sense that they can be used for inputs. Once a channel has been transmitted to another location, it becomes *non-local*, and thus *immutable*: it can only be used for outputs—inputs are no longer allowed. This makes locality particularly relevant for giving formal semantics to distributed programming languages; a prime example is the *join calculus* [6], whose theory relies on (and is deeply influenced by) the locality principle [7].

$\pi\mathsf{ILL}$ guarantees locality for shared channels: a server can be defined using a replicated input, so the channel on which this server would be provided cannot be received earlier in the process. Consider the following example, taken from [4]:

$$(\nu x)(x(y).!y(z).P \mid (\nu q)x\langle q\rangle.Q)$$

Let us attempt to find a typing for P in this process using $\pi\mathsf{ILL}$. We can apply the cut rule to split the parallel composition, of which the left component is $x(y).!y(z).P$. Now, there are two rules we can apply (read bottom-up):

$$\frac{\Gamma;\Delta,y:A,x:B \vdash_{\mathsf{ILL}} !y(z).P :: w:C}{\Gamma;\Delta,x:A\otimes B \vdash_{\mathsf{ILL}} x(y).!y(z).P :: w:C}\,(\otimes\mathrm{L}) \qquad \frac{\Gamma;\Delta,y:A \vdash_{\mathsf{ILL}} !y(z).P :: x:B}{\Gamma;\Delta \vdash_{\mathsf{ILL}} x(y).!y(z).P :: x:A\multimap B}\,(\multimap\mathrm{R})$$

In both cases, the received channel y ends up on the left of the turnstile. There are no rules in $\pi\mathsf{ILL}$ to define a service on a channel on the left and there is no way to move the channel to the right. Hence, we cannot find a typing for P in $\pi\mathsf{ILL}$. In contrast, this process can be typed in $\pi\mathsf{CLL}^{\mathsf{CP}}$ as follows, given $P = P'\{x/u\}$:

$$\dfrac{\dfrac{\dfrac{P' \vdash_{\mathsf{CLL}} \Gamma,u:B;z:A}{!y(z).P' \vdash_{\mathsf{CLL}} \Gamma,u:B;y:!A}\,(!)}{!y(z).P \vdash_{\mathsf{CLL}} \Gamma;y:!A,x:?B}\,(?)}{x(y).!y(z).P \vdash_{\mathsf{CLL}} \Gamma;x:(!A)\parr(?B)}\,(\parr)$$

Symmetry. The rely-guarantee distinction of $\pi\mathsf{ILL}$ is what makes it enforce locality for shared channels. The distinction is visible in $\pi\mathsf{ILL}$'s distinction between left and right in its judgments. Despite the two-sidedness of its sequents, $\pi\mathsf{ULL}$ can also type non-local processes. This is due to the full symmetry in the inference rules: anything that can be done on the left of the turnstile can be done on the right. As we will show formally in the rest of this section, this symmetry corresponds to the single-sidedness of $\pi\mathsf{CLL}^{\mathsf{CP}}$: $\pi\mathsf{ULL}$ and $\pi\mathsf{CLL}^{\mathsf{CP}}$ can type exactly the same processes. We will also show formally that $\pi\mathsf{ULL}$, and thus $\pi\mathsf{CLL}^{\mathsf{CP}}$, can type more processes than $\pi\mathsf{ILL}$, because of the restriction of the right side of the turnstile to exactly one channel/type pair, making $\pi\mathsf{ILL}$ an asymmetrical typing system.

Formal results. Our formal results rely on the sets of processes typable in the three typing systems, given in Definition 3.1.

Definition 3.1. *Let $\mathbb{P}$ denote the set of all processes induced by Definition 2.3. Then*

$$\mathfrak{U} = \{P \in \mathbb{P} \mid \exists \Gamma, \Delta, \Lambda \textit{ such that } \Gamma; \Delta \vdash P :: \Lambda\},$$
$$\mathfrak{C} = \{P \in \mathbb{P} \mid \exists \Gamma, \Delta \textit{ such that } P \vdash_{\mathsf{CLL}} \Gamma; \Delta\},$$
$$\mathfrak{I} = \{P \in \mathbb{P} \mid \exists \Gamma, \Delta, x \in \mathit{fn}(P), A \textit{ such that } \Gamma; \Delta \vdash_{\mathsf{ILL}} P :: x : A\}.$$

Our first observation is that all $\pi\mathsf{ULL}$-typable processes are $\pi\mathsf{CLL}^{\mathsf{CP}}$-typable. Moreover, the reverse is true as well. For the latter fact we need to convert a typing inference with one-sided sequents to a two-sided system, which means that we need the ability to control the side of the turnstile specific propositions end up on. This is taken care of by Lemma 3.1, which is an extension of [4, Prop.5.1, p.19] to include exponential types (! and ?). The main result, Theorem 3.2, is that $\pi\mathsf{CLL}^{\mathsf{CP}}$ and $\pi\mathsf{ULL}$ type exactly the same processes.

Lemma 3.1. *For any typing contexts $\Gamma, \Delta, \Lambda, \Pi$ and process P such that $\Gamma; \Delta \vdash P :: \Lambda, \Pi$, we have $\Gamma; \Delta, \Pi^{\perp} \vdash P :: \Lambda$.*

Proof (sketch). By induction on the structure of the type inference tree. If the last inferred proposition is to be moved to the left, after type inversion, the appropriate left rule can be used where a right rule was used. Other propositions can be moved using the induction hypothesis. □

Theorem 3.2. $\mathfrak{U} = \mathfrak{C}$.

Proof (sketch). There are two things to prove: (i) $\mathfrak{U} \subseteq \mathfrak{C}$, and (ii) $\mathfrak{C} \subseteq \mathfrak{U}$. (i) can be proven by induction on the structure of the typing inferences. The idea is that for every rule of $\pi\mathsf{ULL}$ there is an analogous rule in $\pi\mathsf{CLL}^{\mathsf{CP}}$. As for (ii), for any P such that $P \vdash_{\mathsf{CLL}} \Gamma; \Delta$, it suffices to show that there are $\Delta_L, \Delta_R = \Delta$ such that $\Gamma^{\perp}, (\Delta_L)^{\perp} \vdash P :: \Delta_R$. This can be done by induction on the structure of the typing inference of P. After type inversion, the induction hypothesis can be used, which moves channel/type pairs to either side of the turnstile. This results in many subcases, each of which can be solved with appropriate applications of $\pi\mathsf{ULL}$ rule analogous to the last used $\pi\mathsf{CLL}^{\mathsf{CP}}$ rule, using Lemma 3.1 in some cases. □

The comparison between $\pi\mathsf{ULL}$ and $\pi\mathsf{ILL}$ can be done similarly. However, it is more interesting to examine the set of inference rules. If we restrict all sequents in a $\pi\mathsf{ULL}$ typing inference to have exactly one channel/type on the right of the turnstile, we end up with a subset of usable inference rules: those marked with an '$*$' in Figure 1. Upon further examination, we see that this set of rules coincides exactly with the set of inference rules for $\pi\mathsf{ILL}$. The consequence is that $\pi\mathsf{ULL}$ can type all $\pi\mathsf{ILL}$-typable processes. Finally, any $\pi\mathsf{ULL}$/$\pi\mathsf{CLL}^{\mathsf{CP}}$-typable process violating the locality principle suffices to show the second main result, Theorem 3.3: $\pi\mathsf{ILL}$ is less expressive than $\pi\mathsf{ULL}$. An important corollary of Theorem 3.2 and Theorem 3.3 is that $\pi\mathsf{ILL}$ is less expressive than $\pi\mathsf{CLL}^{\mathsf{CP}}$, confirming the observation by Caires, Pfenning, and Toninho [4].

Definition 3.2 (r-degree)**.** *The size of the right side of a $\pi\mathsf{ULL}$ sequent is the sequent's r-*degree*. Given contexts Γ, Δ, Λ and process P, the ULL sequent $\Gamma; \Delta \vdash P :: \Lambda$ has r-degree $|\Lambda|$.*

Theorem 3.3. $\mathfrak{I} \subset \mathfrak{U}$.

Proof (sketch). Let

$$\mathfrak{U}^* := \{P \in \mathfrak{U} \mid \exists \Gamma, \Delta, A \text{ s.t. } \Gamma; \Delta \vdash P :: x : A \text{ with a proof tree containing only sequents of } r\text{-degree } 1\}.$$

By induction on the structure of the typing inference, it can be shown that all processes in $\mathfrak{U}^*$ have typing inferences using only those rules in Figure 1 marked with an $*$. It follows by contradiction that the last applied rule cannot be without an $*$, and the usage of $*$-marked rules in the rest of the inference follows from the induction hypothesis. The $*$-marked rules with r-degree 1 coincide exactly with the typing inference rules of πILL, and so it follows that $\mathfrak{I} = \mathfrak{U}^*$. Clearly, $\mathfrak{U}^*$ is a subset of $\mathfrak{U}$. Now, given $u : B; \cdot \vdash P :: z : A$, there are several ways to type the process $x(y).!y(z).P$ in πULL, but all of them use sequents of r-degree different from 1, so it is not in $\mathfrak{U}^*$. Hence, $\mathfrak{U}^* \subset \mathfrak{U}$, confirming the thesis. □

4 Conclusion

Using Girard's Logic of Unity [9] as a basis, we have developed United Linear Logic as a means to formally compare the session type systems derived from concurrent interpretations of classical and intuitionistic linear logic. Much as Logic of Unity is a logic that encompasses classical, intuitionistic and linear logic, the session type system interpretation of ULL (dubbed πULL), can type all πCLL$^{\mathsf{CP}}$- and πILL-typable processes. This allows us to compare type systems based on different linear logics.

In πILL, judgments distinguish between several channels whose behavior is being relied on (on the left of the judgment) and a single behavior provided along a designated channel (on the right). To retain this rely-guarantee reading, πULL uses two-sided sequents in its typing inferences. However, πULL does not distinguish between the sides of its sequents; it is fully symmetrical. This symmetry allows it to mimic the single-sidedness of πCLL$^{\mathsf{CP}}$'s sequents: placing a mirror besides πCLL$^{\mathsf{CP}}$'s inference rules reveals the inference rules of πULL. Similarly, πILL can be recovered from πULL by restricting the right side of its sequents to exactly one channel. Not all inference rules remain usable, resulting in an asymmetrical system that coincides exactly with πILL.

Our results formally corroborate the observation by Caires, Pfenning and Toninho: the difference between session type systems based on classical and intuitionistic linear logic is in the enforcement of locality [4]. πCLL$^{\mathsf{CP}}$ is able to type non-local processes, because it does not distinguish between linear channels, received or not, because of its single-sided sequents. Similarly, πULL can type non-local processes because of its symmetry. Restricting πULL into πILL removes exactly those rules needed to violate the locality principle. This reveals that πILL respects locality because of its asymmetry.

Acknowledgements. We are grateful to Joseph Paulus for initial discussions. We would also like to thank the anonymous reviewers for their suggestions, which were helpful to improve the presentation.

References

[1] Andrew Barber & Gordon D. Plotkin (1996): *Dual Intuitionistic Linear Logic*. Technical Report ECS-LFCS-96-347, University of Edinburgh, School of Informatics, Laboratory for Foundations of Computer Science, Edinburgh.

[2] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[3] Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Towards Concurrent Type Theory*. In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, ACM, pp. 1–12, doi:10.1145/2103786.2103788.

[4] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear Logic Propositions as Session Types*. *Mathematical Structures in Computer Science* 26(3), pp. 367–423, doi:10.1017/S0960129514000218.

[5] Bor-Yuh Evan Chang, Kaustuv Chaudhuri & Frank Pfenning (2003): *A Judgmental Analysis of Linear Logic*. Technical Report CMU-CS-03-131R, Department of Computer Science, Carnegie Mellon University, doi:10.1184/R1/6587498.v1.

[6] Cédric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget & Didier Rémy (1996): *A Calculus of Mobile Agents*. In Ugo Montanari & Vladimiro Sassone, editors: *CONCUR '96: Concurrency Theory*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 406–421, doi:10.1007/3-540-61604-7_67.

[7] Cédric Fournet & Cosimo Laneve (2001): *Bisimulations in the Join-Calculus*. *Theoretical Computer Science* 266(1), pp. 569–603, doi:10.1016/S0304-3975(00)00283-8.

[8] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1–101, doi:10.1016/0304-3975(87)90045-4.

[9] Jean-Yves Girard (1993): *On the Unity of Logic*. *Annals of Pure and Applied Logic* 59(3), pp. 201–217, doi:10.1016/0168-0072(93)90093-S.

[10] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR'93*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[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] Olivier Laurent (2018): *Around Classical and Intuitionistic Linear Logics*. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, ACM, Oxford, United Kingdom, pp. 629–638, doi:10.1145/3209108.3209132.

[13] Massimo Merro (2000): *Locality and Polyadicity in Asynchronous Name-Passing Calculi*. In Jerzy Tiuryn, editor: *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 238–251, doi:10.1007/3-540-46432-8_16.

[14] Massimo Merro & Davide Sangiorgi (2004): *On Asynchrony in Name-Passing Calculi*. *Mathematical Structures in Computer Science* 14(5), pp. 715–767, doi:10.1017/S0960129504004323.

[15] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I*. *Information and Computation* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[16] Davide Sangiorgi & David Walker (2003): *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

[17] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-Based Language and Its Typing System*. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou & Sergios Theodoridis, editors: *PARLE'94 Parallel Architectures and Languages Europe*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[18] Philip Wadler (2012): *Propositions As Sessions*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, ACM, Copenhagen, Denmark, pp. 273–286, doi:10.1145/2364527.2364568.

Generating Interactive WebSocket Applications in TypeScript

Anson Miu
Imperial College London

Francisco Ferreira
Imperial College London

Nobuko Yoshida
Imperial College London

Fangyi Zhou
Imperial College London

Advancements in mobile device computing power have made interactive web applications possible, allowing the web browser to render contents dynamically and support low-latency communication with the server. This comes at a cost to the developer, who now needs to reason more about correctness of communication patterns in their application as web applications support more complex communication patterns.

Multiparty session types (MPST) provide a framework for verifying conformance of implementations to their prescribed communication protocol. Existing proposals for applying the MPST framework in application developments either neglect the event-driven nature of web applications, or lack compatibility with industry tools and practices, which discourages mainstream adoption by web developers.

In this paper, we present an implementation of the MPST framework for developing interactive web applications using familiar industry tools using TypeScript and the *React.js* framework. The developer can use the Scribble protocol language to specify the protocol and use the Scribble toolchain to validate and obtain the *local protocol* for each role. The local protocol describes the interactions of the global communication protocol observed by the role. We encode the local protocol into TypeScript types, catering for server-side and client-side targets separately. We show that our encoding guarantees that only implementations which conform to the protocol can type-check. We demonstrate the effectiveness of our approach through a web-based implementation of the classic *Noughts and Crosses* game from an MPST formalism of the game logic.

1 Introduction

Modern interactive web applications aim to provide a highly responsive user experience by minimising the communication latency between clients and servers. Whilst the HTTP request-response model is sufficient for retrieving static assets, applying the same stateless communication approach for interactive use cases (such as real-time multiplayer games) introduces undesirable performance overhead. Developers have since adopted other communication transport abstractions over HTTP connections such as the WebSockets protocol [7] to enjoy low-latency full-duplex client-server communication in their applications over a single persistent connection. Enabling more complex communication patterns caters for more interactive use cases, but introduces additional correctness concerns to the developer.

Consider a classic turn-based board game of *Noughts and Crosses* between two players. Both players, identified by either *noughts (O's)* or *crosses (X's)* respectively, take turns to place a mark on an unoccupied cell of a 3-by-3 grid until one player wins (when their markers form one straight line on the board) or a stalemate is reached (when all cells are occupied and no one wins). A web-based implementation may involve players connected to a game server via WebSocket connections. The players interact with the game from their web browser, which shows a *single-page application* (SPA) of the game client written in a popular framework like *React.js* [20]. SPAs feature a single HTML page and dynamically render content via JavaScript in the browser. Players take turns to make a move on the game board, which sends a message to the server. The server implements the game logic to progress the game forward until

S. Balzer, L. Padovani (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 12–22, doi:10.4204/EPTCS.314.2

a result (either a win/loss or draw) can be declared, where either the move of the other player or the game result is sent to players.

Whilst WebSockets make this web-based implementation possible, they introduce the developer to a new family of communication errors, even for this simple game. In addition to the usual testing for game logic correctness, the developer needs to test against *deadlocks* (e.g. both players waiting for each other to make a move at the same time) and *communication mismatches* (e.g. player 1 sending a boolean to the game server instead of the board coordinates). The complexity of these errors correlates to the complexity of the required tests and scales with the complexity of communication patterns involved.

Multiparty Session Types (MPST) [5] provide a framework for formally specifying a structured communication pattern between concurrent processes and verifying implementations for correctness with respect to the communications aspect. By specifying the client-server interactions of our game as an MPST protocol and verifying the implementations against the protocol for conformance, MPST theory guarantees well-formed implementations to be free from communication errors.

We see the application of the MPST methodology to generating interactive TypeScript web applications to be an interesting design space — to what extent can the MPST methodology be applied to deliver a workflow where developers use the generated TypeScript APIs in their application to guarantee protocol conformance by construction? Such a workflow would ultimately decrease the overhead for incorporating MPST into mainstream web development, which reduces development time by programmatically verifying communication correctness of the implementation.

Contributions This paper presents a workflow for developing type-safe interactive SPAs motivated by the MPST framework: **(1)** An endpoint API code generation workflow targeting TypeScript-based web applications for multiparty sessions; **(2)** An encoding of session types in server-side TypeScript that enforces static linearity; and **(3)** An encoding of session types in browser-side TypeScript using the React framework that guarantees affine usage of communication channels.

2 The Scribble Framework

Development begins with specifying the expected communications between participants as a *global protocol* in Scribble [23], a MPST-based protocol specification language and code generation toolchain. We specify the *Noughts and Crosses* game as a Scribble protocol in Listing 1. In the protocol, the role `Svr` stands for the Server, and the roles `P1` and `P2` stand for the two Players respectively.

We leverage the Scribble toolchain to check for protocol well-formedness. This directly corresponds to multiparty session type theory [16]: a Scribble protocol maps to some *global type*, and the Scribble toolchain implements the algorithmic projection defined in [5] to derive valid local type *projections* for all participants. We obtain a set of *endpoint protocols* (corresponds to *local types*) — one for each role from a well-formed global protocol. An endpoint protocol only preserves the interactions defined by the global protocol in which the target role is involved, and corresponds to an equivalent *Endpoint Finite State Machine* (EFSM) [6]. The EFSM holds information about the permitted IO actions for the role. We use the EFSMs as a basis for API generation and adopt the formalisms in [11].

3 Encoding Session Types in TypeScript

Developers can implement their application using APIs generated from the EFSM to guarantee correctness by construction. Our approach integrates the EFSM into the development workflow by encoding

```
module NoughtsAndCrosses;
type <typescript> "Coordinate" from "./Types" as Point; // Position on board

global protocol Game(role Svr, role P1, role P2) {
  Pos(Point) from P1 to Svr;
  choice at Svr {
    Lose(Point) from Svr to P2; Win(Point) from Svr to P1;
  } or {
    Draw(Point) from Svr to P2; Draw(Point) from Svr to P1;
  } or {
    Update(Point) from Svr to P2; Update(Point) from Svr to P1;
    do Game(Svr, P2, P1); // Continue the game with player roles swapped
  }
}
```

Listing 1: *Noughts and Crosses* in a Scribble protocol.

session types as TypeScript types. Communication over the WebSocket protocol introduces additional constraints: communication is always initiated in the front-end and driven by user interactions, whilst back-end roles can only accept connections. This motivates our design of encoding the session types differently for server (Section 3.1) and client (Section 3.3) targets.

3.1 Server-Side API Generation

We refer to the `Svr` EFSM (Figure 1) as a running example in this section. For server-side targets, we encode EFSM states into TypeScript types and consider branching (receiving) and selection (sending) states separately. We assign TypeScript encodings of states to their state identifiers in the EFSM, providing syntactic sugar when referring to the successor state when encoding the current state. For any state S in the EFSM, we refer to the TypeScript type alias of its encoding as $[\![S]\!]$. We outline the encoding below using examples from the *Noughts and Crosses* game (Listing 2).

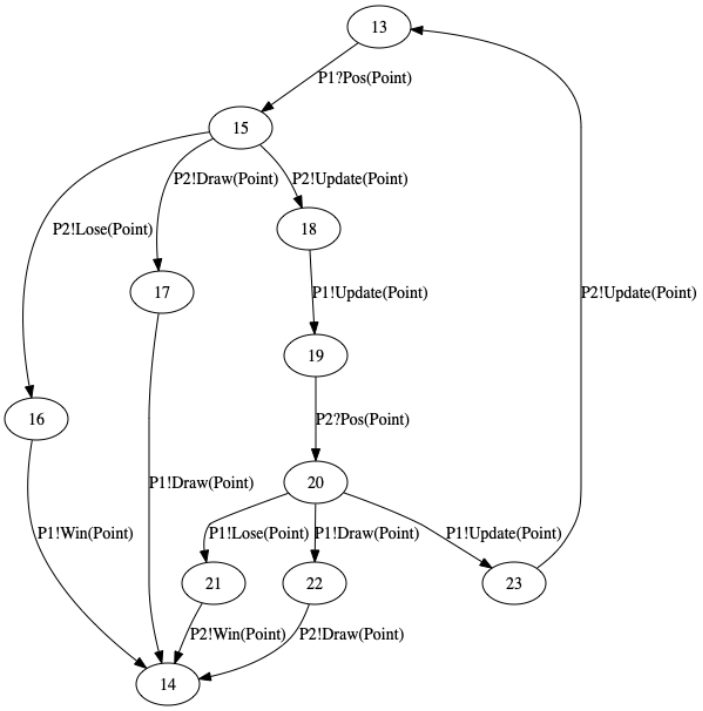


Figure 1: EFSM for `Svr`.

Branching State We consider a receiving state as a unary branching state for conciseness. A branching state is encoded as an *object literal* [18] (a record type), with each branch $i \in I$ (I denoting set of all branches), corresponding to a member field. A branch expecting to receive a message

labelled $\texttt{label}_i$ carrying payload of type $\texttt{T}_i$ with successor state S_i is encoded as an *member field* named $\texttt{label}_i$ of function type $(payload : \texttt{T}_i) \to [\![S_i]\!]$. The developer implements a branching operation by passing callbacks for each branch, parameterised by the expected message payload type for that branch.

Selection State We consider a sending state as a unary selection state for conciseness. A selection state is encoded as a *union type* [18] of internal choice encodings: each internal choice $i \in I$ (I denoting set of all choices), sending a message labelled $\texttt{label}_i$ carrying payload of type $\texttt{T}_i$ with successor state S_i is encoded as a *tuple type* of $\texttt{[Labels.label}_i\texttt{, T}_i\texttt{, } [\![S_i]\!]\texttt{]}$. The developer implements a selection operation by passing the selected label and payload to send in the message. We generate a *string enum* (named `Labels`) wrapping the labels in the protocol.

```
1 export type S13 = { Pos: (payload: Point) => S15 };
2 export type S15 = [ Labels.Lose, Point, S16 ]
3                 | [ Labels.Draw, Point, S17 ]
4                 | [ Labels.Update, Point, S18 ];
```

Listing 2: Example encodings from *Noughts and Crosses* `Svr` EFSM.

In the case of Listing 2, the developer is expected to implement `S13` which handles the `Pos` message sent by `P1`, and the code in `S13` returns a value of type `S15`, which corresponds to a selection of messages to send to `P2`. Listing 4 illustrates how the developer may implement these types.

We make a key design decision *not* to expose communication channels in the TypeScript session type encodings to provide *static* linearity guarantees (Section 3.1.2). Our encoding sufficiently exposes seams for the developer to inject their program logic, whilst the generated session API (Section 3.1.1) handles the sending and receiving of messages.

3.1.1 Session Runtime

The generated code for our session runtime performs communication in a protocol-conformant manner, but does not expose these IO actions to the developer by delegating the aforementioned responsibilities to an inner class. The runtime executes the EFSM by keeping track of the current state (similar to the generated code in [10]) and only permitting the specified IO actions at the current state. The runtime listens to message (receiving) events on the communication channel, invokes the corresponding callback to obtain the value to send next, and performs the sending. The developer instantiates a session by constructing an instance of the session runtime class, providing the WebSocket endpoint URL (to open the connection) and the initial state (to execute the EFSM).

3.1.2 Linear Channel Usage

Developers writing their implementation using the generated APIs enjoy channel linearity by construction. Our library design prevents the two conditions detailed below:

Repeated Usage We do not expose channels to the developer, which makes *reusing channels* impossible. For example, to send a message, the generated API only requires the payload that needs to be sent, and the session runtime performs the send internally, guaranteeing this action is done *exactly once* by construction.

Unused Channels The initial state must be supplied to the session runtime constructor in order to instantiate a session; this initial state is defined in terms of the successor states, which in turn has references to its successors and so forth. The developer's implementation will cover the terminal state (if it exists), and the session runtime guarantees this terminal state will be reached by construction.

3.2 The React Framework

Our browser-side session type encodings for browser-side targets build upon the *React.js* framework, developed by Facebook [20] for the *Model-View-Controller* (MVC) architecture. React is widely used in industry to create scalable single-page TypeScript applications, and we intend for our proposed workflow to be beneficial in an industrial context. We introduce the key features of the framework.

Components A component is a reusable UI element which contains its own markup and logic. Components implement a `render()` function which returns a React element, the smallest building blocks of a React application, analogous to the view function in the MVU architecture. Components can keep *state*s and the `render()` function is invoked upon a change of state.

For example, a simple counter can be implemented as a component, with its `count` stored as state. When rendered, it displays a button which increments `count` when clicked and a `div` that renders the current `count`. If the button is clicked, the `count` is incremented, which triggers a re-rendering (since the state has changed), and the updated `count` is displayed.

Components can also render other components, which gives rise to a parent/child relationship between components. Parents can pass data to children as *props* (short for properties). Going back to the aforementioned example, the counter component could render a child component `<StyledDiv count={this.state.count} />` in its `render()` function, propagating the `count` from its state to the child. This enables reusability, and for our use case, gives control to the parent on what data to pass to its children (e.g. pass the payload of a received message to a child to render).

3.3 Browser-Side API Generation

We refer to the `P1` EFSM (Figure 2) as a running example in this section. Preserving behavioural typing and channel linearity is challenging for browser-side applications due to EFSM transitions being triggered by user events: in the case of *Noughts and Crosses*, once the user makes a move by clicking on a cell on the game board, this click event must be deactivated until the user's next turn, otherwise the user can click again and violate channel linearity. Our design goal is to enforce this statically through the generated APIs.

For browser-side targets, we extend the approach presented in [9] on *multiple model types* motivated by the *Model-View-Update* (MVU) architecture. An MVU application features a *model* encapsulating application state, a *view function* rendering the state on the Document Object Model (DOM), and an *update function* handling *messages* produced by the rendered model to produce a new model. The concept of model types express type dependencies between these components: a *model type* uniquely defines a *view function*, set of *messages* and *update function* – rather than producing a new model, the update function defines valid transitions to other model types. We leverage the correspondence between model types and states in the EFSM: each state in the EFSM is a model type, the set of messages represent the possible (IO) actions available at that state, and the update function defines which successor state to transition to, given the supported IO actions at this state.

3.3.1 Model Types in React

State An EFSM state is encoded as an *abstract* React component. This is an abstract class to require the developer to provide their own view function, which translates conveniently to the `render()` function of React components. Our session runtime (Section 3.3.2) "executes" the EFSM and renders the current state. Upon transitioning to a successor state, the successor's view function will be invoked, as per the semantics expressed in [9].

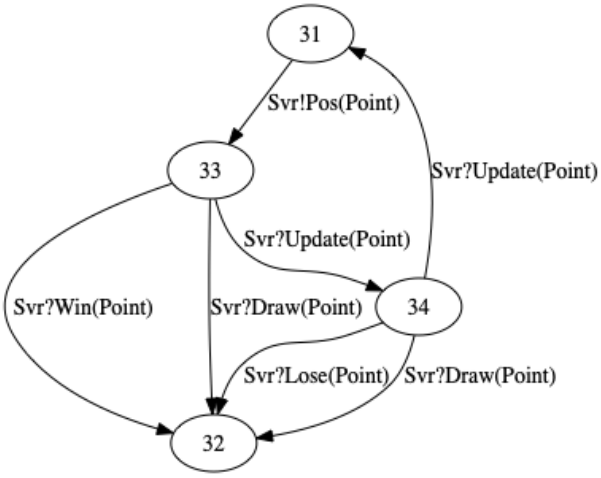


Figure 2: EFSM for `P1`.

Model Transitions Transitions are encoded as React component props onto the encoded states by the session runtime (Section 3.3.2). We motivate the design choice of not exposing channel resources to provide guarantees on channel usage. React components in TypeScript are *generic* [18], parameterised by the permitted types of prop and state. The parameters allow us to leverage the TypeScript compiler to verify that the props for model transitions stay local to the state they are defined for. The model transitions for EFSMs are message send and receive.

Sending We make the assumption that message sending is triggered by some user-driven UI event (e.g. clicking a button, pressing a key on the keyboard) which interacts with some DOM element. We could pass a `send()` function as a prop to the sending state, but the developer would be free to call the function multiple times which makes channel reuse possible. Instead, we pass a *factory function* as a prop, which will, given an HTML event and an event handler function, return a fresh React component that binds the sending action on construction. So once the bound event is triggered, our session runtime executes the event handler function to obtain the payload to send, perform the send *exactly once* and transition to (which, in practice, means render) the successor state.

```
1 // Inside some render() function..
2 {board.map((row, x) => (
3     row.map((col, y) => {
4         const SelectPoint = this.props.Pos('click', (event: UIEvent) => {
5             event.preventDefault();
6             return { x: x, y: y };}
7         return <SelectPoint><td>.</td></SelectPoint>;
8 });}
```

Listing 3: Model transition for message sending in *Noughts and Crosses* `P1` implementation.

We demonstrate the semantics using the *Noughts and Crosses* example in Listing 3. The session runtime passes the factory function `this.props.Pos` as a prop. For each x-y coordinate on the game

board, we create a `SelectPoint` React component from the factory function (which reads "build a React component that sends the `Pos` message with x-y coordinates as payload when the user clicks on it") and we wrap a table cell (the game board is rendered as an HTML table) inside the `SelectPoint` component to bind the click event on the table cell.

Receiving The React component for a receiving state is required to define a handler for each supported branch. Upon a message receive event, the session runtime invokes the handler of the corresponding branch with the message payload and renders the successor state upon completion.

3.3.2 Session Runtime

The session runtime can be interpreted as an abstraction on top of the React VDOM that implements the EFSM by construction. The session runtime itself is a React component too, named after the endpoint role identifier: it opens the WebSocket connection to the server, keeps track of the current EFSM state as part of its React component state, and most importantly, renders the React component encoding of the active EFSM state. Channel communications are managed by the runtime, which allows it to render the successor of a receive state upon receiving a message from the channel. Similarly, the session runtime is responsible for passing the required props for model transitions to EFSM state React components. The session runtime component is rendered by the developer and requires, as props, the *endpoint URL* (so it can open the connection) and a list of *concrete state components*.

The developer writes their own implementation of each state (mainly to customise how the state is rendered and inject business logic into state transitions) by extending the abstract React class components. The session runtime requires references to these concrete components in order to render the user implementation accordingly.

3.3.3 Affine Channel Usage

A limitation of our browser-side session type encoding is only being able to guarantee that channel resources are used *at most once* as opposed to *exactly once*.

Communication channels are not exposed to the developer so multiple sends are impossible. This does not restrict the developer from binding the send action to exactly one UI event: for *Noughts and Crosses*, we bind the `Pos(Point)` send action to each unoccupied cell on the game board, but the generated runtime ensures that, once the cell is clicked, the send is only performed once and the successor state is rendered on the DOM, so the channel resource used to send becomes unavailable.

However, our approach *does not* statically detect whether all transitions in a certain state are bound to some UI event. This means that it is possible for an implementation to *not* handle transitions to a terminal state but still type-check, so we cannot prevent unused states. Equally, our approach does not prevent a client closing the browser, which would drop the connection.

4 Case Study

We apply our framework to implement a web-based implementation of the *Noughts and Crosses* running example in TypeScript; the interested reader can find the full implementation in [14]. In addition to MPST-safety, we show that our library design welcomes idiomatic JavaScript practices in the user implementation and is interoperable with common front- and back-end frameworks.

```
1  const handleP1Move: S13 = (move: Point) => {
2      board.P1(move);          // User logic
3      if (board.won()) {
4          return [Labels.Lose, move, [Labels.Win, move]];
5      } else if (board.draw()) {
6          return [Labels.Draw, move, [Labels.Draw, move]];
7      } else {
8          return [Labels.Update, move, [Labels.Update, move, handleP2Move]];
9      }
10 }
11
12 // Instantiate session - `handleP2Move` defined similarly as S19
13 new NoughtsAndCrosses.Svr(webSocketServer, handleP1Move);
```

Listing 4: Session runtime instantiation for *Noughts and Crosses* `Svr`.

Game Server We set up the WebSocket server as an Express.js [8] application on top of a Node.js [17] runtime. We define our own game logic in a `Board` class to keep track of the game state and expose methods to query the result. This custom logic is integrated into our `handleP1Move` and `handleP2Move` handlers (Listing 4), so the session runtime can handle `Pos(Point)` messages from players and transition to the permitted successor states (Listing 1) according to the injected game logic: if `P1` played a winning move (Line 4), `Svr` sends a `Lose` message to `P2` with the winning move, and also sends a `Win` message to `P1`; if `P1`'s move resulted in a draw (Line 6), `Svr` sends `Draw` messages to both `P2` and `P1`; otherwise, the game continues (Line 8), so `Svr` updates both `P2` and `P1` with the latest move and proceeds to handle `P2`'s turn.

Note that, by TypeScript's structural typing [18], replacing `handleP2Move` on Line 8 with a recursive occurrence of `handleP1Move` would be type-correct — this allows for better code reuse as opposed to defining additional abstractions to work around the limitations of nominal typing in [11]. There is also full type erasure when transpiling to JavaScript to run the server code, so the types defined in TypeScript will not appear in the JavaScript after type-checking. This means state space explosion is not a runtime consideration.

Game Clients We implement the game client for `P1` and `P2` by extending from the generated abstract React (EFSM state) components and registering those to the session runtime component.

For the sake of code reuse, [14] uses *higher-order components* (HOC) to build the correct state implementations depending on which player the user chooses to be. We leverage the *Redux* [1] state management library to keep track of the game state, thus showing the flexibility of our library design in being interoperable with other libraries and idiomatic JavaScript practices. Our approach encourages the separation of concerns between the communication logic and program logic — the generated session runtime keeps track of the state of the EFSM to ensure protocol conformance by construction, whilst *Redux* solely manages our game state.

5 Related Work

The two main approaches for incorporating our MPST workflow into application development are native language support for first-class linear channel resources [22] and code generation. The latter closely

relates to our proposal; we highlight two areas of existing work under this approach that motivate our design choice.

Endpoint API Generation Neykova and Yoshida targeted Python applications and the generation of runtime monitors [15] to dynamically verify communication patterns. Whilst the same approach could be applied to JavaScript, we can provide more static guarantees with TypeScript's gradual typing system. Scribble-Java [11] proposed to encode the EFSM states and transitions as classes and instance methods respectively, with behavioural typing achieved statically by the type system and channel linearity guarantees achieved dynamically since channels are exposed and aliasing is not monitored. Scribble-Java can generate callback-style APIs similar to the approach we present, but this approach is arguably less idiomatic for Java developers.

Session Types in Web Development King et al. [13] targeted web development in PureScript using the *Concur UI* framework and proposed a type-level encoding of EFSMs as multi-parameter type classes. However, it presents a trade-off between achieving static linearity guarantees from the type-level EFSM encoding under the expressive type system and providing an intuitive development experience to developers, especially given the prevalence of JavaScript and TypeScript applications in industry. Fowler [9] focused on applying binary session types in front-end web development and presented approaches that tackle the challenge of guaranteeing linearity in the event-driven environment, whereas our work is applicable to multiparty scenarios.

Our work applies the aforementioned approaches in a *multiparty* context using industrial tools and practices to ultimately encourage MPST-safe web application development workflows in industry.

6 Conclusion and Future Work

We have presented an MPST-based framework for developing full-stack interactive TypeScript applications with WebSocket communications. The implementation conforms to a specified protocol, statically providing linear channel usage guarantees and affine channel usage guarantees for back-end and front-end targets respectively.

Future work includes incorporating *explicit connection actions* introduced in [12] in our API generation to better model real-world communication protocols that may feature in interactive web applications. Server-side implementations may perform asynchronous operations on the received messages, so supporting asynchronous values (such as JavaScript *Promises* [3]) in IO actions would be a welcome addition. Whilst our approach supports multiparty sessions, the nature of WebSockets require some server-based role in the communication protocol and clients to interact via the server. Extending support to WebRTC [21] would cater for peer-to-peer communication between browsers, which further opens up possibilities for communication protocols supported by our approach.

Acknowledgements

We thank the anonymous reviewers for their feedback.
This work was supported in part by EPSRC projects EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, and EP/T006544/1.

References

[1] Dan Abramov (2015): *Redux - A predictable state container for JavaScript apps.* Available at `https://redux.js.org/`.

[2] Gavin Bierman, Martn Abadi & Mads Torgersen (2014): *Understanding TypeScript*. In RichardEditor Jones, editor: *ECOOP 2014 Object-Oriented Programming*, Lecture Notes in Computer Science, Springer, pp. 257–281, doi:10.1007/978-3-662-44202-9_11.

[3] MDN contributors (2020): *Promise*. Available at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`. Library Catalog: developer.mozilla.org.

[4] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2007): *Links: Web Programming Without Tiers*, pp. 266–296. 4709, Springer Berlin Heidelberg, doi:10.1007/978-3-540-74792-5_12.

[5] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani & Nobuko Yoshida (2015): *A Gentle Introduction to Multiparty Asynchronous Session Types*. In: *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming*, *LNCS* 9104, Springer, pp. 146–178, doi:10.1007/978-3-319-18941-3_4.

[6] Pierre-Malo Denilou & Nobuko Yoshida (2013): *Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types*. In: *40th International Colloquium on Automata, Languages and Programming*, *LNCS* 7966, Springer, Berlin, Heidelberg, pp. 174–186, doi:10.1007/978-3-642-39212-2_18.

[7] I. Fette & A. Melnikov (2011): *The WebSocket Protocol*. RFC 6455, RFC Editor. Available at `https://www.rfc-editor.org/rfc/rfc1654.txt`.

[8] Node.js Foundation: *Express - Node.js web application framework*. Available at `https://expressjs.com/`.

[9] Simon Fowler (2019): *Model-View-Update-Communicate: Session Types meet the Elm Architecture*. *arXiv:1910.11108 [cs]*. Available at `http://arxiv.org/abs/1910.11108`. ArXiv: 1910.11108.

[10] Rosita Gerbo & Luca Padovani (2019): *Concurrent Typestate-Oriented Programming in Java*. *Electronic Proceedings in Theoretical Computer Science* 291, pp. 24–34, doi:10.4204/EPTCS.291.3. ArXiv: 1904.01286.

[11] Raymond Hu & Nobuko Yoshida (2016): *Hybrid Session Verification through Endpoint API Generation*. In: *19th International Conference on Fundamental Approaches to Software Engineering*, *LNCS* 9633, Springer, pp. 401–418, doi:10.1007/978-3-662-49665-7_24.

[12] Raymond Hu & Nobuko Yoshida (2017): *Explicit Connection Actions in Multiparty Session Types*, pp. 116–133. 10202, Springer Berlin Heidelberg, doi:10.1007/978-3-662-54494-5_7.

[13] Jonathan King, Nicholas Ng & Nobuko Yoshida (2019): *Multiparty Session Type-safe Web Development with Static Linearity*. *Electronic Proceedings in Theoretical Computer Science* 291, pp. 35–46, doi:10.4204/EPTCS.291.4.

[14] Anson Miu (2020): *ansonmiu0214/scribble-noughts-and-crosses*. Available at `https://github.com/ansonmiu0214/scribble-noughts-and-crosses`.

[15] Rumyana Neykova & Nobuko Yoshida (2017): *How to Verify Your Python Conversations*. *Behavioural Types: from Theory to Tools*, pp. 77–98, doi:10.13052/rp-9788793519817.

[16] Rumyana Neykova & Nobuko Yoshida (2019): *Featherweight Scribble*, pp. 236–259. 11665, Springer International Publishing, doi:10.1007/978-3-030-21485-2_14.

[17] Node.js: *Node.js*. Available at `https://nodejs.org/en/`.

[18] Microsoft Research: *TypeScript Language Specification*. Available at `https://github.com/microsoft/TypeScript`.

[19] Facebook Open Source: *Introducing JSX React*. Available at `https://reactjs.org/docs/introducing-jsx.html`.

[20] Facebook Open Source: *React A JavaScript library for building user interfaces*. Available at `https://reactjs.org/`.

[21] Justin Uberti & Peter Thatcher (2011): *WebRTC*. Available at `https://webrtc.org/`.

[22] Hongwei Xi (2017): *Applied Type System: An Approach to Practical Programming with Theorem-Proving*. *arXiv:1703.08683 [cs]*. Available at `http://arxiv.org/abs/1703.08683`. ArXiv: 1703.08683.

[23] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2013): *The Scribble Protocol Language*. In: *8th International Symposium on Trustworthy Global Computing*, *LNCS* 8358, Springer, pp. 22–41, doi:10.1007/978-3-319-05119-2_3.

Duality of Session Types: The Final Cut

Simon J. Gay
School of Computing Science
University of Glasgow, UK
Simon.Gay@glasgow.ac.uk

Peter Thiemann
Institut für Informatik
University of Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Vasco T. Vasconcelos
Faculdade de Ciências
University of Lisbon, Portugal
vmvasconcelos@ciencias.ulisboa.pt

Duality is a central concept in the theory of session types. Since a flaw was found in the original definition of duality for recursive types, several other definitions have been published. As their connection is not obvious, we compare the competing definitions, discuss tradeoffs, and prove some equivalences. Some of the results are mechanized in Agda.

1 Introduction

Duality is a central concept in the theory of session types. If S is a session type describing a two-party interaction from the viewpoint of one party, then $\overline{S}$ describes the interaction from the viewpoint of the other party. For example, $S = \mu X.?\mathsf{int}.X$ describes indefinitely receiving integers, and its dual $\overline{S} = \mu X.!\mathsf{int}.X$ describes indefinitely sending integers. If the users of the two endpoints of a channel follow types S and $\overline{S}$, respectively, then correct communication takes place.

The original papers on session types [7, 8, 11] define the dual $\overline{S}$ of a session type S by structural induction on S:

$$\overline{\mathsf{end}} = \mathsf{end} \qquad \overline{!T.S} = ?T.\overline{S} \qquad \overline{?T.S} = !T.\overline{S}$$

Recursion is only introduced in the last paper in the series, [8], where recursive session types are handled via the following rules.

$$\overline{X} = X \qquad \overline{\mu X.S} = \mu X.\overline{S}$$

With this definition, indeed we have $\overline{\mu X.?\mathsf{int}.X} = \mu X.!\mathsf{int}.X$, given that duality exchanges input and output. Gay & Hole [5, 6] define a more general duality *relation* $\perp$ so that, for example, $\mu X.?\mathsf{int}.X \perp \mu X.!\mathsf{int}.!\mathsf{int}.X$. The definition is coinductive. This relation gives greater flexibility in typing derivations and follows the idea that duality is a behavioural relation on automata. The relationship between the duality function and the duality relation is intended to be that $\overline{S} \perp S$ for every session type S.

Bernardi & Hennessy [3, 4] show that the duality function $\overline{(\cdot)}$ violates the duality relation for recursive session types when the recursion variable can occur as the type of a message, as in $S = \mu X.?X.X$. In this example, we have $\overline{S} = \mu X.!X.X$. Noting that an occurrence of X stands for the whole μ type, the type of the message in S is S but the type of the message in $\overline{S}$ is $\overline{S}$. In other words, we have dual types in which the type of the message being sent is not the same as the type of the message being received, which violates soundness of any type system that uses this definition of duality. We refer to $\overline{(\cdot)}$ as *naive duality* because it initially seems reasonable but is not correct in all situations.

One way to solve this problem is to require that recursion variables only occur in tail position in a session type, such as X in $?T.X$. As far as we know, almost all papers that use naive duality can be

S. Balzer, L. Padovani (Eds.): Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 23–33, doi:10.4204/EPTCS.314.3

fixed by restricting to tail recursion, because their examples and applications are all tail recursive. One exception is a paper by Vasconcelos [12], which has an interesting application of the type $\mu X.!X.X$ to encode replication, but that could be easily solved with a tail recursive type at the expense of creating an extra channel. Bernardi & Hennessy give examples of pi-calculus processes that can only be typed by using non-tail-recursive session types, but they are specially crafted for the purpose. Nevertheless, it is more satisfactory to have a duality function that works for all session types.

Bernardi & Hennessy [4] give an alternative, correct duality function and justify it with respect to their model of session types which is based on a theory of contracts. The key idea is that a session type can be converted into an equivalent type in which all message types are closed. In Section 3 we present their definition and a variation of it, and reformulate their correctness result in a standard model of recursive types.

Bernardi, Dardha, Gay & Kouzapas [2] discuss several definitions of duality, focusing on the fact that there can be different sound definitions which give rise to different typing relations. One of their definitions is that of Bernardi & Hennessy [4]. They point out that some results claimed by Gay & Hole [6] are false for non-tail-recursive types.

Lindley & Morris [9] give another definition of the duality function. It maps a type variable X in tail position to a *negative type variable* $\overline{X}$, but in a message position it remains as X. As well as being a technical convenience, negative variables allow interesting types such as $\mu X.!\overline{X}.X$. Lindley & Morris justify their definition on general type-theoretic grounds, but do not directly prove its correctness with respect to the duality relation. In Section 4 we do so, as well as giving an equivalent and arguably simpler variation, and another variation that turns out to be equivalent to the Bernardi-Hennessy definition.

As well as proving the results mentioned above on paper, we have begun work on mechanising them in Agda. We summarise the mechanisation in Section 5.

2 Basic Definitions about Session Types

We work formally with a subset of session types, consisting of input and output (no branch or select), and int as a representative data type. All definitions and proofs can be straightforwardly extended to cover branch and select; reasoning about equivalence and duality of session types is not affected by the details of data types.

Definition 1 (Types and Session Types) *Let X,Y,Z range over a denumerable set of* type variables*. Types (T,U) and session types (R,S) are defined by*

$$T,U ::= \mathsf{int} \mid S \qquad\qquad R,S ::= \mathsf{end} \mid ?T.S \mid !T.S \mid X \mid \mu X.S$$

Session types must be contractive*, meaning that they must not contain sub-expressions of the form $\mu X.\mu X_1.\ldots\mu X_n.X$ for $n \geq 0$. The expression $\mu X.S$ binds type variable X with scope S. The set $\mathsf{fv}(T)$ of* free type variables *in a type T is defined as usual, and so is* α-congruence*. The set of closed session types is denoted by SType and the set of closed types is denoted by Type, so that $\mathsf{Type} = \mathsf{SType} \cup \{\mathsf{int}\}$. We identify types that are α-congruent and follow the Barendregt convention on variables [1].*

In the sequel, we use term *type* for any contractive type generated by the grammar for T. When we mean a closed type, we shall speak of $T \in \mathsf{Type}$. The same reasoning applies to session types, where the term *session type* denotes a contractive type generated by the grammar for S, and $S \in \mathsf{SType}$ denotes a closed session type.

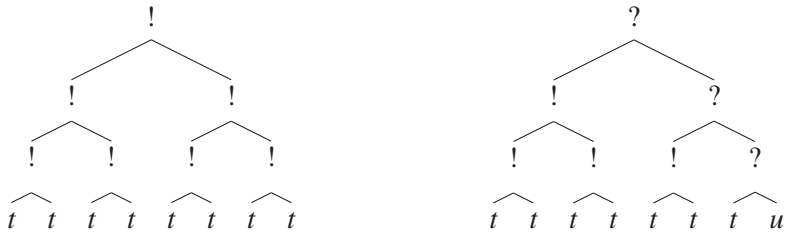
Definition 2 (Substitution) *The result of substituting type U for the free occurrences of variable X in type T—notation $T[U/X]$—is defined inductively as follows, where $Y \neq X$.*

$$\begin{array}{llll} \mathsf{int}[U/X] = \mathsf{int} & (?S.T)[U/X] = ?S[U/X].T[U/X] & X[U/X] = U & (\mu X.S)[U/X] = \mu X.S \\ \mathsf{end}[U/X] = \mathsf{end} & (!S.T)[U/X] = !S[U/X].T[U/X] & Y[U/X] = Y & (\mu Y.S)[U/X] = \mu Y.S[U/X] \end{array}$$

Closed session types are interpreted as regular trees in the standard way presented by Pierce [10, Chapter 21]. A regular tree is a (possibly infinite) tree with a finite number of distinct subtrees.

Definition 3 (Types as Trees) *Types are represented by regular trees whose nodes are taken from the set $\{\mathsf{int}, \mathsf{end}, !, ?\}$, int and end have no descendants, $!$ and $?$ have two descendants, and int can only occur as root or at the immediate left of $!$ or $?$. We write $\mathsf{treeof}(T)$ for the tree representation of T.*

Example 4 *Let S be the session type $\mu X.!X.X$. The regular tree $t = \mathsf{treeof}(S)$ can be depicted as below left. The tree u such that $t \asymp u$ can be depicted as below right.*



Equivalence of session types is equality of trees. We give a coinductive syntactic characterisation of equivalence.

Definition 5 (Syntactic Equivalence of Types) *If $\mathscr{E}$ is a relation on* Type *then $F_{\approx}(\mathscr{E})$ is the relation on* Type *defined by:*

$$\begin{aligned} F_{\approx}(\mathscr{E}) = {} & \{(\mathsf{end}, \mathsf{end})\} \\ & \cup \{(\mathsf{int}, \mathsf{int})\} \\ & \cup \{(?T_1.S_1, ?T_2.S_2) \mid (T_1, T_2), (S_1, S_2) \in \mathscr{E}\} \\ & \cup \{(!T_1.S_1, !T_2.S_2) \mid (T_1, T_2), (S_1, S_2) \in \mathscr{E}\} \\ & \cup \{(S_1, \mu X.S_2) \mid (S_1, S_2[\mu X.S_2/X]) \in \mathscr{E}\} \\ & \cup \{(\mu X.S_1, S_2) \mid (S_1[\mu X.S_1/X], S_2) \in \mathscr{E} \text{ and } S_2 \neq \mu Y.S_3\} \end{aligned}$$

A relation $\mathscr{E}$ on Type *is a* type bisimulation *if $\mathscr{E} \subseteq F_{\approx}(\mathscr{E})$. Syntactic equivalence of types, $\approx$, is the largest type bisimulation.*

Proposition 6 (Type equivalence is tree equality [10]) *Let $T, U \in$ Type. Then $T \approx U$ if and only if $\mathsf{treeof}(T) = \mathsf{treeof}(U)$.*

The duality relation is defined on regular trees.

Definition 7 (Duality on Trees) *Two trees, s and t, are related by duality—notation $s \asymp t$—if they have the same structure and, for each pair of corresponding nodes, if the nodes are* in the right spine of the tree *they are related as below, otherwise they are the same.*

$$\mathsf{int} \leftrightarrow \mathsf{int} \qquad \mathsf{end} \leftrightarrow \mathsf{end} \qquad ? \leftrightarrow ! \qquad ! \leftrightarrow ?$$

Because $\asymp$ is bijective and every tree is related to some other (unique) tree, we can also regard it as a self-inverse function, which we denote by $\mathsf{dual}(\cdot)$.

Proceeding as for type equivalence, we now give a coinductive syntactic characterisation of the duality relation, restricting attention to session types because int can only occur in message positions, where duality is never applied. This principle is applied to all of our syntactic definitions of duality.

Definition 8 (Syntactic Duality of Session Types) *If $\mathscr{D}$ is a relation on* SType *then $F_\perp(\mathscr{D})$ is the relation on* SType *defined by:*

$$\begin{aligned} F_\perp(\mathscr{D}) = {} & \{(\mathsf{end},\mathsf{end})\} \\ & \cup \{(?T_1.S_1, !T_2.S_2) \mid T_1 \approx T_2 \text{ and } (S_1,S_2) \in \mathscr{D}\} \\ & \cup \{(!T_1.S_1, ?T_2.S_2) \mid T_1 \approx T_2 \text{ and } (S_1,S_2) \in \mathscr{D}\} \\ & \cup \{(S_1, \mu X.S_2) \mid (S_1, S_2[\mu X.S_2/X]) \in \mathscr{D}\} \\ & \cup \{(\mu X.S_1, S_2) \mid (S_1[\mu X.S_1/X], S_2) \in \mathscr{D} \text{ and } S_2 \neq \mu Y.S_3\} \end{aligned}$$

A relation $\mathscr{D}$ on SType *is a* session duality *if $\mathscr{D} \subseteq F_\perp(\mathscr{D})$. Duality of session types, $\perp$, is the largest session duality.*

Proposition 9 (Type Duality Is Tree Duality) *Let $R,S \in$ SType. Then $R \perp S$ if and only if* $\mathsf{treeof}(R) \asymp \mathsf{treeof}(S)$.

Proof Similar to that of Proposition 6. □

This section introduces duality as a relation on session types. It turns out that, given a session type S, one can construct a session type S' such that $\mathsf{treeof}(S) \asymp \mathsf{treeof}(S')$, or equivalently $S \perp S'$. The next two sections show two different approaches to the problem, both starting from session types in syntactic form (Definition 1).

3 Duality à la Bernardi-Hennessy

Bernardi and Hennessy [4] observe that the problem of building a dual session type with naive duality (as explained in Section 1) is caused by free variables in message types. They give a method for constructing a dual type for an arbitrary session type S:

1. Convert S into an equivalent type S' in which every message type is closed. This step is called *message closure* (Definition 14, later).
2. Apply naive duality to S' (Definition 10, below).

In this section we present the details of this approach. First we gather the definition of naive duality from Section 1.

Definition 10 (Naive Duality Function) *The naive duality function on session types is inductively defined as follows.*

$$\overline{?T.S} = !T.\overline{S} \qquad \overline{!T.S} = ?T.\overline{S} \qquad \overline{\mathsf{end}} = \mathsf{end} \qquad \overline{X} = X \qquad \overline{\mu X.S} = \mu X.\overline{S}$$

We use the term *tail recursive* for session types in which all message types are closed. It turns out that these are *not* types with variables in tail positions only. A counterexample is $\mu X.!(?\mathsf{int}.X).\mathsf{end}$ where X occurs in tail position, but there is a message type that is not closed, namely $?\mathsf{int}.X$. To define tail recursive types we introduce a type formation system that essentially keeps track of the free variables in processes, in such a way that types such as the above are deemed ill-formed.

Definition 11 (Tail Recursive Types) *Let $\mathscr{X}$ be a set of type variables. The set of* tail recursive types over $\mathscr{X}$*, notation $\mathscr{X} \vdash T$, is defined inductively as follows.*

$$\frac{}{\mathscr{X} \vdash \mathsf{int}} \qquad \frac{}{\mathscr{X} \vdash \mathsf{end}} \qquad \frac{\emptyset \vdash S \quad \mathscr{X} \vdash T}{\mathscr{X} \vdash ?S.T} \qquad \frac{\emptyset \vdash S \quad \mathscr{X} \vdash T}{\mathscr{X} \vdash !S.T} \qquad \frac{X \in \mathscr{X}}{\mathscr{X} \vdash X} \qquad \frac{\mathscr{X}, X \vdash T}{\mathscr{X} \vdash \mu X.T}$$

The set of tail recursive types *is the set of types T such that $\emptyset \vdash T$.*

We can easily see that, if $\mathscr{X} \vdash \mu X.!S.T$, then X does not occur free in S. In particular the type $\mu X.!(?\mathsf{int}.X).\mathsf{end}$ identified above is not tail recursive.

Gay & Hole [6] claim to prove that for all $S \in \mathsf{SType}$, $\overline{S} \perp S$. They use a slightly different definition of $\perp$ in which types are completely unfolded before analysing their structure. Unfolding means repeatedly transforming top-level $\mu X.S$ to $S[\mu X.S/X]$ until a non-μ type is exposed. However, the proof contains the claim that if the unfolding of S is $?T.S'$ then the unfolding of $\overline{S}$ is $!T.\overline{S'}$, which is not true if T contains type variables. Their proof does, however, show the following result.

Proposition 12 (Soundness of Naive Duality for Tail Recursive Types [6]) *If S is a tail recursive type, then $S \perp \overline{S}$.*

This supports the Bernardi-Hennessy approach, because it shows that if a session type can be converted to an equivalent type that is tail recursive, then it is sufficient to apply naive duality to the tail recursive type.

Message closure builds a tail recursive session type by collecting substitutions $[\mu X.S/X]$ for each $\mu X.S$ type encountered and applying the accumulated substitution to messages. To define message closure we need the notion of a sequence of substitutions.

Definition 13 (Sequence of Substitutions) *A* sequence of substitutions *is given by the following grammar:*

$$\sigma ::= \varepsilon \mid [S/X];\sigma$$

The application of a sequence of substitutions σ *to a type T—notation $T\sigma$—is defined as $T\varepsilon = T$ and $T([S/X];\sigma) = (T[S/X])\sigma$. A sequence of substitutions σ is* closing *for T if* $\mathsf{fv}(T\sigma) = \varepsilon$.

Definition 14 (Message Closure [4]) *For any type T and sequence of substitutions σ closing for T, the type* $\mathsf{mclo}(T,\sigma)$ *is defined inductively by the following rules.*

$$\begin{aligned} \mathsf{mclo}(\mathsf{end},\sigma) &= \mathsf{end} & \mathsf{mclo}(X,\sigma) &= X \\ \mathsf{mclo}(!T.S,\sigma) &= !(T\sigma).\mathsf{mclo}(S,\sigma) & \mathsf{mclo}(\mu X.S,\sigma) &= \mu X.\mathsf{mclo}(S,[(\mu X.S)/X];\sigma) \\ \mathsf{mclo}(?T.S,\sigma) &= ?(T\sigma).\mathsf{mclo}(S,\sigma) \end{aligned}$$

Define $\mathsf{mclo}(S)$ *as* $\mathsf{mclo}(S,\varepsilon)$.

Bernardi and Hennessy prove that taking the naive dual of the message closure of a type is sound with respect to a notion of compatibility based on a labelled transition system for session types. We will prove soundness with respect to regular trees. First, however, we show that if $S \in \mathsf{SType}$ then $\mathsf{mclo}(S)$ is tail recursive.

The next two lemmas are easily proved by induction.

Lemma 15 *If $\mathscr{X} \vdash T$, then $\mathsf{fv}(T) \subseteq \mathscr{X}$.*

Lemma 16 (Strengthening) *If $\mathscr{X}, X \vdash T$ and $X \notin \mathsf{fv}(T)$, then $\mathscr{X} \vdash T$.*

Combining them, we can identify exactly the type variables that occur free in message positions.

Corollary 17 *If $\mathscr{X} \vdash T$, then $\mathsf{fv}(T) \vdash T$.*

Proof From Lemma 15 we know that $\mathsf{fv}(T) \subseteq \mathscr{X}$. Use Strengthening (Lemma 16) repeatedly to remove from $\mathscr{X}$ type variables not in $\mathsf{fv}(T)$. □

Finally, we reason about $\mathsf{mclo}(T,\sigma)$.

Lemma 18 *If σ is a closing substitution for T, then $\mathsf{dom}(\sigma) \vdash \mathsf{mclo}(T,\sigma)$.*

Proof Straightforward induction on the definition of $\mathsf{mclo}(T,\sigma)$. □

Corollary 19 *If T is closed, then $\mathsf{mclo}(T)$ is tail recursive.*

Proof If T is closed, then ε is a closing substitution for T. Lemma 18 ensures that $\emptyset \vdash \mathsf{mclo}(T,\varepsilon)$, hence $\emptyset \vdash \mathsf{mclo}(T)$ by definition. □

Example 20 *The Bernardi-Hennessy approach to duality applied to our running example $S = \mu X.!X.X$.*

$$\begin{aligned}\overline{\mathsf{mclo}(S)} = \overline{\mathsf{mclo}(S,\varepsilon)} &= \overline{\mu X.\mathsf{mclo}(!X.X,[S/X])} \\ &= \overline{\mu X.(!X[S/X]).\mathsf{mclo}(X,[S/X])} \\ &= \overline{\mu X.!S.X} = \mu X.\overline{!S.X} = \mu X.?S.\overline{X} = \mu X.?S.X\end{aligned}$$

It turns out that the two steps—application of message closure and the computation of naive duality—can be combined into a single step, performing message closure during the process of computing the dual type. This is captured by the definition below, which constructs the dual of a type in a single pass over its abstract syntax tree.

Definition 21 (Duality with On-the-fly Message Closure) *For any session type S and sequence of substitutions σ closing for S, the session type $\mathsf{dual}_{\mathsf{BH}}(S,\sigma)$ is defined inductively by the following rules.*

$$\begin{aligned}&\mathsf{dual}_{\mathsf{BH}}(\mathsf{end},\sigma) = \mathsf{end} && \mathsf{dual}_{\mathsf{BH}}(X,\sigma) = X \\ &\mathsf{dual}_{\mathsf{BH}}(!T.S,\sigma) = ?(T\sigma).\mathsf{dual}_{\mathsf{BH}}(S,\sigma) && \mathsf{dual}_{\mathsf{BH}}(\mu X.S,\sigma) = \mu X.\mathsf{dual}_{\mathsf{BH}}(S,[(\mu X.S)/X];\sigma) \\ &\mathsf{dual}_{\mathsf{BH}}(?T.S,\sigma) = !(T\sigma).\mathsf{dual}_{\mathsf{BH}}(S,\sigma)\end{aligned}$$

Define $\mathsf{dual}_{\mathsf{BH}}(S)$ as $\mathsf{dual}_{\mathsf{BH}}(S,\varepsilon)$.

Example 22 *Here is duality with on-the-fly message closure in action for our running example $S = \mu X.!X.X$.*

$$\begin{aligned}\mathsf{dual}_{\mathsf{BH}}(S) = \mathsf{dual}_{\mathsf{BH}}(S,\varepsilon) &= \mu X.\mathsf{dual}_{\mathsf{BH}}(!X.X,[S/X]) \\ &= \mu X.(?X[S/X]).\mathsf{dual}_{\mathsf{BH}}(X,[S/X]) \\ &= \mu X.?S.\mathsf{dual}_{\mathsf{BH}}(X,[S/X]) \\ &= \mu X.?S.X\end{aligned}$$

The economy with respect to the original definition, Example 20, should be apparent.

Example 23 *Consider the problematic type of Bernardi and Hennessy [3]. Let $S_2 = \mu Y.!Y.X$ and $S_1 = \mu X.S_2$. We then have:*

$$\begin{aligned}\mathsf{dual}_{\mathsf{BH}}(S_1) = \mathsf{dual}_{\mathsf{BH}}(S_1,\varepsilon) &= \mu X.\mathsf{dual}_{\mathsf{BH}}(S_2,[S_1/X]) \\ &= \mu X.\mu Y.\mathsf{dual}_{\mathsf{BH}}(!Y.X,[S_1/X][S_2/Y]) \\ &= \mu X.\mu Y.?(Y[S_1/X][S_2/Y]).\mathsf{dual}_{\mathsf{BH}}(X,[S_1/X][S_2/Y]) \\ &= \mu X.\mu Y.?S_2.\mathsf{dual}_{\mathsf{BH}}(X,[S_1/X][S_2/Y]) \\ &= \mu X.\mu Y.?S_2.X\end{aligned}$$

Applying message closure on the fly does not change the tail recursive type that we obtain.

Proposition 24 *If $S \in \mathsf{SType}$ then $\mathsf{dual}_{\mathsf{BH}}(S)$ is syntactically equal to $\overline{\mathsf{mclo}(S)}$.*

Proof Prove by structural induction on S that for any sequence of substitutions σ closing for S, $\mathsf{dual}_{\mathsf{BH}}(S,\sigma)$ is syntactically equal to $\mathsf{mclo}(S,\sigma)$. □

We can now show that duality with on-the-fly message closure is sound with respect to duality on regular trees.

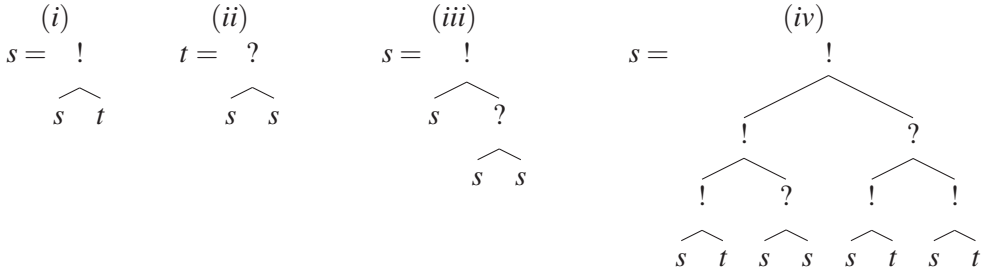
Proposition 25 *If* $S \in \mathsf{SType}$ *then* $\mathsf{treeof}(\overline{\mathsf{mclo}(S)}) \asymp \mathsf{treeof}(S)$.

Proof Instead of proving this directly, we go via definitions and results from Section 4. Proposition 24 shows that $\overline{\mathsf{mclo}(S)} = \mathsf{dual}_{\mathsf{BH}}(S)$. Proposition 36 shows that $\mathsf{dual}_{\mathsf{BH}}(S) = \mathsf{dual}_{\mathsf{LMN}}(S)$, where $\mathsf{dual}_{\mathsf{LMN}}(S)$ is defined in Definition 34. Therefore $\mathsf{treeof}(\overline{\mathsf{mclo}(S)}) = \mathsf{treeof}(\mathsf{dual}_{\mathsf{LMN}}(S))$. Finally, Proposition 37 shows that $\mathsf{treeof}(\mathsf{dual}_{\mathsf{LMN}}(S)) \asymp \mathsf{treeof}(S)$. □

4 Duality à la Lindley-Morris

The Lindley-Morris definition of the duality function [9] uses negative type variables $\overline{X}$, which we therefore add to the syntax in Definition 1. In type $\mu X.T$, both the positive variable X and the negative variable $\overline{X}$ are bound in T. Corresponding to this extension, we generalise the definition of $\mathsf{treeof}(\cdot)$ (Definition 3) so that $\mathsf{dual}(\cdot)$ (Definition 7) is applied to the subtrees that arise from negative variables.

Example 26 *Let* S *be the type* $\mu X.!X.\overline{X}$. *Let* $s = \mathsf{treeof}(S)$ *and let* $t = \mathsf{dual}(s)$. *Tree* s *can be depicted as (i) below. To obtain tree* t*, (ii) below, we dualise the root, keep* s *for the left subtree and use the dual of* t *(that is,* s*) for the right subtree (cf. the rule* $\overline{!T.S} = ?T.\overline{S}$ *in Definition 10). Substituting* t *into tree (i) gives tree (iii). Tree (iv) shows a few more nodes in the expansion of* s.



The definition of the duality function also requires a particular form of substitution that exchanges negative variables $\overline{X}$ and positive variables X.

Definition 27 (Negative Variable Substitution) *The result of substituting* $\overline{X}$ *for the free occurrences of* X *in* T*—notation* $T\{\overline{X}/X\}$*—is defined inductively as follows.*

$$
\begin{array}{ll}
X\{\overline{X}/X\} = \overline{X} & \mathsf{int}\{\overline{X}/X\} = \mathsf{int} \\
\overline{X}\{\overline{X}/X\} = X & \mathsf{end}\{\overline{X}/X\} = \mathsf{end} \\
Y\{\overline{X}/X\} = Y \ \ \textit{if } Y \neq X & (?S.T)\{\overline{X}/X\} = ?(S\{\overline{X}/X\}).T\{\overline{X}/X\} \\
\overline{Y}\{\overline{X}/X\} = \overline{Y} & (!S.T)\{\overline{X}/X\} = !(S\{\overline{X}/X\}).T\{\overline{X}/X\} \\
 & (\mu Y.S)\{\overline{X}/X\} = \mu Y.S\{\overline{X}/X\}
\end{array}
$$

Definition 28 (Lindley-Morris Duality, Original Version [9])

$$
\begin{array}{ll}
\mathsf{dual}_{\mathsf{LM}}(\mathsf{end}) = \mathsf{end} & \mathsf{dual}_{\mathsf{LM}}(X) = \overline{X} \\
\mathsf{dual}_{\mathsf{LM}}(?T.S) = !T.\mathsf{dual}_{\mathsf{LM}}(S) & \mathsf{dual}_{\mathsf{LM}}(\overline{X}) = X \\
\mathsf{dual}_{\mathsf{LM}}(!T.S) = ?T.\mathsf{dual}_{\mathsf{LM}}(S) & \mathsf{dual}_{\mathsf{LM}}(\mu X.S) = \mu X.(\mathsf{dual}_{\mathsf{LM}}(S)\{\overline{X}/X\})
\end{array}
$$

Example 29

$$\begin{aligned}\mathsf{dual}_{\mathsf{LM}}(\mu X.!X.X) &= \mu X.\mathsf{dual}_{\mathsf{LM}}((!X.X)[\overline{X}/X]) = \mu X.(?X.\mathsf{dual}_{\mathsf{LM}}(X))[\overline{X}/X]\\ &= \mu X.(?X.\overline{X})[\overline{X}/X] = \mu X.?X[\overline{X}/X].\overline{X}[\overline{X}/X]\\ &= \mu X.?\overline{X}.\overline{X}[\overline{X}/X] = \mu X.?\overline{X}.X\end{aligned}$$

This definition of duality is sound with respect to trees.

Proposition 30 *If* $S \in \mathsf{SType}$ *then* $\mathsf{dual}_{\mathsf{LM}}(S) \asymp S$.

Proof This is one of the results that we have mechanized in Agda (Section 5). □

There is an alternative formulation of Lindley-Morris duality that works with conventional substitution (Definition 2 with the additional clause $\overline{Y}[S/Z] = \overline{Y}$, i.e., no substitution for negative variables). The idea is that the (bound) occurrences of X in the dual of $\mu X.S$ are occurrences not of X (which stands for S) but of $\overline{X}$ (which stands for the dual of S). So we first substitute $\overline{X}$ for X in S and only then apply the duality function.

Definition 31 (Lindley-Morris Duality, Polished)

$$\begin{aligned}\mathsf{dual}_{\mathsf{LMP}}(\mathsf{end}) &= \mathsf{end} & \mathsf{dual}_{\mathsf{LMP}}(X) &= \overline{X}\\ \mathsf{dual}_{\mathsf{LMP}}(?T.S) &= !T.\mathsf{dual}_{\mathsf{LMP}}(S) & \mathsf{dual}_{\mathsf{LMP}}(\overline{X}) &= X\\ \mathsf{dual}_{\mathsf{LMP}}(!T.S) &= ?T.\mathsf{dual}_{\mathsf{LMP}}(S) & \mathsf{dual}_{\mathsf{LMP}}(\mu X.S) &= \mu X.\mathsf{dual}_{\mathsf{LMP}}(S[\overline{X}/X])\end{aligned}$$

Example 32

$$\begin{aligned}\mathsf{dual}_{\mathsf{LMP}}(\mu X.!X.X) &= \mu X.\mathsf{dual}_{\mathsf{LMP}}((!X.X)[\overline{X}/X])\\ &= \mu X.\mathsf{dual}_{\mathsf{LMP}}(!\overline{X}.\overline{X}) = \mu X.?\overline{X}.\mathsf{dual}_{\mathsf{LMP}}(\overline{X}) = \mu X.?\overline{X}.X\end{aligned}$$

Proposition 33 *For any session type* S, $\mathsf{dual}_{\mathsf{LM}}(S)$ *is syntactically equal to* $\mathsf{dual}_{\mathsf{LMP}}(S)$.

Proof By structural induction on S, using a lemma that $\mathsf{dual}_{\mathsf{LMP}}$ commutes with substitution. □

If we are constructing the dual of a session type that contains no negative variables, we might want to avoid introducing negative variables when dualising a recursive type $\mu X.S$. We can achieve this by using Definition 31 and, at the end, replacing all occurrences of $\overline{X}$ (there are no bound occurrences of $\overline{X}$) by the original type $\mu X.S$.

Definition 34 (Lindley-Morris Duality, Yielding No New Negative Variables)

$$\begin{aligned}\mathsf{dual}_{\mathsf{LMN}}(\mathsf{end}) &= \mathsf{end} & \mathsf{dual}_{\mathsf{LMN}}(X) &= \overline{X}\\ \mathsf{dual}_{\mathsf{LMN}}((?T.S)) &= !T.\mathsf{dual}_{\mathsf{LMN}}(S) & \mathsf{dual}_{\mathsf{LMN}}(\overline{X}) &= X\\ \mathsf{dual}_{\mathsf{LMN}}((!T.S)) &= ?T.\mathsf{dual}_{\mathsf{LMN}}(S) & \mathsf{dual}_{\mathsf{LMN}}((\mu X.S)) &= \mu X.((\mathsf{dual}_{\mathsf{LMN}}(S[\overline{X}/X]))[\mu X.S/\overline{X}])\end{aligned}$$

Example 35

$$\begin{aligned}\mathsf{dual}_{\mathsf{LMN}}(S) = \mathsf{dual}_{\mathsf{LMN}}((\mu X.!X.X)) &= \mu X.\mathsf{dual}_{\mathsf{LMN}}(((!X.X)[\overline{X}/X]))[S/\overline{X}]\\ &= \mu X.\mathsf{dual}_{\mathsf{LMN}}((!\overline{X}.\overline{X}))[S/\overline{X}]\\ &= \mu X.(?\overline{X}.\mathsf{dual}_{\mathsf{LMN}}(\overline{X}))[S/\overline{X}]\\ &= \mu X.(?\overline{X}.X)[S/\overline{X}] = \mu X.?S.X\end{aligned}$$

This version of the Lindley-Morris definition coincides with the Bernardi-Hennessy definition.

Proposition 36 *For any session type S,* $\mathsf{dual}_{\mathsf{BH}}(S)$ *is syntactically equal to* $\mathsf{dual}_{\mathsf{LMN}}(S)$.

Proof If σ is a sequence of substitutions $[T_1/X_1]\dots[T_n/X_n]$ then let $\overline{\sigma} = [T_1/\overline{X_1}]\dots[T_n/\overline{X_n}]$ and $\hat{\sigma} = [\overline{X_1}/X_1]\dots[\overline{X_n}/X_n]$. Prove by structural induction on S that for any sequence of substitutions σ closing for S, $\mathsf{dual}_{\mathsf{BH}}(S,\sigma) = \mathsf{dual}_{\mathsf{LMN}}((S\hat{\sigma}))\overline{\sigma}$. The result follows by taking $\sigma = \varepsilon$. □

Finally, the Lindley-Morris definition is sound with respect to regular trees.

Proposition 37 *If* $S \in \mathsf{SType}$ *then* $\mathsf{treeof}(\mathsf{dual}_{\mathsf{LMN}}(S)) \asymp \mathsf{treeof}(S)$.

Proof First show that $\mathscr{D} = \{(S, \mathsf{dual}_{\mathsf{LMN}}(S)) \mid S \in \mathsf{SType}\}$ is a session duality (Definition 8). This establishes $\mathsf{dual}_{\mathsf{LMN}}(S) \perp S$. Then use Proposition 9. □

The substitutions in Definition 34, or equivalently in the definition of message closure (Definition 21) increase the size of the type. A simple example shows that this increase can be at least quadratic. If $S = \mu X.?X.\cdots?X.X$ with n inputs, so that the size of S is $n+2$, then $\mathsf{mclo}(S) = \mu X.?S.\cdots?S.X$ of size $n(n+2)+2$. In contrast, Definitions 28 and 31 preserve the size of the type because they only substitute variables for variables. In an implementation of a programming language with session types, it is possible to avoid computational issues resulting from these syntactic size increases, by working with a graph representation of regular trees.

5 Mechanized Results

We mechanized some of the results of the paper in Agda and are working towards a full mechanized account of all results. For accessibility, we paraphrase the definitions in standard mathematical notation rather than Agda syntax. Cognoscenti may explore the Agda source code corresponding to the development in this section in file `Duality.agda` at `https://github.com/peterthiemann/dual-session`.

The baseline for the mechanization is the coinductive formalization of session types (Definition 38), which we consider as the ground truth. In this setting, a session type is a potentially infinite tree as contained in the greatest fixpoint SType^∞ of function S_{gen}.

Definition 38 (Coinductive Session Types)

$$S_{\mathsf{gen}}(\mathscr{S}) = \{\mathsf{end}\} \cup \{!T.S, ?T.S \mid S \in \mathscr{S}, T \in \{\mathsf{int}\} \cup \mathscr{S}\}$$

Defining duality for coinductive session types is a straightforward corecursively defined function which we call $\mathsf{dual}(\cdot)$, reusing the name from Definition 7 because it implements that function.

Definition 39 (Corecursive Duality Function)

$$\mathsf{dual}(\mathsf{end}) = \mathsf{end} \qquad \mathsf{dual}(!T.S) = ?T.\mathsf{dual}(S) \qquad \mathsf{dual}(?T.S) = !T.\mathsf{dual}(S)$$

It is also straightforward to define the duality relation (cf. Def. 8) as the greatest fixpoint $(\perp)$ of $F_\perp(\cdot)$.

Definition 40 (Duality on Coinductive Session Types) *If* $\mathscr{D}$ *is a binary relation on tree types, then*

$$F_\perp(\mathscr{D}) = \{(\mathsf{end},\mathsf{end})\} \cup \{(!T.S, ?T.S^\perp), (?T.S, !T.S^\perp) \mid (S, S^\perp) \in \mathscr{D}\}$$

Given these definitions, it is easy to show that the corecursive duality function is sound and complete with respect to the duality relation (cf. Proposition 9).

Proposition 41 $S \perp S'$ *if and only if* $S' = \mathsf{dual}(S)$.

To formalize session types inductively, we insist that μ-types are in normal form where there are no consecutive μ-abstractions, i.e., no subterms of the form $\mu X.\mu Y.S$, and the body of a μ is never a variable. Normal forms are contractive by construction and every contractive session type (according to Definition 1) can be converted to its equivalent normal form by repeatedly coalescing subterms of the form $\mu X.\mu Y.S$ to $\mu X.S[X/Y]$ and transforming subterms of the form $\mu X.Y$ to Y, assuming $X \neq Y$. The Agda formalization enforces normal forms using two mutually recursive syntactic categories, S and S', for session types:

$$S ::= S' \mid \mu X.S' \mid X \mid \overline{X} \qquad S' ::= \mathsf{end} \mid !T.S \mid ?T.S \qquad T ::= \mathsf{int} \mid S$$

For this representation, we state various definitions of duality as shown in Sections 3 and 4. Next, we define an embedding $\lfloor\cdot\rfloor$ from SType to tree types by unfolding the recursion. This function corresponds to the $\mathsf{treeof}(\cdot)$ function (Definition 3).

$$\lfloor \mu X.S' \rfloor = \lfloor S'[\mu X.S'/X] \rfloor' \quad \lfloor \mathsf{end} \rfloor' = \mathsf{end} \quad \lfloor !T.S \rfloor' = !\lfloor T \rfloor.\lfloor S \rfloor \quad \lfloor ?T.S \rfloor' = ?\lfloor T \rfloor.\lfloor S \rfloor \quad \lfloor \mathsf{int} \rfloor = \mathsf{int}$$

This definition is mutually recursive ($\lfloor\cdot\rfloor$ applies to S and $\lfloor\cdot\rfloor'$ applies to S') and it is guarded (i.e., it yields a proper, potentially infinite term) because $\lfloor\cdot\rfloor'$ always yields a top-level constructor.

We successfully mechanised a range of results from this paper among them Proposition 30, restated here with the embedding function.

Proposition 42 *For all $S \in \mathsf{SType}$, $\mathsf{dual}(\lfloor S \rfloor) = \lfloor \mathsf{dual}_{\mathsf{LM}}(S) \rfloor$.*

6 Conclusion

We surveyed the competing definitions of session type duality in the presence of recursion. Starting from an interpretation of session types as trees, and a duality relation on trees, we establish soundness of the Bernardi-Hennessy and the Lindley-Morris definitions of duality on syntactic session types. We further come up with streamlined versions of these definitions and justify the original flawed definition of duality (naive duality) when restricted to tail recursive session types. We have mechanized some results in Agda, and are working on mechanizing the others.

In summary, we have tied up the remaining loose ends in the definition of duality of session types. Many of the issues in prior work are caused by syntax, namely by reliance on μ-types to express recursion. Taking a standard interpretation of recursive types as regular trees, and the corresponding formalization by coinductive definitions in Agda, is effective in proving the soundness of syntactic definitions.

Acknowledgements. Simon Gay was partially supported by the UK EPSRC grant EP/K034413/1 "From Data Types to Session Types: A Basis for Concurrency and Distribution" and by the EU Horizon 2020 MSCA-RISE project 778233 "BehAPI: Behavioural Application Program Interfaces". Vasco T. Vasconcelos was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020, and by COST Action CA15123 EUTypes. We thank Sam Lindley and Garrett Morris for discussions.

References

[1] Hendrik Pieter Barendregt (1985): *The Lambda Calculus — its Syntax and Semantics*. *Studies in Logic and the Foundations of Mathematics* 103, North-Holland.

[2] Giovanni Bernardi, Ornela Dardha, Simon J. Gay & Dimitrios Kouzapas (2014): *On Duality Relations for Session Types*. In: *TGC*, doi:10.1007/978-3-662-45917-1_4.

[3] Giovanni Bernardi & Matthew Hennessy (2014): *Using Higher-Order Contracts to Model Session Types (Extended Abstract)*. In: *CONCUR*, doi:10.1007/978-3-662-44584-6_27.

[4] Giovanni Bernardi & Matthew Hennessy (2016): *Using higher-order contracts to model session types*. *Logical Methods in Computer Science* 12(2), doi:10.2168/LMCS-12(2:10)2016.

[5] Simon J. Gay & Malcolm Hole (1999): *Types and Subtypes for Client-Server Interactions*. In: *ESOP*, doi:10.1007/3-540-49099-X_6.

[6] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Informatica* 42(2-3), doi:10.1007/s00236-005-0177-z.

[7] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR*, doi:10.1007/3-540-57208-2_35.

[8] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP*, doi:10.1007/BFb0053567.

[9] Sam Lindley & J. Garrett Morris (2016): *Talking bananas: structural recursion for session types*. In: *ICFP*, doi:10.1145/2951913.2951921.

[10] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[11] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE*, doi:10.1007/3-540-58184-7_118.

[12] Vasco T. Vasconcelos (2012): *Fundamentals of session types*. *Information and Computation* 217, pp. 52–70, doi:10.1016/j.ic.2012.05.002.

Bounded verification of message-passing concurrency in Go using Promela and Spin

Nicolas Dilley
University of Kent

Julien Lange
University of Kent

This paper describes a static verification framework for the message-passing fragment of the Go programming language. Our framework extracts models that over-approximate the message-passing behaviour of a program. These models, or behavioural types, are encoded in Promela, hence can be efficiently verified with Spin. We improve on previous works by verifying programs that include communication-related parameters that are unknown at compile-time, i.e., programs that spawn a parameterised number of threads or that create channels with a parameterised capacity. These programs are checked via a bounded verification approach with bounds provided by the user.

1 Introduction

Go is an increasingly popular programming language that is known for its lightweight threads (called *goroutines*) and native support for message-passing concurrency. Go programmers are encouraged to coordinate threads by exchanging messages over channels, rather than using shared memory protected by mutexes [13]. In a recent empirical survey [3], we have discovered that more than 70% of the most popular Go projects on GitHub use message-passing primitives. Additionally, Tu et al. [18] showed that message-passing based software is as liable to errors as other concurrent programming techniques. They also showed that Go concurrency bugs are hard to detect and have a long life time. This is reflected in a recent survey amongst Go programmers reporting that programmers often do not feel they are able to effectively repair bugs related to Go's concurrency features [17]. Concretely, message-passing concurrency bugs in Go fall in two categories: (*i*) blocking errors, where a goroutine is permanently waiting for a matching send/receive action and (*ii*) channel errors, where a goroutine attempts to close or send to a channel that is already closed.

The Go ecosystem provides little support for users to detect concurrency bugs. Its type system only ensures that each channel instance carries a single specified data type. While a *run-time* global deadlock detector is available, it is silently disabled by some libraries. To help programmers produce correct concurrent software, several authors have proposed techniques to verify Go programs both statically (at compile-time) [7, 8, 10, 12, 14] and dynamically (at run-time) [15, 16]. One of the more mature techniques for statically verifying Go programs is Godel [8] which relies on the similarity of Go's message-passing aspect to CCS [11]. Godel follows an approach based on behavioural types where Go programs are over-approximated by CCS-like processes, which in turns are model-checked, using mCRL2 [2] for safety and liveness properties. Because mCRL2 only deals with finite-state models, Godel has one key limitation: it does not support programs that spawn new threads in for-loops, e.g., the program in Figure 1 is not supported. This restriction limits the applicability of Godel to real-world code-bases. Indeed, 58% of the Go projects we studied in [3] feature thread-spawning in for-loops.

Figure 1 shows a typical Go program where several worker threads are concurrently sending data to the parent thread via channel `a`. Note that this program spawns $|\texttt{files}|$ threads and creates a channel

S. Balzer, L. Padovani (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 34–45, doi:10.4204/EPTCS.314.4

```
1  func main() {
2    files := getFiles()                // decl. of getFiles() omitted
3    a := make(chan string, len(files)) // create bounded buffer 'a'
4    for i := 0; i < len(files); i++ {
5      go worker(a, files[i], i) //spawn worker()
6    }
7    for i := 0; i < len(files); i++ {
8      <-a // receive from 'a'
9    }
10 }
11 func worker(a chan int, f string, i int) {
12   a <- parseFile(f, i) // send data on 'a' (decl. of parseFile() omitted)
13 }
```

Figure 1: File processing example

whose capacity is |`files`|. The length of `files` is unknown at compile-time, hence this program cannot be checked for concurrency errors with existing static verification techniques for Go [7, 8, 10, 12, 14].

Our approach Our short-term objective is to improve the approach from [8] so that we can detect bugs in programs that feature communication-related parameters that are unknown at compile-time. We focus on two kinds of communication-related parameters: (*i*) those that determine the number of threads a program may spawn at run-time and (*ii*) those that determine the capacity of channels. For example, the number of threads and the capacity of channel `a` are unknown at compile-time in Figure 1. To fulfil our objective, we augment the behavioural types technique of [8] with (*i*) an *intra*-procedural analysis to identify unknown communication-related parameters, and (*ii*) a bounded verification wrt. these parameters. Concretely, we infer behavioural types from Go programs which may feature (undefined) communication-related parameters. If so, we ask the users to instantiate these parameters with bounds so that we can model-check the inferred behavioural types. The main challenges are to ask for user-provided bounds only when necessary and to ensure that these bounds are used consistently. We address these challenges by keeping track of variables that may be used in channel creation statements or for-loops that spawn threads.

Our long-term objective is to study automated *repair* of message-passing errors in Go. To anticipate for this next step, we deviate from [8] in several ways. (1) We infer behavioural types directly from Go source code instead of its lower-level (SSA) representation. (2) We use Promela and Spin instead of mCRL2 to encode and verify behavioural types. Promela has the advantage of being much closer to Go. It has an imperative Go-like syntax and natively supports synchronous and asynchronous channels. As a consequence, it will be easier to syntactically map an error in a Promela model to its source program. (3) We divide Go programs into independent partitions. This allows us to detect partial deadlocks and to identify the location of defects more precisely, while making our tool faster. Figure 2 gives an overview of our approach, which we have implemented in a tool called GOMELA [4].

Synopsis In § 2, we present a core subset of Go, called *MiniGo*, as well as typical bugs that we want to rule out. In § 3, we give a detailed algorithm to extract Promela models from Go programs, while keeping track of communication-related parameters. In § 4 we present our implementation and its empirical evaluation. We discuss related work and conclude in § 5.

2 *MiniGo* and message-passing concurrency errors

For the sake of presentation, we use a fragment of Go that is focused on its message-passing features and call it *MiniGo* — we describe how our tool deals with a larger subset of Go in Section 4. The syntax

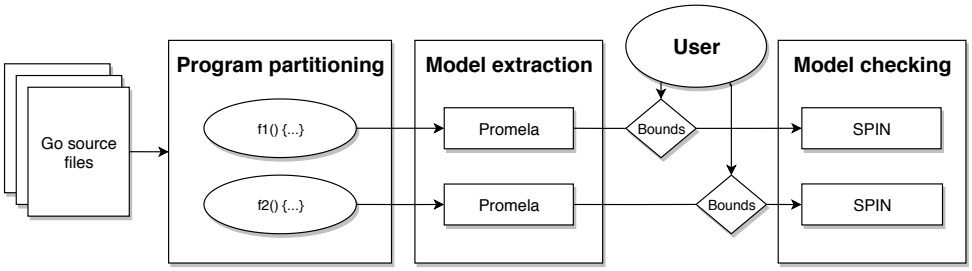


Figure 2: GOMELA workflow.

of *MiniGo* is given in Figure 3. We only discuss its semantics informally and refer to [7] for a formal account of the semantics of a variation of our language.

We use v to range over *non-channel* variables, ch to range over *channel* variables, x to range over any variables, e to range over expressions (excluding channel variables), a to range over expressions (possibly including channel variables), id to range over function names, and n to range over integer literals. We use r to range over mutators of for-loop indices. We use $\widetilde{a}$ to range over a list of expressions and overload the notation for lists of statements ($\tilde{s}$), etc. We write $\tilde{s}_1\tilde{s}_2$ for the concatenation of $\tilde{s}_1$ and $\tilde{s}_2$. We write $\texttt{chans}(\widetilde{a})$ (resp. $\texttt{chans}(\tilde{x})$) for the maximal sub-list of $\widetilde{a}$ (resp. $\tilde{x}$) that contains only channel variables.

A *MiniGo* program p consists of a list of function declarations $\tilde{d}$, possibly including a `main` function (the program's entry point). Each function declaration specifies a list of parameters $\tilde{x}$ (possibly including channel variables) and a function body $\tilde{s}$.

Statement ch `:= make(chan,`e`)` creates a new channel of capacity n, when e evaluates to n. If $n = 0$ then the channel is synchronous, otherwise it is asynchronous. Communication statements α interact with channels: $v \leftarrow ch$ receives a value from channel ch and binds it to variable v; while $ch \leftarrow e$ sends the evaluation of expression e on channel ch. Send actions are blocking when the channel is synchronous or has reached its maximal capacity. A channel can be closed with a $\texttt{close}(ch)$ operation. Any send or close action on a closed channel triggers a run-time error. Any receive action on a closed channel succeeds, if the channel is empty a default value is returned. Select statements $\texttt{select}\{\widetilde{c}\}$ are guarded choices: they block until one of the guarding communication operations succeeds; after which the corresponding case is executed. If multiple operations are available, one is chosen non-deterministically. Select statements may include a unique `default` branch, which is taken if all other branches are blocking.

A statement `go` $id(\widetilde{a})$ spawns a new goroutine, i.e., an instance of function $id(\widetilde{a})$ which is executed concurrently with its parent thread. *MiniGo* also includes standard constructs such as general sequencing, conditionals, for-loops, and assignment. For the sake of simplicity we model only the relevant parts of the language of expression (see definition of e). We assume that variable names are pairwise distinct. Additionally, as in [8], we assume that channel are not in e, that variables are immutable, and that recursive functions do not spawn goroutines (for-loops are more common than recursion in Go).

Message-passing errors in *MiniGo* In this work, we are interested in message-passing related bugs that *MiniGo* programs may encounter at run-time. We distinguish three types of such bugs. A **global deadlock** is a situation where at least one goroutine is waiting for a send or receive action to succeed, while *all* the other goroutines are either blocked or terminated. A **partial deadlock** is a situation where at least one goroutine is permanently stuck while waiting for a send or receive action to succeed. Go developers refer to partial deadlocks as *goroutine leaks* because such stuck goroutines never reach the end of their scope, and thus are never garbage-collected. A **channel safety error** is a situation where a send or close operation is triggered on a closed channel.

$$
\begin{array}{lcl}
p & := & \widetilde{d} \\
s & := & ch \texttt{ := make(chan,}e\texttt{)} \\
 & | & \alpha \mid \texttt{select}\{\widetilde{c}\} \mid \texttt{close}(ch) \\
 & | & id(\widetilde{a}) \mid \texttt{go } id(\widetilde{a}) \mid \{\tilde{s}\} \\
 & | & \texttt{if } e \texttt{ then } \tilde{s}_1 \texttt{ else } \tilde{s}_2 \mid \texttt{for } v\texttt{:=}e_1\texttt{;}e_2\texttt{;}r\{\tilde{s}_3\} \\
\alpha & := & v \leftarrow ch \mid ch \leftarrow e
\end{array}
\qquad
\begin{array}{lcl}
c & := & \texttt{case } \alpha\texttt{:}\tilde{s} \mid \texttt{default:}\tilde{s} \\
e & := & \texttt{true} \mid \texttt{false} \mid n \mid v \mid \ldots \\
a & := & ch \mid e \\
x & := & ch \mid v \\
d & := & \texttt{func } id(\tilde{x})\{\tilde{s}\} \\
r & := & v\texttt{++} \mid v\texttt{--} \mid \ldots
\end{array}
$$

Figure 3: Syntax of *MiniGo* (*ch* ranges over channel variables and *v* stands for non-channel variables).

3 Extracting Promela models from *MiniGo* programs

We adopt an approach based on behavioural types to produce a sound analysis of *MiniGo* for channel safety and global deadlock errors, following [7, 8]. Note that this approach is generally unsound wrt. liveness properties such as partial deadlock freedom without a termination checker, see [7, Section 5]. In this context, behavioural types are an over-approximation of the interactions between goroutines, i.e., they record send and receive actions, while abstracting away from the computational aspects. Typically, conditional statements are assigned behavioural types that correspond to non-deterministic choices in process calculi. In our work, behavioural types take the form of Promela models, which we extract from *MiniGo* source code. Remarkably, we keep track of some computational aspects when they affect the structure of the communication of the program, e.g., in Figure 1 we need to keep track of `len(files)`.

Given a *MiniGo* program p, we extract Promela models as follows. For each function declaration $\texttt{func } id(\tilde{x})\{\tilde{s}\}$ where $\tilde{x}$ does not contain any channel variables, we generate a model which consists of three parts: (1) a model entry point (`init` process in Promela) that contains the translation of $\tilde{s}$; (2) a list of process declarations (`proctype` in Promela), one for each distinct function call occurring (inter-procedurally) in $\tilde{s}$; (3) a set of monitor processes, one for each channel created in $\tilde{s}$. Each of these models correspond to a partition of a *MiniGo* program. Because these partitions do not have free channel variables, they are effectively independent. Hence, we can verify them independently by considering each function declaration without channel parameters as a program entry point. As a consequence, we obtain a more precise and wider analysis of code-bases, while reducing the computational cost of our analysis, comparing to [8]. In particular, we can detect some *partial* deadlocks in the program under considering by identifying global deadlocks in some of its partitions.

Hereafter, we take the following conventions: Promela strings generated by our algorithm are written in `typewriter blue`. *MiniGo* code is written in *italic*. Our approach is formalised through functions (in `typewriter black`) and algorithms (in **bold-red**) that manipulate *MiniGo* programs. Each identifier in *MiniGo* is translated to the equivalent string in Promela, e.g., *ch*1 in *MiniGo* is translated to `ch1`. For the sake of readability, we omit the concatenation operator between literal Promela strings and strings generated by translation functions.

Function declarations Given a function body $\tilde{s}$, for each distinct (blocking) function call $id(\widetilde{a})$ occurring inter-procedurally in $\tilde{s}$ such that $\texttt{chans}(\widetilde{a}) \neq []$, we define a Promela process (`proctype`) as follows:

$$\texttt{proctype } id\texttt{(chanParams(}id\texttt{),ch)\{TransStmts(}\varnothing\texttt{,body(}id\texttt{));ch!0\}}$$

where `ch` is a channel used to signal the termination of the function call (with `ch!0`).
For each distinct non-blocking function call `go` $id(\widetilde{a})$, such that $\texttt{chans}(\widetilde{a}) \neq []$, we define the process:

$$\texttt{proctype go\_}id\texttt{(chanParams(}id\texttt{))\{TransStmts(}\varnothing\texttt{,body(}id\texttt{))\}}$$

where $\texttt{chanParams}(id)$ (resp. $\texttt{body}(id)$) returns the *channel* parameters (resp. body) of function *id* and $\varnothing$ denotes the empty map. Observe that the non-channel parameters are abstracted away. We use

```
function TransStmts(Δ, s s̃)
    switch s :
        case ch ← e   ch.in!0;ch.sending?state;TransStmts(Δ, s̃)
        case v ← ch   ch.in?0;TransStmts(Δ, s̃)
        case close(ch)   ch.closing?state;TransStmts(Δ, s̃)
        case if e then s̃1 else s̃2
            if
                :: true ->  TransStmts(Δ, s̃1)
                :: true ->  TransStmts(Δ, s̃2)
            fi; TransStmts(Δ, s̃)
        case select{c̃}
            if
                TransStmts(Δ, c̃)
            fi; TransStmts(Δ, s̃)
        case case α : s̃1 ::  TransStmts(Δ, α) -> TransStmts(Δ, s̃1)
        case default : {s̃1} ::  true -> TransStmts(Δ, s̃1)
        case id(ã)
            if chans(ã) ≠ [] then
                ch = [0] of {int}; run id(chans(ã), ch);ch?0; TransStmts(Δ, s̃)
            otherwise TransStmts(Δ, s̃)
        case go id(ã)
            if chans(ã) ≠ [] then
                run go_id (chans(ã)); TransStmts(Δ, s̃)
            otherwise TransStmts(Δ, s̃)
        case for v :=e1 ;e2 ;r {s̃3}
            let (Δ′, x, y) = lookup_Δ(v :=e ;e ;r) in
            if spawns(s̃3) ∨ Δ == Δ′ then
            for (i: x .. y ){TransStmts(Δ′, s̃3)}; TransStmts(Δ′, s̃)
            otherwise
              do    :: true ->  TransStmts(Δ, s̃3)
                    :: true ->  break;
              od;
            TransStmts(Δ, s̃)
        case ch := make(chan, e)
            let (Δ′, _, y) = lookup_Δ(i := 0; i < e; i++) in
            Chandef ch;
            chan ch.in = [y] of {int};
            run chanmonitor(ch);
            TransStmts(Δ′, s̃)
```

Algorithm 1: Extracting Promela from *MiniGo* statements. We assume that `TransStmts`$(\Delta, [])$ returns the empty string.

`proctype` instead of `inline` definition as the latter cannot include declarations of new channels. Next, we define function `TransStmts` which translates *MiniGo* statements to Promela.

Algorithm 1 specifies how we extract a model from a list of *MiniGo* statements. We use b to range over the control statements of a for-loop, i.e., b ranges over triples of the form $(v := e_1 ; e_2 ; r)$.

Function `TransStmts` takes two parameters: (1) Δ maps expressions (corresponding to communication-related parameters) to Promela strings, and (2) a list of *MiniGo* statements.

Channel primitives For each *MiniGo* channel we generate a custom Promela structure, called `Chandef`, which contains three channels: `in` carries the exchanged messages, while `sending` (resp. `closing`) is used to monitor send (resp. closing) actions. A send statement is translated to a send statement in Promela (on channel `in`), followed by a receive statement on the corresponding channel monitor (on the `sending` channel, see below). A receive statement is translated to a Promela receive (on channel `in`). A close statement is translated to a Promela receive on channel `closing`.

$$\downarrow b = \begin{cases} e_1 & \text{if } b \text{ is } v := e_1;\ v < e_2;\ v\texttt{++} \\ e_2 & \text{if } b \text{ is } v := e_1;\ v > e_2;\ v\texttt{--} \\ \bot & \text{otherwise} \end{cases} \qquad \uparrow b = \begin{cases} e_2 & \text{if } b \text{ is } v := e_1;\ v < e_2;\ v\texttt{++} \\ e_1 & \text{if } b \text{ is } v := e_1;\ v > e_2;\ v\texttt{--} \\ \bot & \text{otherwise} \end{cases}$$

$$\texttt{lookup}_\Delta(b) = \begin{cases} (\Delta, x, y) & \text{if } \downarrow(b) = \bot \text{ or } \uparrow(b) = \bot, \text{ with } x \text{ and } y \text{ fresh} \\ (\Delta', \Delta'(\downarrow b), \Delta'(\uparrow b)) & \text{otherwise, where } \Delta' = \Delta[e \mapsto x \mid e \in \{\downarrow(b), \uparrow(b)\} \setminus dom(\Delta) \text{ with } x \text{ fresh}] \end{cases}$$

Figure 4: Auxiliary functions for Algorithm 1, where we assume $\Delta(n) = n$ for each integer n.

Conditionals An if-then-else statement is translated to an `if` block in Promela. It behaves as a non-deterministic internal choice (`true` is an always-enabled guard). A select statement is translated to a non-deterministic choice, using an `if` block where each non-default branch is guarded by a send or receive action. Default branches are translated to a branch that is always available.

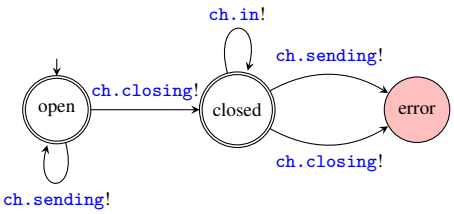
Control statements A blocking function call is translated to Promela code that spawns an instantiation of the corresponding Promela process (using the `run` keyword), then waits for it to terminate by waiting on fresh channel `ch`. For spawning function calls, i.e., `go` $id(\tilde{a})$, there are two cases. If the parameters include channels, the algorithm returns Promela code that spawns the corresponding process. Otherwise it omits the call entirely — as it will be checked in the model of an independent partition.

To translate `for` v`:=`e_1`;`e_2`;`$r\{\tilde{s}_3\}$, we need to first check (*i*) whether we can extract well-identified bounds that we consider as communication-related parameters and (*ii*) whether the loop contains (inter-procedurally) spawning function calls, i.e., `spawns`$(\tilde{s}_3)$ holds (whose straightforward definition is omitted). If `spawns`$(\tilde{s}_3)$ holds then the range of the loop needs to be finite. Hence, either we set-up Promela variables (that the user will instantiate) to define a range; or we are able to identify a static range (integer literal bounds). When the loop does not spawn new threads, we only use a finite Promela for-loop if all involved variables have been seen before (Δ==Δ', see below). In all other cases, we use a non-deterministic loop using a Promela `do` block which can be exited at any iteration with a `break` operation.

We keep track of re-usable communication-related parameters with the map Δ and use the function `lookup`, defined in Figure 4, to query it. Functions $\downarrow(b)$ and $\uparrow(b)$ respectively return the lower and upper bounds of for-loop control statement b. When the control statements of a for-loop are well-formed (they obey a recognisable pattern, i.e., $\downarrow b \neq \bot \wedge \uparrow b \neq \bot$), the `lookup` function returns a new map Δ' and the lower ($\Delta'(\downarrow b)$) and upper ($\Delta'(\uparrow b)$) bounds of the for-loop. Map Δ' augments Δ with mappings from newly identified *expressions* to fresh Promela variables.

Channel creation A channel creation statement is translated to the instantiation of a `Chandef` structure and the spawning of its `chanmonitor` process. We initialise the `in` channel of the `Chandef` structure with a capacity corresponding to its *MiniGo* equivalent. If the capacity is not a integer literal, the `lookup` function ensures that we either re-use Promela variables, or generate fresh ones. Note that channels `sending` and `closing` in `Chandef` are always synchronous.

Channel monitors To detect channel safety errors, we keep track of the state of *MiniGo* channels. We use Promela processes to monitor channel actions, i.e., send, receive, and close. As an optimisation, we only create such monitors when a `close` primitive appears in the program. Processes corresponding to goroutines interact with channel monitors via an instance `ch` of a `ChanDef` structure which contains three channels (`in`, `sending`, and `closing`). The automa-



```
1  #define len_files_0  15
2
3  typedef Chandef {
4    chan in = [0] of {int};
5    chan sending = [0] of {int};
6    chan closing = [0] of {bool};
7  }
8  init {
9    chan _ch0_in = [len_files_0] of {int};
10   Chandef _ch0;
11   _ch0.in = _ch0_in;
12   run chanMonitor(_ch0)
13
14   for(i : 1.. len_files_0) {
15     run mainworker(_ch0)
16   };
17   for(i : 1.. len_files_0) {
18     _ch0.in?0
19   };
20 }
21 proctype mainworker(Chandef a) {
22   a.in!0;
23   a.sending?0
24 }
25
26 proctype chanMonitor(Chandef ch) {
27   bool open = true;
28   do
29   :: true ->
30     if
31     :: open ->
32       if
33       :: ch.sending!open;
34       :: ch.closing!true ->
35         open = false
36       fi
37     :: else ->
38       if
39       :: ch.sending!open ->
40         assert(false)
41       :: ch.closing!true ->
42         assert(false)
43       :: ch.in!0;
44       fi
45     fi
46   od
47 }
```

Figure 5: Model extracted from Listing 1 with Algorithm 1

ton on the right represents the behaviour of this monitor (`chanMonitor`). Figure 5 (lines 26- 47) gives the corresponding Promela code. In *MiniGo* and Go, when a channel is *closed*, sending on it or closing it will raise an error, hence the transitions from state *closed* to state *error* in the automaton. Also, receive actions on closed channels always succeed, hence the self-loop on state *closed*.

Example 1. The model extracted from Figure 1 with Algorithm 1 is given in Figure 5. Lines 8-20 contain the translation of the `main` function. Note that a `chanMonitor` is spawned at line 12. The definition of this process is given in lines 26-47. The translation of the `worker` function is in lines 21-24.

Figure 5 contains one (unknown) communication-related parameter: `len(files)` which is used in the capacity of channel `a`, and in the loops at lines 4 and 7 of Figure 1. Observe that the communication-related parameter `len(files)` is set to 15 here (see line 1). Our implementation also allows user to specify such parameters as program (command line) arguments. Expression `len(files)` is first seen by Algorithm 1 in a channel creation statement. At this point, it adds `len(files)` $\mapsto$ `'len_files_0'` in map Δ. When the algorithm processes both loops, it invokes lookup_{Δ}(`i:=0;i<len(files);i++`) to obtain lower and upper bounds (0 and `'len_files_0'`, respectively). Note that the loop in line 4 is a spawning loop, hence it must be translated to a finite loop in Promela to obtain a finite model. The loop in line 7 is not spawning, but it is still translated to a finite loop because all its bounds are already in Δ.

4 Implementation and evaluation

Implementation We have implemented our approach in a tool called GOMELA [4]. Given a Go program, our tool extracts Promela models (as described in Section 3). If necessary, the user enters values (bounds) for the statically unknown communication-related parameters produced by Algorithm 1 (i.e., the Promela variables). For instance, the user provides a bound to instantiate `len(files)` in Figure 1. This value is then used as a bound in the for-loops at lines 4 and 7 as well as the capacity of channel `a`.

Table 1: Go programs verified by GOMELA and comparison with Godel [8]. All programs are available online [4].

#	Programs & `Partitions`	GOMELA |States|	GOMELA CS	GOMELA GD	GOMELA Infer (ms)	GOMELA Spin (ms)	Godel Time	Godel ψ_s	Godel ψ_g	Manual analysis CS	Manual analysis GD
1	file processing (|`files`|=15)	376880	✓	✓	85	1666	$\perp$	$\perp$	$\perp$	✓	✓
2	file processing v1 (|`files`|=15)	109	✓	✗	86	1057	$\perp$	$\perp$	$\perp$	✓	✗
3	file processing v2 (|`files`|=15)	110	✓	✗	85	1027	$\perp$	$\perp$	$\perp$	✓	✗
4	prod-cons (k=5,n=10,m=10)	4802785	✓	✓	85	31239	$\perp$	$\perp$	$\perp$	✓	✓
5	alt-bit	35	✓	✓	956	1391	460	✓	✓	✓	✓
6	concsys	96	-	-	768	8194	588	✓	✓	-	-
	- `ConcurrentSearch()`	49	✓	✓	-	1126	-	-	-	✓	✓
	- `ConcurrentSearchWC()`	26	✓	✗	-	1335	-	-	-	✓	✗
	- `First()`	3	✓	✓	-	1183	-	-	-	✓	✓
	- `ReplicaSearch()`	9	✓	✗	-	1326	-	-	-	✓	✗
	- `SequentialSearch()`	3	✓	✓	-	1073	-	-	-	✓	✓
	- `FakeSearch()`	3	✓	✓	-	1083	-	-	-	✓	✓
	- `main()`	3	✓	✓	-	1068	-	-	-	✓	✓
7	cond-recur	13	✓	✓	84	1278	623	✓	✓	✓	✓
8	dinephil	968	✓	✓	804	1478	859	✓	✓	✓	✓
9	dinephil5	71439	✓	✓	835	1801	8681	✓	✓	✓	✓
10	double-close	18	✗	-	85	1486	463	✗	✓	✗	✓
11	fanin-alt	34	✓	✗	769	1686	786	✓	✓	✓	✗
12	fanin	15	✓	✓	681	1495	621	✓	✓	✓	✓
13	fixed	14	-	-	740	2379	656	✓	✓	-	-
	- `Work()`	2	✓	✓	-	1117	-	-	-	✓	✓
	- `main()`	12	✓	✓	-	1262	-	-	-	✓	✓
14	forselect	21	✓	✓	755	1507	813	✓	✓	✓	✓
15	jobsched	48	-	-	781	2745	589	✓	✓	-	-
	- `main()`	45	✓	✓	-	1532	-	-	-	✓	✓
	- `morejob()`	3	✓	✓	-	1213	-	-	-	✓	✓
16	mismatch	12	-	-	714	2316	603	✓	✗	-	-
	- `Work()`	2	✓	✓	-	1044	-	-	-	✓	✓
	- `main()`	10	✓	✗	-	1272	-	-	-	✓	✗
17	sel	21	✓	✗	84	1719	326	✓	✗	✓	✗
18	selFixed	19	✓	✓	82	1330	572	✓	✓	✓	✓
19	philo	18	✓	✗	85	1362	537	✓	✗	✓	✗
20	starvephil	67	✓	✗	759	1343	836	✓	✗	✓	✗
21	nonlive	7	✓	✓	850	1194	550	✓	✓	✓	✓
22	nonlive v1	7	✓	✗†	850	1152	366	✓	✗†	✓	✓
23	prod-cons	61	✓	✓	87	1390	508	✓	✓	✓	✓
24	prod3-cons3	5746	✓	✓	643	1534	19963	✓	✓	✓	✓
25	prodconsclose	12185424	✓	✓	163	18672	25348	✓	✓	✓	✓
26	stuckmsg	5	✓	✓	86	1109	790	✓	✓	✓	✓
27	data-dependent	12	✓	✗†	86	1170	694	✓	✗†	✓	✓
	Column number	1	2	3	4	5	6	7	8	9	10

GOMELA uses Spin to check whether each model is free from channel safety errors and global deadlocks. Spin reports any global deadlock and any trace that leads to an `assert(false)` statements. We use the former to check for global deadlocks (GD) in a program's partitions, and the latter to check for channel safety errors (CS). Spin detects when the main process (`init`) terminates while another process is still running. We use this to detect some goroutine leaks, i.e., a particular case of partial deadlocks.

In addition to *MiniGo* statements, GOMELA deals with constants (used as communication-related parameters), anonymous functions, break statements, for range loops and switch statements. Occurrences of integer constants are replaced with their actual values. To deal with anonymous functions, GOMELA generates Promela corresponding function declarations (using fresh names) and its (unique) invocation. Go's `break` statements are translated as Promela break statements. For-ranges loop (`for range` $list\{\tilde{s}_1\}$) are treated as for-loops with control statements of the form: to `for` i`:=0;`$i < len(list)$`;i++`. Finally, `switch` statements are translated into n-ary internal choices (similar to an if-then-else).

Evaluation To evaluate our tool, we ran it on several benchmarks, including some from [8, Table 1]. The results of this evaluation are in Table 1. In Table 1, Column 1 gives the number of states in the model, as given by Spin. In Column 2 (resp. 3) a ✓-mark says that the corresponding program partition is channel-

safe (resp. free of global deadlock); a ✗-mark says that the property is violated. Column 4 (resp. 5) shows the time (milliseconds) taken to extract (resp. verify) the Promela model of a partition. The timing for programs are the sum of all of their partitions. Column 6 shows the time (milliseconds) taken by Godel [8] to verify channel safety (ψ_s, Column 7) and global deadlock (ψ_g, Column 8) properties in. A $\perp$-mark means that Godel does not support this program. A ✓-mark in Column 9 (resp. 10) says that it was not possible to find any channel errors (resp. global deadlocks) manually, ✗† highlights false alarms.

In Table 1, Program 1 is the example in Figure 1 where the user has set `len(files)` to 15. Programs 2 and 3 are variations of Program 1 with a global deadlock and a goroutine leak, respectively. Program 4 spawns `n` producers and `m` consumers that interact (repeatedly) over a channel with capacity `k`. Observe that these are not supported by Godel because they include thread-spawning in a for-loop. Programs 5 to 26 are taken from [8, Table 1]. We note that Godel runs marginally faster than GOMELA on programs with small models. This is due to Spin's start up time of $\sim$1s. For larger programs (more than 5000 states) GOMELA performs much better than Godel, see, e.g., Programs 9, 24, and 25. This suggests that our tool scales better than Godel on larger code-bases.

Below we comment on the errors identified in the programs from Table 1.

- Program 6 has two partitions that contain global deadlocks. In `ConcurrentSearchWC`, a function spawns three goroutines that send a message on a shared channel `c`. That function may terminate silently (via timeout) hence leaving three unmatched send actions in the goroutines. `ReplicaSearch` includes a similar pattern where the parent thread may terminate while leaving some goroutines permanently blocked.
- Program 10 contains a channel safety error where a channel is closed twice by two different threads. We note that Spin aborts the verification as soon as it finds such errors.
- Program 11 creates two producers and one consumer. The consumer may terminate silently in which case both producers are blocked permanently. Program 12 is similar to Program 11 except that the consumer never terminates, thus fixing the bug.
- Program 16 contains two partitions: `Work` consist of a non-terminating loop (without communications), while `main` contains a global deadlock due to a mismatch between the number of send and receive actions. Note that if we were modelling this program with one monolithic partition, we would not detect a global deadlock because `Work` never blocks.
- In Program 17 each goroutine may get stuck because of a mismatch between the number of send and receive actions.
- Program 19 is an encoding of the (starving) philosopher problem using only two philosophers, taken from [14]. Program 20 is another encoding of the (starving) dining philosophers from [8].
- Programs 21 and 22 are two variants of a program that contain two goroutines: one is waiting for a message, while the other contains a non-terminating for-loop. In Program 21, the non-terminating for-loop is followed by a (unreachable) matching send action. These programs show the limits of the behavioural types approach: they contain proper partial deadlocks. The problem in Programs 21 is only detected in Godel by using an additional termination checker.
- Program 27 is given in Figure 6. This program gives a typical example of a false alarm raised by our tool, and any existing approach based on behavioural types. Because if-then-else statements are translated to non-deterministic choices, our approach is unable to determine that the two conditional blocks are "synchronised" by the same invocation of function `f()`.

Limitations Our approach is applicable to *MiniGo*, extended with the syntactic constructs discussed above. Several key limitations need to be tackled to address the full Go language. We assume that variables are immutable, as a consequence we cannot soundly analyse programs that, e.g., mutate a list

```
1  func main() {
2    a := make(chan int)
3    go send(a)
4    if f() {  // decl. of f() elided
5      a <- 0
6    } else {}
7  }
8
```

```
9  func send(a chan int) {
10   if f() {
11     <-a
12   } else {}
13 }
14 /* f() is a deterministic function
      without side-effects */
```

Figure 6: Data-dependent choice (Program 27).

`files` in between using `len(files)` as a communication-related parameter. Go has object oriented-like features, such as structs, methods, and interfaces which we currently do not support. Virtual method calls (on interfaces) are particularly difficult to model. As in [7, 8, 10, 12, 14], we do not support channel passing (since we abstract away the data sent over channels). We note that our empirical survey [3] found that only 6% of projects used channels that carry channels.

5 Related work, conclusions and future work

Related work Spin and Promela have been used extensively in software verification. Notably Java PathFinder [5] translates Java programs to Promela models which are then verified for deadlocks and violations of user-provided assertions. Also, Zaks and Joshi [19] use Spin to verify multi-threaded C programs using their LLVM representation and custom virtual machine.

Several works focus on the verification of message-passing concurrency in Go [7, 8, 10, 12, 14, 15, 16]. Four papers studied static verification using behavioural models. Ng and Yoshida [12] proposed *dingohunter*, the first static global deadlock detection tool for Go. It relies on communicating finite-states machines [1] and multiparty compatibility [9]. Their work does not support asynchronous channels nor programs that spawn goroutines or create channels in loops or conditionals. Stadtmüller et al. introduced *gopherlyzer* [14] which detects global deadlocks using forkable regular expressions. This work does not support channel closures, asynchronous channels, nor goroutines spawned in loops. Lange et al. [7, 8] proposed Gong and Godel, whose approach serves as a basis for this work. Gong uses an *ad-hoc* checker which supports bounded verification of infinite-state models, but did not scale well. Instead, Godel uses mCRL2 [2] as a back-end. Because mCRL2's communication model is very different from Go's the encoding from behavioural types to mCRL2's language is very intricate, see [6]. Promela is a more convenient language for this purpose, but because Spin supports only LTL, while mCRL2 supports the μ-calculus, it is not possible to check the liveness property specified in [7, 8]. However, we can still identify some goroutine leaks by checking whether their corresponding models reach their end states.

Conclusions Our work builds on the approach in [8] and improves it to support statically unknown communication-related parameters via a bounded analysis. Our approach allows us to support programs that spawn a parameterised number of goroutines or channel capacities. Our evaluation shows that our tool scales well and produces models that can be easily understood and adjusted by programmers.

Future work Our short term plans are to support additional concurrency-related Go features, e.g., barriers (`WaitGroup`). We will also improve our algorithms to support more complex for-loops control statements, and to perform a fully *inter*-procedural analysis of communication-related parameters. In the longer term, we plan to use our tool to detect concurrency errors and suggest repairs for large code-bases. We plan to perform a large-scale empirical evaluation of this toolchain on the dataset identified in [3].

References

[1] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *J. ACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[2] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink & Tim A. C. Willemse (2013): *An Overview of the mCRL2 Toolset and Its Recent Advances*. In: *TACAS*, *Lecture Notes in Computer Science* 7795, Springer, pp. 199–213, doi:10.1007/978-3-642-36742-7_15.

[3] Nicolas Dilley & Julien Lange (2019): *An Empirical Study of Messaging Passing Concurrency in Go Projects*. In: *SANER*, IEEE, pp. 377–387, doi:10.1109/SANER.2019.8668036.

[4] Nicolas Dilley & Julien Lange (2020): *Gomela*. `http://github.com/nicolasdilley/gomela`.

[5] Klaus Havelund & Thomas Pressburger (2000): *Model Checking JAVA Programs using JAVA PathFinder*. *STTT* 2(4), pp. 366–381, doi:10.1007/s100090050043.

[6] Julien Lange, Nicholas Ng & Bernardo Toninho: *Godel Checker*. `https://bitbucket.org/MobilityReadingGroup/godel-checker`.

[7] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2017): *Fencing off Go: liveness and safety for channel-based programming*. In: *POPL*, ACM, pp. 748–761, doi:10.1145/3009837.3009847.

[8] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2018): *A static verification framework for message passing in Go using behavioural types*. In: *ICSE*, ACM, pp. 1137–1148, doi:10.1145/3180155.3180157.

[9] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In: *POPL*, ACM, pp. 221–232, doi:10.1145/2676726.2676964.

[10] Jan Midtgaard, Flemming Nielson & Hanne Riis Nielson (2018): *Process-Local Static Analysis of Synchronous Processes*. In: *SAS*, *Lecture Notes in Computer Science* 11002, Springer, pp. 284–305, doi:10.1007/978-3-319-99725-4_18.

[11] Robin Milner (1984): *Lectures on a calculus for communicating systems*. In: *International Conference on Concurrency*, Springer, pp. 197–220, doi:10.1007/3-540-15670-4_10.

[12] Nicholas Ng & Nobuko Yoshida (2016): *Static deadlock detection for concurrent Go by global session graph synthesis*. In: *CC*, ACM, pp. 174–184, doi:10.1145/2892208.2892232.

[13] Rob Pike (2015): *Go Proverbs*. `https://www.youtube.com/watch?v=PAAkCSZUG1c`.

[14] Kai Stadtmüller, Martin Sulzmann & Peter Thiemann (2016): *Static Trace-Based Deadlock Analysis for Synchronous Mini-Go*. In: *APLAS*, *Lecture Notes in Computer Science* 10017, pp. 116–136, doi:10.1007/978-3-319-47958-3_7.

[15] Martin Sulzmann & Kai Stadtmüller (2017): *Trace-Based Run-Time Analysis of Message-Passing Go Programs*. In: *Haifa Verification Conference*, *Lecture Notes in Computer Science* 10629, Springer, pp. 83–98, doi:10.1007/978-3-319-70389-3_6.

[16] Martin Sulzmann & Kai Stadtmüller (2018): *Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks*. In: *PPDP*, ACM, pp. 22:1–22:13, doi:10.1145/3236950.3236959.

[17] The Go team (2016-17): *Go Survey Results*. `blog.golang.org/survey[year]-results`.

[18] Tengfei Tu, Xiaoyu Liu, Linhai Song & Yiying Zhang (2019): *Understanding Real-World Concurrency Bugs in Go*. In: *ASPLOS*, ACM, pp. 865–878, doi:10.1145/3297858.3304069.

[19] Anna Zaks & Rajeev Joshi (2008): *Verifying Multi-threaded C Programs with SPIN*. In: *SPIN*, *Lecture Notes in Computer Science* 5156, Springer, pp. 325–342, doi:10.1007/978-3-540-85114-1_22.

Mixed Sessions: the Other Side of the Tape

Filipe Casal Andreia Mordido Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
fmrcasal@fc.ul.pt afmordido@fc.ul.pt vmvasconcelos@fc.ul.pt

Vasconcelos et al. [11] introduced side A of the tape: there is an encoding of classical sessions into mixed sessions. Here we present side B: there is translation of (a subset of) mixed sessions into classical sessions. We prove that the translation is a minimal encoding, according to the criteria put forward by Kouzapas et al. [6].

1 Classical Sessions, Mixed Sessions

Mixed sessions were introduced by Vasconcelos et al. [11] as an extension of classical session types [3, 5, 10]. They form an interesting point in the design space of session-typed systems: an extremely concise process calculus (four constructors only) that allows the natural expression of algorithms quite cumbersome to write in classical sessions. The original paper on mixed sessions [11] shows that there is an encoding of classical sessions into mixed sessions. This abstract shows that the converse is also true for a fragment of mixed sessions.

A translation of mixed sessions into classical sessions would allow to leverage the tools available for the latter: one could program in mixed sessions, translate the source code into classical sessions, check the validity of the source code against the type system for the target language, and run the original program under an interpreter for classical sessions (SePi [2], for example). A mixed-to-classical encoding would further allow a better understanding of the relative expressiveness of the two languages.

Processes in classical binary sessions [3, 4, 5, 9] (here we follow the formulation in [10]) communicate by exchanging messages on bidirectional channels. We introduce classical sessions by means of a few examples. Each channel is denoted by its two ends and introduced in a process P as $(\mathbf{new}\ xy)P$. Writing a value v on channel end x and continuing as P is written as $x!v.P$. Reading a value from a channel end y, binding it to variable z and continuing as Q is written as $y?z.Q$. When the two processes get together under a new binder that ties together the two ends of the channel, such as in

$$(\mathbf{new}\ xy)\ x!v.P \mid y?z.Q$$

value v is communicated from the x channel end to the y end. The result is process $(\mathbf{new}\ xy)\,P \mid Q[v/z]$, where notation $Q[v/z]$ denotes the result of replacing v for z in Q.

Processes may also communicate by offering and selecting options in choices. The different choices are denoted by labels, ℓ and m for example. To select choice ℓ on channel end x and continue as P we write $x\ \mathbf{select}\ \ell.P$. To offer a collection of options at channel end y and continue with appropriate continuations Q and R, we write $\mathbf{case}\ y\ \mathbf{of}\ \{\ell \to Q, m \to R\}$. When **select** and **case** processes are put together under a **new** that binds together the two ends of a channel, such as in

$$(\mathbf{new}\ xy)\ x\ \mathbf{select}\ \ell.P \mid \mathbf{case}\ y\ \mathbf{of}\ \{\ell \to Q,\ m \to R\}$$

branch Q is selected in the **case** process. The result is the process $(\mathbf{new}\ xy)\,P \mid Q$. Selecting a choice is called an internal choice, offering a collection of choices is called an external choice. We thus see that

S. Balzer, L. Padovani (Eds.): Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 46–60, doi:10.4204/EPTCS.314.5

classical sessions comprise four atomic interaction primitives. Furthermore, choices are directional in the sense that one side offers a collection of possibilities, the other selects one of them.

To account for unbounded behavior classical sessions count with replication: an input process that yields a new copy of itself after reduction, written $y*?z.Q$. A process of the form

```
(new xy) x!v.P | y*?z.Q
```

reduces to (new xy) P | Q[v/z] | y*?z.Q. If we use the **lin** prefix to denote an ephemeral process and the **un** prefix to denote a persistent process, an alternative syntax for the above process is (new xy) **lin** x!v.P | **un** y?z.Q.

Mixed sessions blur the distinction between internal and external choice. Under a unified language construct—mixed choice—processes may non-deterministically select one choice from a multiset of output choices, or branch on one choice, again, from a multiset of possible input choices. Together with an output choice, a value is (atomically) sent; together with an input choice, a value is (again, atomically) received, following various proposals in the literature [1, 8, 12]. The net effect is that the four common operations on session types—output, input, selection, and branching—are effectively collapsed into one: mixed choice. Mixed choices can be labelled as ephemeral (linear, consumed by reduction) or persistent (unrestricted, surviving reduction), following conventional versus replicated inputs in some versions of the pi-calculus [7]. Hence, in order to obtain a core calculus, all we have to add is name restriction, parallel composition, and inaction (the terminated process), all standard in the pi-calculus.

We introduce mixed sessions by means of a few examples. Processes communicate by offering/selecting choices with the same label and opposite polarities.

```
(new xy) lin x (m!3.P + n?z.Q) | lin y (m?w.R + n!5.S + p!7.T)
```

The above processes communicate over the channel with ends named x and y and reduce in one step along label m to (new xy) P | R[3/w] or along label n to (new xy) Q[5/z] | S.

Non-determinism in mixed sessions can be further achieved by allowing duplicated labels in choices. An example in which a 3 or a 5 is non-deterministically sent over the channel is

```
(new xy) lin x (m!3.P + m!5.Q) | lin y (m?z.R)
```

This process reduces in one step to either (new xy) P | R[3/z] or (new xy) Q | R[5/z]. Unrestricted behavior in choices is achieved by the **un** qualifier in the choice syntax.

```
(new xy) un x (m!3.P + m!5.P) | un y (m?z.Q)
```

This process reduces to itself together with either of the choices taken,

```
(new xy)
un x (m!3.P + m!5.P) |
un y (m?z.Q) |
P | Q[3/z]
```

or

```
(new xy)
un x (m!3.P + m!5.P) |
un y (m?z.Q) |
P | Q[5/z]
```

The complete set of definitions for the syntax, operational semantics, and type system for mixed sessions are in appendix, Figures 6 to 8. For technical details and main results, we direct the reader to reference [11]. The complete set of definitions for the syntax, operational semantics, and type system for classical sessions are in appendix, Figure 9. For further details, we refer the reader to references [10, 11].

2 Mixed Sessions as Classical Sessions

This section shows that a subset of the language of mixed sessions can be embedded in that of classical sessions. We restrict our attention to choices that reduce against choices with the same qualifier, that is,

```
new x y: lin&{m: !int.end,
              n: ?bool.end}

// lin x (m!3.0 + n?w.0)
case x of
    m → new s1 t1: ∗+{ℓ}
        s1 select ℓ |
        case t1 of
            ℓ → x!3
    n → new s2 t2: ∗+{ℓ}
        s2 select ℓ |
        case t2 of
            ℓ → x?w

|

// lin y (m?z.0)
new s3 t3: ∗+{ℓ}
s3 select ℓ |
case t3 of
    ℓ → y select m.
        new s4 t4: ∗+{ℓ}
        s4 select ℓ |
        case t4 of
            ℓ → y?z
```

Figure 1: Translation of (**new** xy)(**lin** x (m!3.0 + n?w.0) | **lin** y (m?z.0))

we do not consider the case where an ephemeral (lin) process reduces against a persistent (un) one. For this reason, we assume that a process and its type always have the same lin/un qualifier.

One of the novelties in mixed sessions is the possible presence of duplicated label-polarity pairs in choices. This introduces a form of non-determinism that can be easily captured in classical sessions. The NDchoice classical session process creates a race condition on a new channel with endpoints s,t featuring multiple selections on the s endpoint, for only one branch on the t endpoint. This guarantees that exactly one of the branches is non-deterministically selected. The remaining selections must eventually be garbage collected. We assume that $\prod_{1\leq i\leq n} Q_i$ denotes the process $Q_1 \mid \ldots \mid Q_n$ for $n > 0$, and that Π binds tighter than the parallel composition operator.

$$\text{NDChoice}\{P_i\}_{i\in I} = (\nu st)\left(\prod_{i\in I} s \lhd l_i.\mathbf{0} \mid t \rhd \{l_i\colon P_i\}_{i\in I}\right)$$

The type S of channel end s is of the form $\mathsf{un}\oplus\{l_i\colon S\}_{i\in I}$, an equation that can be solved by type $\mu a.\mathsf{un}\oplus\{l_i\colon a\}_{i\in I}$, and which SePi abbreviates to $*\oplus\{l_i\}_{i\in I}$. The qualifier must be un because s occurs in multiple threads in NDChoice; recursion arises because of the typing rules for processes reading or writing in unrestricted channels.

Equipped with NDChoice we describe the translation of mixed sessions to classical sessions via variants of the examples in Section 1. All examples fully type check and run in SePi [2]. To handle duplicated label-polarity pairs in choices, we organize choice processes by label-polarity fragments. Each such fragment represents a part of a choice operation where all possible outcomes have the same label and polarity. When a reduction occurs, one of the branches is taken, non-deterministically, using the NDChoice operator. After a non-deterministic choice of the branch, and depending on the polarity of the fragment, the process continues by either writing on or reading from the original channel.

The translation of choice processes is guided by their types. For each choice we need to know its qualifier (lin, un) and its view ($\oplus$, &), and this information is present in types alone.

Figure 1 shows the translation of the mixed process (**new** xy)(**lin** x (m!3.0 + n?w.0) | **lin** y (m?z.0)), where x is of type **lin**&{m!**int**.**end**, n?**bool**.**end**}. The corresponding type in classical sessions is **lin**&{m:!**int**.**end**, n:?**bool**.**end**}, which should not come as a surprise. Because channel end x is of an external choice type (&), the choice on x is encoded as a **case** process. The other end of the channel, y, is typed as an internal choice ($\oplus$) and is hence translated as a **select** process. Occurrences of the NDChoice process

```
new x y: lin&{m: !int.end}

// lin x (m!3.0 + m!5.0)
case x of
    m → new s1 t1: ∗+{ℓ1,ℓ2}
        s1 select ℓ1 |
        s1 select ℓ2 |
        case t1 of
            ℓ1 → x!3
            ℓ2 → x!5
|
// lin y (m?z.0)
new s2 t2: ∗+{ℓ}
s2 select ℓ |
case t2 of
    ℓ → y select m.
        new s3 t3: ∗+{ℓ}
        s3 select ℓ |
        case t3 of
            ℓ → y?z
```

Figure 2: Translation of (**new** xy)(**lin** x (m!3.0 + m!5.0) | **lin** y (m?z.0))

appear in a degenerate form, always applied to a single branch. We have four of them: three for each of the branches in **case** processes (s_1t_1, s_2t_2, and s_4t_4) and one for the external choice in the mixed session process (s_3t_3).

In general, an external choice is translated into a classical branching (**case**) over the unique labels of the fragments of the process, but where the polarity of each label is inverted. The internal choice, in turn, is translated as (possibly nondeterministic collection of) classical **select** process but keeps the label polarity. This preserves the behavior of the original process: in mixed choices, a reduction occurs when a branch $l^!v.P$ matches another branch $l^?z.Q$ with the same label but with dual polarity ($l^!$ against $l^?$), while in a classical session the labels alone must match (l against l). Needless to say, we could have followed the strategy of dualizing internal choices rather than external.

If we label reduction steps with the names of the channel ends on which they occur, we can see that, in this case a $\xrightarrow{xy}$ reduction step in mixed sessions is mimicked by a long series of classical reductions, namely $\xrightarrow{s_3t_3}\xrightarrow{xy}\xrightarrow{s_1t_1}\xrightarrow{s_4t_4}\xrightarrow{xy}$ or $\xrightarrow{s_3t_3}\xrightarrow{xy}\xrightarrow{s_4t_4}\xrightarrow{s_1t_1}\xrightarrow{xy}$. Notice the three reductions to resolve non-determinism (on s_it_i) and the two reductions on xy to encode branching followed by message passing, an atomic operation in mixed sessions.

Figure 2 shows an example of a mixed choice process with a duplicated label-polarity pair, m!. If we assign to x type **lin**&{m!**int**}, then we know that the choice on x is encoded as **case** and that on y as **select**. In this case, the NDChoice operator is applied in a non-degenerate manner to decide whether to send the values 3 or 5 on x channel end, by means of channel s_1t_1. Again we can see that the one step reduction on channel xy in the original mixed session process originates a sequence of five reduction steps in classical sessions, namely $\xrightarrow{s_2t_2}\xrightarrow{xy}\xrightarrow{s_1t_1}\xrightarrow{s_3t_3}\xrightarrow{xy}$ or $\xrightarrow{s_2t_2}\xrightarrow{xy}\xrightarrow{s_3t_3}\xrightarrow{s_1t_1}\xrightarrow{xy}$. In this case, however, the computation is non-deterministic: the last reduction step may carry integer 3 or 5.

Figure 3 shows the encoding of mixed choices on unrestricted channels. The mixed choice process is that of Figure 2 only that the two ephemeral choices (**lin**) have been replaced by their persistent counterparts (**un**). The novelty, in this case, is the loops that have been created around the **case** and the **select** process. Loops in classical sessions can be implemented with a replicated input: a process of the form v∗?x.P is a persistent process that, when invoked with a value v becomes the parallel composition P[v/x] | v∗?x.P. The general form of the loops we are interested in are (**new** uv : ∗!())(u!() | v∗?x.P), where *continue calls* in process P are of the form u!(). The contents of the messages that control the loop are not of interest and so we use the unit type (), so that u is of type ∗!(). We can easily see the calls

```
type Unr = lin&{m: !integer.end}
new x y: *?Unr

// un x (m!3.0 + m!5.0)
new u1 v1: *!()
u1!() |
v1*?(). x?a.
case a of
    m → new s1 t1: *+{ℓ1,ℓ2}
          s1 select ℓ1 |
          s1 select ℓ2 |
          case t1 of
              ℓ1 → a!3 . u1!()
              ℓ2 → a!5 . u1!()
```

```
// un y (m?z.0)
new u2 v2: *!()
u2!() |
v2*?().
new s t: *+{ℓ}
s select ℓ |
case t of
    ℓ → new a b: Unr
        y!a . b select m .
        new s2 t2: *+{ℓ}
        s2 select ℓ |
        case t2 of
            ℓ → b?z . u2!()
```

Figure 3: Translation of (**new** xy)(**un** x (m!3.0 + m!5.0)| **un** y (m?z.0))

$$
\begin{array}{lll}
(\!|\mathsf{lin}\oplus\{l_i^\star S_i.T_i\}_{i\in I}|\!) = \mathsf{lin}\oplus\{l_i{}^\star\colon \mathsf{lin}\star_i(\!|S_i|\!).(\!|T_i|\!)\}_{i\in I} & & \\
(\!|\mathsf{lin}\,\&\{l_i^\star S_i.T_i\}_{i\in I}|\!) = \mathsf{lin}\,\&\{l_i^\bullet\colon \mathsf{lin}\star_i(\!|S_i|\!).(\!|T_i|\!)\}_{i\in I} & & \text{where } \star_i \perp \bullet_i \\
(\!|\mathsf{un}\oplus\{l_i^\star S_i.T_i\}_{i\in I}|\!) = \mu b.\mathsf{un}!(\mathsf{lin}\oplus\{l_i{}^\star\colon \mathsf{lin}\star_i(\!|S_i|\!).\mathsf{end}\}_{i\in I}).b & & \text{where } T_i \approx \mathsf{un}\oplus\{l_i^\star S_i.T_i\}_{i\in I} \\
(\!|\mathsf{un}\,\&\{l_i^\star S_i.T_i\}_{i\in I}|\!) = \mu b.\mathsf{un}?(\mathsf{lin}\,\&\{l_i^\bullet\colon \mathsf{lin}\star_i(\!|S_i|\!).\mathsf{end}\}_{i\in I}).b & & \text{where } \star_i \perp \bullet_i \text{ and } T_i \approx \mathsf{un}\,\&\{l_i^\star S_i.T_i\}_{i\in I}
\end{array}
$$

(Homomorphic for end, unit, bool, $\mu a.T$, and a)

Figure 4: Translating mixed session types to traditional session types

$u_1!()$ and $u_2!()$ in the last lines in Figure 3, reinstating the unrestricted choice process. In this case, one step reduction in mixed sessions corresponds to a long sequence of transitions in their encodings.

We now present translations for types and processes in general. The translation of mixed choice session types into classical session types is in Figure 4. In general, the (atomic) branch-communicate nature of mixed session types, $\{l_i^\star S_i\}$, is broken in its two parts, $\{l_i\colon \star S_i\}$, branch first, communicate after. In mixed sessions, choice types are labelled by label-polarity pairs ($l^!$ or $l^?$); in classical session choices are labelled by labels alone. Because we want the encoding of a label $l^!$ to match the encoding of $l^?$, we must dualize one of them. We arbitrarily chose do dualize the labels in the & type. The typing rules for classical unrestricted processes of type $S = \mathsf{un}\sharp\{l_i^\star S_i.T_i\}_{i\in I}$ require T_i to be equivalent ($\approx$) to S itself. We take advantage of this restriction when translating un types.

The translation of mixed choice processes is in Figure 5. Since the translation is guided by the type of the process to be translated, we also provide the typing context to the translation function, hence the notation $(\!|\Gamma \vdash P|\!)$. Because label-polarity pairs may be duplicated in choice processes, we organize such processes in label-polarity fragments, so that a process of the form $qx\sum_{i\in I} l_i^{\star_i} v_i.P_i$ (where $q ::= \mathsf{lin} \mid \mathsf{un}$ and $\star ::= ! \mid ?$) can be written as $qx\sum_{i\in I}(\sum_{j\in J} l_i^! v_{ij}.P_{ij} + \sum_{k\in K} l_i^? y_{ik}.P'_{ik})$. Each label-polarity fragment ($l_i^!$ or $l_i^?$) groups together branches with the same label and the same polarity. Such fragments may be empty for external choices, for not all label-polarity pairs in an external choice type need to be covered in the corresponding process (internal choice processes do not need to cover all choices offered by the external

$$(\!|\Gamma \vdash \mathsf{lin} x \sum_{i\in I}(\sum_{j\in J} l_i^! v_{ij}.P_{ij} + \sum_{k\in K} l_i^? y_{ik}.P'_{ik})|\!) = x \rhd \{l_i^? :\ \mathrm{NDChoice}\{x!v_{ij}.(\!|\Gamma_3, x\colon T_i \vdash P_{ij}|\!)\}_{j\in J},$$
$$l_i^! :\ \mathrm{NDChoice}\{x?y_{ik}.(\!|\Gamma_2 \circ \Gamma_3, x\colon T'_i, y_{ik}\colon S'_i \vdash P'_{ik}|\!)\}_{k\in K}\}_{i\in I}$$

where $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$ and $\Gamma_1 \vdash x\colon \mathsf{lin}\&\{l_i^! S_i.T_i, l_i^? S'_i.T'_i\}_{i\in I}$ and $\Gamma_2 \vdash v_{ij}\colon S_i$.

$$(\!|\Gamma \vdash \mathsf{lin} x \sum_{i\in I}(\sum_{j\in J} l_i^! v_{ij}.P_{ij} + \sum_{k\in K} l_i^? y_{ik}.P'_{ik})|\!) = \mathsf{NDChoice}\{x \lhd l_i^!.\mathrm{NDChoice}\{x!v_{ij}.(\!|\Gamma_3, x\colon T_i \vdash P_{ij}|\!)\}_{j\in J},$$
$$x \lhd l_i^?.\mathrm{NDChoice}\{x?y_{ik}.(\!|\Gamma_2 \circ \Gamma_3, x\colon T'_i, y_{ik}\colon S'_i \vdash P'_{ik}|\!)\}_{k\in K}\}_{i\in I}$$

where $\Gamma = \Gamma_1 \circ \Gamma_2 \circ \Gamma_3$ and $\Gamma_1 \vdash x\colon \mathsf{lin}\oplus\{l_i^! S_i.T_i, l_i^? S'_i.T'_i\}_{i\in I}$ and $\Gamma_2 \vdash v_{ij}\colon S_i$.

$$(\!|\Gamma \vdash \mathsf{un} x \sum_{i\in I}(\sum_{j\in J} l_i^! v_{ij}.P_{ij} + \sum_{k\in K} l_i^? y_{ik}.P'_{ik})|\!) = (\nu uv)(u!() \mid \mathsf{un}\, v?\_.x?a.$$
$$a \rhd \{l_i^? :\ \mathrm{NDChoice}\{a!v_{ij}.(u!() \mid (\!|\Gamma \vdash P_{ij}|\!))\}_{j\in J},$$
$$l_i^! :\ \mathrm{NDChoice}\{a?y_{ik}.(u!() \mid (\!|\Gamma, y_{ik}\colon S'_i \vdash P'_{ik}|\!))\}_{k\in K}\}_{i\in I})$$

where $\mathsf{un}(\Gamma)$ and $\Gamma \vdash x\colon \mathsf{un}\&\{l_i^! S_i.T_i, l_i^? S'_i.T'_i\}_{i\in I}$ and $\Gamma \vdash v_{ij}\colon S_i$ and $T_i \approx T'_i \approx \mathsf{un}\sharp\{l_i^! S_i.T_i, l_i^? S'_i.T'_i\}_{i\in I}$.

$$(\!|\Gamma \vdash \mathsf{un} x \sum_{i\in I}(\sum_{j\in J} l_i^! v_{ij}.P_{ij} + \sum_{k\in K} l_i^? y_{ik}.P'_{ik})|\!) = (\nu uv)(u!() \mid \mathsf{un}\, v?\_.$$
$$\mathsf{NDChoice}\{(\nu ab)x!a.b \lhd l_i^!.\mathrm{NDChoice}\{b!v_{ij}.(u!() \mid (\!|\Gamma \vdash P_{ij}|\!))\}_{j\in J},$$
$$(\nu ab)x!a.b \lhd l_i^?.\mathrm{NDChoice}\{b?y_{ik}.(u!() \mid (\!|\Gamma, y_{ik}\colon S'_i \vdash P'_{ik}|\!))\}_{k\in K}\}_{i\in I})$$

where $\mathsf{un}(\Gamma)$ and $\Gamma \vdash x\colon \mathsf{un}\oplus\{l_i^! S_i.T_i, l_i^? S'_i.T'_i\}_{i\in I}$ and $\Gamma \vdash v_{ij}\colon S_i$ and $T_i \approx T'_i \approx \mathsf{un}\sharp\{l_i^! S_i.T_i, l_i^? S'_i.T'_i\}_{i\in I}$.

$$(\!|\Gamma \vdash (\nu xy)P|\!) = (\nu xy)(\!|\Gamma, x\colon S, y\colon T \vdash P|\!) \qquad \text{where } S \perp T$$
$$(\!|\Gamma_1 \circ \Gamma_2 \vdash P_1 \mid P_2|\!) = (\!|\Gamma_1 \vdash P_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!)$$
$$(\!|\Gamma \vdash \mathbf{0}|\!) = \mathbf{0}$$
$$(\!|\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\, v\, \mathsf{then}\, P_1\, \mathsf{else}\, P_2|\!) = \mathsf{if}\, v\, \mathsf{then}\, (\!|\Gamma_2 \vdash P_1|\!)\, \mathsf{else}\, (\!|\Gamma_2 \vdash P_2|\!) \qquad \text{where } \Gamma_1 \vdash v\colon \mathsf{bool}$$

Figure 5: Translating mixed session processes to classical session processes

counterpart). The essence of the translation is discussed in the three examples above.

We distinguish four cases for choices, according to qualifiers (lin or un) and views ($\oplus$ or &) in types. In all of them an NDChoice process takes care of duplicated label-polarity pairs in branches. Internal choice processes feature an extra occurrence of NDChoice to non-deterministically select between output and input *on the same label*. Notice that external choice must still accept both choices, so that it is not equipped with an NDChoice. Finally, unrestricted mixed choices require the encoding of a loop, accomplished by creating a new channel for the effect (uv), installing a replicated input $\mathsf{un}\, v?\_.P$ at one end of the channel, and invoking the input once to "start" the loop and again at the end of the interaction on channel end x. The calls are all accomplished with processes of the form $u!()$. The contents of the messages are of no interest and so we use the unit value $()$.

Following the encoding for types, the encoding for external choice processes exchanges the polarities of choice labels: a label $l_i^!$ in mixed sessions is translated into $l_i^?$, and vice-versa, in the cases for lin& and un& choices. This allows reduction to happen in classical sessions, where we require an exact match between the label of the **select** process and that of the **case** process.

3 A Minimal Encoding

This section covers typing and operational correspondences; we follow Kouzapas et al. [6] criteria for typed encodings, and aim at a minimal encoding.

Let $\mathscr{C}$ range over classical processes, and $\mathscr{M}_0$ range over the fragment of mixed choice processes where lin processes only reduce against lin processes, and un processes only reduce against un processes, i.e., the reduction rules for $\mathscr{M}_0$ are those for mixed processes, except for [R-LinUn] and [R-UnLin] (Figure 6). The function $(\!|\cdot|\!) : \mathscr{M}_0 \longrightarrow \mathscr{C}$ in Figure 5 denotes a translation from mixed choice processes in $\mathscr{M}_0$ to classical processes in $\mathscr{C}$. We overload the notation and denote by $(\!|\cdot|\!)$ the encoding of both types (Figure 4) and processes (Figure 5).

We start by addressing typing criteria. The *type preservation* criterion requires that $(\!|\mathrm{op}(T_1,\ldots,T_n)|\!) = \mathrm{op}((\!|T_1|\!),\ldots,(\!|T_n|\!))$. Our encoding, in Figure 4, can be called weakly type preserving in the sense that we preserve the direction of type operations, but not the exact type operator. For example, a un$\oplus$ type is translated in a un! type (and un& type is translated in un?). Both $\oplus$ and ! can be seen as output types (and & and ? as input), so that direction is preserved.

We now move to *type soundness*, but before we need to be able to type the NDChoice operator.

Lemma 1. *The following is an admissible typing rule for typing* NDChoice.

$$\frac{\Gamma \vdash P_i \quad i \in I}{\Gamma \vdash \mathrm{NDChoice}\{P_i\}_{i\in I}}$$

Proof. The typing derivation of the expansion of NDChoice leaves open the derivations for $\Gamma \vdash P_i$. □

The type soundness theorem for our translation is item 5 below; the remaining items help in building the main result.

Theorem 2 (Type Soundness)**.**

1. *If* un T, *then* un $(\!|T|\!)$.
2. *If* un Γ, *then* un $(\!|\Gamma|\!)$.
3. *If* $S <: T$, *then* $(\!|S|\!) <: (\!|T|\!)$
4. *If* $\Gamma \vdash v\colon T$, *then* $(\!|\Gamma|\!) \vdash v\colon (\!|T|\!)$.
5. *If* $\Gamma \vdash P$, *then* $(\!|\Gamma|\!) \vdash (\!|\Gamma \vdash P|\!)$.

Proof. 1: By case analysis on T and the fact that types are contractive. 2: By induction on Γ using case 1. 3: By coinduction on the hypothesis. 4: By rule induction on the hypothesis using items 2 and 3. 5: By coinduction on the hypothesis, using items 2 and 4, and lemma 1. □

The *syntax preservation* criterion consists of ensuring that parallel composition is translated into parallel composition and that name restriction is translated into name restriction, which is certainly the case with our translation. It further requires the translation to be name invariant. Our encoding transforms

each channel end in itself and hence is trivially name invariant. We conclude that our translation is syntax preserving.

We now address the criteria related to the operational semantics. We denote by $\Rightarrow$ the reflexive and transitive closure of the reduction relations, $\rightarrow$, in both the source and target languages. Sometimes we use subscript $\mathscr{M}_0$ to denote the reduction of mixed choice processes and the subscript $\mathscr{C}$ for the reduction of classical processes, even though it should be clear from context. The behavioral equivalence $\asymp$ for classical sessions we are interested in extends structural congruence $\equiv$ with the following rule

$$(\nu ab)\prod_{i\in I} a \lhd l_i.\mathbf{0} \asymp \mathbf{0}.$$

The new rule allows collecting processes that are left by the encoding of non-deterministic choice. We call it *extended structural congruence*. The following lemma characterizes the reductions of NDChoice processes: they reduce to one of the processes that are to be chosen and leave an inert term G.

Lemma 3. $\mathrm{NDChoice}\{P_i\}_{i\in I} \rightarrow P_k \mid G \asymp P_k$, *for any* $k \in I$.

Proof. $\mathrm{NDChoice}\{P_i\}_{i\in I} \rightarrow P_k \mid G$, where $G = (\nu st)\prod_{i\in I}^{i\neq k} s \lhd l_i.\mathbf{0}$ and $G \asymp \mathbf{0}$. □

We now turn our attention to *barbs* and *barb preservation*. We say that a typed classical session process P *has a barb in* x, notation $\Gamma \vdash P \downarrow_x$, if $\Gamma \vdash P$ and

- either $P \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)(x!v.Q \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$
- or $P \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)(x \lhd l.Q \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$.

On the other hand, we say that a typed mixed session process P *has a barb in* x, notation $\Gamma \vdash P \downarrow_x$, if $\Gamma \vdash P$ and $P \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)(qx\sum_{i\in I} M_i \mid R)$ where $x \notin \{x_i, y_i\}_{i=1}^n$ and $\Gamma \vdash x\colon q \oplus \{U_i\}_{i\in I}$. Notice that only types can discover barbs in processes since internal choice is indistinguishable from external choice at the process level in $\mathscr{M}_0$.

The processes with *weak barbs* are those which reduce to a barbed process: we say that a process P *has a weak barb in* x, notation $\Gamma \vdash P \Downarrow_x$, if $P \Rightarrow P'$ and $\Gamma' \vdash P' \downarrow_x$.

The following theorem fulfills the *barb preservation criterion*: if a mixed process has a barb, its translation has a weak barb on the same channel.

Theorem 4 (Barb Preservation)**.** *The translation* $(\!|\cdot|\!) : \mathscr{M}_0 \longrightarrow \mathscr{C}$ *preserves barbs, that is, if* $\Gamma \vdash P \downarrow_x$, *then* $(\!|\Gamma|\!) \vdash (\!|\Gamma \vdash P|\!) \Downarrow_x$.

Proof. An analysis of the translations of processes with barbs. In the case that x is linear, rearranging the choice in P in fragments, we obtain that $P \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)(\mathsf{lin}\, x\sum_{i\in I}(\sum_{j\in J} l_i^! v_{ij}.P_{ij} + \sum_{k\in K} l_i^? y_{ik}.P'_{ik}) \mid R)$ and so its translation is

$$\begin{aligned}(\!|\Gamma \vdash P|\!) \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)(\mathsf{NDChoice}\{\ & x \lhd l_i^!.\mathrm{NDChoice}\, x!v_{ij}.(\!|\Gamma_3, x\colon T_i \vdash P_{ij}|\!)\}_{j\in J},\\ & x \lhd l_i^?.\mathrm{NDChoice}\, x?y_{ik}.(\!|\Gamma_2 \circ \Gamma_3, x\colon T_i', y_{ik}\colon S_i' \vdash P'_{ik}|\!)\}_{k\in K}\}_{i\in I} \mid\\ & (\!|\Gamma' \vdash R|\!)).\end{aligned}$$

This process makes internal reduction steps in the resolution of the outermost NDChoice, non-deterministically choosing one of the possible fragments, via Lemma 3. However, independently of which branch is chosen, they are all of the form $x \lhd \ell.C$, which has a barb in x. That is: $(\!|\Gamma \vdash P|\!) \Rightarrow (\nu x_n y_n)\ldots(\nu x_1 y_1)(x \lhd \ell.C \mid (\!|\Gamma' \vdash R|\!) \mid G)$, which has a barb in x. The G term is the inert remainder of the

NDChoice reduction. In the unrestricted case, we have $P \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)(\mathsf{un}\, x \sum_{i\in I}(\sum_{j\in J} l_i^! v_{ij}.P_{ij} + \sum_{k\in K} l_i^? y_{ik}.P'_{ik}) \mid R)$. The translation is

$$\begin{aligned}(\!|\Gamma \vdash P|\!) \equiv (\nu x_n y_n)\ldots(\nu x_1 y_1)((\nu uv)(u!() \mid \mathsf{un}\, v?\_.\mathsf{NDChoice}\{\\ (\nu ab)x!a.b \lhd l_i^!.\mathsf{NDChoice}\{b!v_{ij}.(u!() \mid (\!|\Gamma_1 \vdash P_{ij}|\!))\}_{j\in J},\\ (\nu ab)x!a.b \lhd l_i^?.\mathsf{NDChoice}\{b?y_{ik}.(u!() \mid (\!|\Gamma_1, y_{ik} : S'_i \vdash P'_{ik}|\!))\}_{k\in K}\}_{i\in I}) \mid (\!|\Gamma_2 \vdash R|\!)).\end{aligned}$$

The process starts by reducing via [R-UnCom] on the u, v channels to the process

$$\begin{aligned}(\!|\Gamma \vdash P|\!) \Rightarrow (\nu x_n y_n)\ldots(\nu x_1 y_1)(\nu uv)(\mathsf{NDChoice}\{\\ (\nu ab)x!a.b \lhd l_i^!.\mathsf{NDChoice}\{b!v_{ij}.(u!() \mid (\!|\Gamma_1 \vdash P_{ij}|\!))\}_{j\in J},\\ (\nu ab)x!a.b \lhd l_i^?.\mathsf{NDChoice}\{b?y_{ik}.(u!() \mid (\!|\Gamma_1, y_{ik} : S'_i \vdash P'_{ik}|\!))\}_{k\in K}\}_{i\in I} \mid (\!|\Gamma_2 \vdash R|\!) \mid U)\end{aligned}$$

where U is the persistent part of the unrestricted process. This process, in turn, reduces via the NDChoice (Lemma 3) to one of the possible branches which are all of the form $(\nu ab)x!a.C$,

$$(\!|\Gamma \vdash P|\!) \Rightarrow (\nu x_n y_n)\ldots(\nu x_1 y_1)(\nu uv)(\nu ab)(x!a.C) \mid (\!|\Gamma' \vdash R|\!) \mid U \mid G).$$

Since P has a barb in x, $x \notin \{x_i, y_i\}_{i=1}^n$ and so this process also has a barb in x, concluding that $(\!|\Gamma \vdash P|\!)$ has indeed a weak barb in x. □

Finally, we look at operational completeness. Operational completeness relates the behavior of mixed sessions against their classical sessions images: any reduction step in mixed sessions can be mimicked by a sequence of reductions steps in classical sessions, modulo extended structural congruence. The ghost reductions result from the new channels and communication inserted by the translation, namely those due to the NDChoice and to the encoding of "loops" for un mixed choices.

Theorem 5 (Reduction Completeness)**.** *The translation* $(\!|\cdot|\!) : \mathscr{M}_0 \longrightarrow \mathscr{C}$ *is* operationally complete*, that is, if* $P \to_{\mathscr{M}_0} P'$*, then* $(\!|\Gamma \vdash P|\!) \Rightarrow_{\mathscr{C}} \asymp_{\mathscr{C}} (\!|\Gamma \vdash P'|\!)$*,*

Proof. By rule induction on the derivation of $P \to_{\mathscr{M}_0} P'$. We detail two cases.

Case [R-Par]. We can show that if $Q_1 \Rightarrow_{\mathscr{C}} Q'_1$, then $Q_1 \mid Q_2 \Rightarrow_{\mathscr{C}} Q'_1 \mid Q_2$, by induction on the length of the reduction. Then we have $(\!|\Gamma \vdash P_1 \mid P_2|\!) = (\!|\Gamma_1 \vdash P_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!)$ with $\Gamma = \Gamma_1 \circ \Gamma_2$. By induction we have $(\!|\Gamma_1 \vdash P_1|\!) \Rightarrow_{\mathscr{C}} Q \asymp_{\mathscr{C}} (\!|\Gamma_1 \vdash P'_1|\!)$. Using the above result and the fact that $\asymp_{\mathscr{C}}$ is a congruence, we get $(\!|\Gamma_1 \vdash P_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!) \Rightarrow_{\mathscr{C}} Q \mid (\!|\Gamma_2 \vdash P_2|\!) \asymp_{\mathscr{C}} (\!|\Gamma_1 \vdash P'_1|\!) \mid (\!|\Gamma_2 \vdash P_2|\!) = (\!|\Gamma \vdash P'_1 \mid P_2|\!)$. The cases for [R-Res] and [R-Struct] are similar.

Case [R-LinLin]. Let $\Gamma, x\colon R, y\colon S = \Gamma' \circ \Gamma'' \circ \Gamma'''$ and $\Gamma' \vdash x\colon \mathsf{lin}\&\{l'^! T_0.R_0, \ldots\}$ and $\Gamma'' \vdash y\colon \mathsf{lin}\oplus\{l'^? U_0.S_0, \ldots\}$, with $T_0 \approx U_0$ and $R_0 \perp S_0$. Let $\Gamma' = \Gamma'_1 \circ \Gamma'_2 \circ \Gamma'_3$ and $\Gamma'' = \Gamma''_1 \circ \Gamma''_2 \circ \Gamma''_3$. We have:

$$\begin{aligned}&(\!|\Gamma \vdash (\nu xy)(\mathsf{lin}x(l'^! v.P + M) \mid \mathsf{lin}y(l'^? z.Q + N) \mid O)|\!)\\ =&(\nu xy)(x \rhd \{l'^?\colon \mathsf{NDChoice}\{x!v.(\!|\Gamma'_3, x : R_0 \vdash P|\!), \ldots\}, \ldots\} \mid\\ &\quad \mathsf{NDChoice}\{y \lhd l'^?.\mathsf{NDChoice}\{y?z.(\!|(\Gamma''_2 \circ \Gamma''_3, y : S_0, z : U_0) \vdash Q|\!), \ldots\}, \ldots\} \mid (\!|\Gamma''' \vdash O|\!))\\ \to\asymp&(\nu xy)(x \rhd \{l'^?\colon \mathsf{NDChoice}\{x!v.(\!|\Gamma'_3, x : R_0 \vdash P|\!), \ldots\}, \ldots\} \mid\\ &\quad y \lhd l'^?.\mathsf{NDChoice}\{y?z.(\!|(\Gamma''_2 \circ \Gamma''_3, y : S_0, z : U_0) \vdash Q|\!), \ldots\} \mid (\!|\Gamma''' \vdash O|\!))\\ \to&(\nu xy)(\mathsf{NDChoice}\{x!v.(\!|\Gamma'_3, x : R_0 \vdash P|\!), \ldots\} \mid\\ &\quad \mathsf{NDChoice}\{y?z.(\!|(\Gamma''_2 \circ \Gamma''_3, y : S_0, z : U_0) \vdash Q|\!), \ldots\} \mid (\!|\Gamma''' \vdash O|\!))\\ \to\to\asymp&(\nu xy)(x!v.(\!|\Gamma'_3, x : R_0 \vdash P|\!) \mid y?z.(\!|(\Gamma''_2 \circ \Gamma''_3, y : S_0, z : U_0) \vdash Q|\!) \mid (\!|\Gamma''' \vdash O|\!))\end{aligned}$$

$$\begin{aligned}
&\to(\nu xy)(\{\!|\Gamma'_3, x : R_0 \vdash P|\!\} \mid \{\!|\Gamma''_2 \circ \Gamma''_3, y : S_0, z : U_0 \vdash Q|\!\}[v/z] \mid \{\!|\Gamma''' \vdash O|\!\}) \\
&=(\nu xy)(\{\!|\Gamma'_3, x : R_0 \vdash P|\!\} \mid \{\!|\Gamma'_2 \circ \Gamma''_2 \circ \Gamma''_3, y : S_0 \vdash Q[v/z]|\!\} \mid \{\!|\Gamma''' \vdash O|\!\}) \\
&=\{\!|\Gamma'_2 \circ \Gamma'_3 \circ \Gamma''_2 \circ \Gamma''_3 \circ \Gamma''' \vdash (\nu xy)(P \mid Q[v/z] \mid O)|\!\} \\
&=\{\!|\Gamma \vdash (\nu xy)(P \mid Q[v/z] \mid O)|\!\}
\end{aligned}$$

Notice that $\Gamma'_1 = \Delta_1, x\colon R$ where Δ_1 is un, hence Δ_1 is in Γ'_2 and in Γ'_3. The same reasoning applies to Γ''_1. Since context Γ'_2 is used to type v, the substitution lemma [10] reintroduces it in the context for $Q[v/z]$.

The case for [R-UnUn] is similar, albeit more verbose. The cases for [R-IfT] and [R-IfF] are direct. □

We can show that the translation does *not* enjoy reduction soundness. Consider the classical process Q to be the encoding of process P of the form $\mathsf{un}y(m^?z.\mathbf{0})$, described in the right part of Figure 3. Soundness requires that if $Q \to_{\mathscr{C}} Q'$, then $P \Rightarrow_{\mathscr{M}_0} P'$ and $Q \Rightarrow_{\mathscr{C}} \asymp_{\mathscr{C}} \{\!|\Gamma \vdash P'|\!\}$. Clearly, Q has an initial reduction step (on channel u_2v_2), which cannot be mimicked by P. But this reduction is a transition internal to process Q, a τ transition. Equipped with a suitable notion of labelled transition systems on both languages that include τ transitions, and by using a weak bisimulation that ignores such transitions, we expect soundness to hold.

4 Further Work

There are two avenues that may be followed. One extends the encoding to the full language of mixed sessions, by taking into consideration the axioms in the reduction relation that match lin choices against un choices. The other pursues semantic preservation [6] by establishing a full abstraction result, requiring the development of typed equivalences for the two languages.

Acknowledgements This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020, and by Cost Action CA15123 EUTypes.

References

[1] Romain Demangeon & Kohei Honda (2011): *Full Abstraction in a Subtyped pi-Calculus with Linear Types*. In: *CONCUR 2011 - Concurrency Theory*, *Lecture Notes in Computer Science* 6901, Springer, pp. 280–296, doi:10.1007/978-3-642-23217-6_19.

[2] Juliana Franco & Vasco Thudichum Vasconcelos (2013): *A Concurrent Programming Language with Refined Session Types*. In: *Software Engineering and Formal Methods*, *Lecture Notes in Computer Science* 8368, Springer, pp. 15–28, doi:10.1007/978-3-319-05032-4_2.

[3] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Inf.* 42(2-3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[4] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR '93, 4th International Conference on Concurrency Theory*, *Lecture Notes in Computer Science* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[5] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *Programming Languages and Systems*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[6] Dimitrios Kouzapas, Jorge A. Pérez & Nobuko Yoshida (2019): *On the relative expressiveness of higher-order session processes*. *Inf. Comput.* 268, doi:10.1016/j.ic.2019.06.002.

[7] Robin Milner (1992): *Functions as Processes*. *Mathematical Structures in Computer Science* 2(2), pp. 119–141, doi:10.1017/S0960129500001407.

[8] Davide Sangiorgi (1998): *An Interpretation of Typed Objects into Typed pi-Calculus*. *Inf. Comput.* 143(1), pp. 34–73, doi:10.1006/inco.1998.2711.

[9] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE '94: Parallel Architectures and Languages Europe*, *Lecture Notes in Computer Science* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

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

[11] Vasco T. Vasconcelos, Filipe Casal, Bernardo Almeida & Andreia Mordido (2020): *Mixed Sessions*. In: *Programming Languages and Systems, 29th European Symposium on Programming, ESOP 2020*, *Lecture Notes in Computer Science* 12075, Springer.

[12] Vasco Thudichum Vasconcelos (1994): *Typed Concurrent Objects*. In: *Object-Oriented Programming*, *Lecture Notes in Computer Science* 821, Springer, pp. 100–117, doi:10.1007/BFb0052178.

A The Syntax, Operational Semantics, and Type System of Mixed and Classical Sessions

Mixed Sessions The syntax of process and the operational semantics are in Figure 6. The syntax of types, and the notions of subtyping and type duality are in Figure 7. The un and lin predicates, the context split and update operations, and the typing rules are in Figure 8.

Classical Sessions The syntax, operational semantics, and type system are in Figure 9.

Mixed syntactic forms

$$
\begin{array}{rll}
v & ::= & \text{Values:} \\
 & x & \text{variable} \\
 & \mathsf{true} \mid \mathsf{false} & \text{boolean values} \\
 & () & \text{unit value} \\
P & ::= & \text{Processes:} \\
 & qx\sum_{i\in I} M_i & \text{choice} \\
 & P \mid P & \text{parallel composition} \\
 & (\nu xx)P & \text{scope restriction} \\
 & \mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, P & \text{conditional} \\
 & \mathbf{0} & \text{inaction} \\
M & ::= & \text{Branches:} \\
 & l^{\star} v.P & \text{branch} \\
\star & ::= & \text{Polarities:} \\
 & ! \mid ? & \text{out and in} \\
q & ::= & \text{Qualifiers:} \\
 & \mathsf{lin} & \text{linear} \\
 & \mathsf{un} & \text{unrestricted}
\end{array}
$$

Structural congruence, $P \equiv P$

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathbf{0} \equiv P$$
$$(\nu xy)P \mid Q \equiv (\nu xy)(P \mid Q) \qquad (\nu xy)\mathbf{0} \equiv \mathbf{0} \qquad (\nu wx)(\nu yz)P \equiv (\nu yz)(\nu wx)P$$

Mixed reduction rules, $P \to P$

$$\mathsf{if\,true\,then}\, P\, \mathsf{else}\, Q \to P \qquad \mathsf{if\,false\,then}\, P\, \mathsf{else}\, Q \to Q \qquad \text{[R-IfT] [R-IfF]}$$

$$(\nu xy)(\mathsf{lin}x(l^{!}v.P + M) \mid \mathsf{lin}y(l^{?}z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid R) \qquad \text{[R-LinLin]}$$

$$(\nu xy)(\mathsf{lin}x(l^{!}v.P + M) \mid \mathsf{un}y(l^{?}z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}y(l^{?}z.Q + N) \mid R) \qquad \text{[R-LinUn]}$$

$$(\nu xy)(\mathsf{un}x(l^{!}v.P + M) \mid \mathsf{lin}y(l^{?}z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}x(l^{!}v.P + M) \mid R) \qquad \text{[R-UnLin]}$$

$$(\nu xy)(\mathsf{un}x(l^{!}v.P + M) \mid \mathsf{un}y(l^{?}z.Q + N) \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}x(l^{!}v.P + M) \mid \mathsf{un}y(l^{?}z.Q + N) \mid R) \qquad \text{[R-UnUn]}$$

$$\frac{P \to Q}{(\nu xy)P \to (\nu xy)Q} \qquad \frac{P \to Q}{P \mid R \to Q \mid R} \qquad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

[R-Res] [R-Par] [R-Struct]

Figure 6: Mixed session types: process syntax and reduction

$$
\begin{array}{rll}
T & ::= & \text{Types:} \\
 & q\sharp\{U_i\}_{i\in I} & \text{choice} \\
 & \mathsf{end} & \text{termination} \\
 & \mathsf{unit} \mid \mathsf{bool} & \text{unit and boolean} \\
 & \mu a.T & \text{recursive type} \\
 & a & \text{type variable} \\
U & ::= & \text{Branches:} \\
 & l^\star T.T & \text{branch} \\
\sharp & ::= & \text{Views:} \\
 & \oplus \mid \& & \text{internal and external} \\
\Gamma & ::= & \text{Contexts:} \\
 & \cdot & \text{empty} \\
 & \Gamma, x\colon T & \text{entry}
\end{array}
$$

The un *predicate*, $\mathsf{un}\,T$, $\mathsf{un}\,\Gamma$

$$\mathsf{un}\,(\mathsf{un}\sharp\{U_i\}_{i\in I}) \quad \mathsf{un}\,(\mu a.T) \text{ if } \mathsf{un}\,T \quad \mathsf{un}\,(\mathsf{end},\mathsf{unit},\mathsf{bool}) \qquad \mathsf{un}\,\cdot \quad \mathsf{un}\,(\Gamma, x\colon T) \text{ if } \mathsf{un}\,\Gamma \wedge \mathsf{un}\,T$$

Branch subtyping, $U <: U$

$$\frac{S_2 <: S_1 \quad T_1 <: T_2}{l^! S_1.T_1 <: l^! S_2.T_2} \qquad \frac{S_1 <: S_2 \quad T_1 <: T_2}{l^? S_1.T_1 <: l^? S_2.T_2}$$

Coinductive subtyping rules, $T <: T$

$$\frac{}{\mathsf{end} <: \mathsf{end}} \qquad \frac{}{\mathsf{unit} <: \mathsf{unit}} \qquad \frac{}{\mathsf{bool} <: \mathsf{bool}} \qquad \frac{S[\mu a.S/a] <: T}{\mu a.S <: T} \qquad \frac{S <: T[\mu a.T/a]}{S <: \mu a.T}$$

$$\frac{J \subseteq I \qquad U_j <: V_j}{q\oplus\{U_i\}_{i\in I} <: q\oplus\{V_j\}_{j\in J}} \qquad \frac{I \subseteq J \qquad U_i <: V_i}{q\&\{U_i\}_{i\in I} <: q\&\{V_j\}_{j\in J}}$$

Polarity duality and view duality, $\sharp \perp \sharp$ and $\star \perp \star$

$$!\perp ? \qquad ? \perp ! \qquad\qquad \oplus \perp \& \qquad \& \perp \oplus$$

Coinductive type duality rules, $T \perp T$

$$\frac{}{\mathsf{end} \perp \mathsf{end}} \qquad \frac{\sharp \perp \flat \qquad \star_i \perp \bullet_i \qquad S_i \approx S_i' \qquad T_i \perp T_i'}{q\sharp\{l_i^\star S_i.T_i\}_{i\in I} \perp q^\flat\{l_i^\bullet S_i'.T_i'\}_{i\in I}}$$

$$\frac{S[\mu a.S/a] \perp T}{\mu a.S \perp T} \qquad \frac{S \perp T[\mu a.T/a]}{S \perp \mu a.T}$$

Figure 7: Mixed session types: types syntax, subtyping, and duality

un *and* lin *predicates*, $\mathsf{un}(T)$, $\mathsf{lin}(T)$

$$\mathsf{un}(\mathsf{end}) \quad \mathsf{un}(\mathsf{unit}) \quad \mathsf{un}(\mathsf{bool}) \quad \mathsf{un}(\mathsf{un}\sharp\{U_i\}) \quad \frac{\mathsf{un}(T)}{\mathsf{un}(\mu a.T)} \qquad \overline{\mathsf{lin}(T)}$$

Context split, $\Gamma = \Gamma \circ \Gamma$

$$\cdot = \cdot \circ \cdot \qquad \frac{\Gamma_1 \circ \Gamma_2 = \Gamma \qquad \mathsf{un}(T)}{\Gamma, x\colon T = (\Gamma_1, x\colon T) \circ (\Gamma_2, x\colon T)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon \mathsf{lin}\, p = (\Gamma_1, x\colon \mathsf{lin}\, p) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x\colon \mathsf{lin}\, p = \Gamma_1 \circ (\Gamma_2, x\colon \mathsf{lin}\, p)}$$

Context update, $\Gamma + x\colon T = \Gamma$

$$\frac{x\colon U \notin \Gamma}{\Gamma + x\colon T = \Gamma, x\colon T} \qquad \frac{\mathsf{un}(T) \qquad T \approx U}{(\Gamma, x\colon T) + x\colon U = (\Gamma, x\colon T)}$$

Typing rules for values, $\Gamma \vdash v\colon T$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash ()\colon \mathsf{unit}} \qquad \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathsf{true}, \mathsf{false}\colon \mathsf{bool}} \qquad \frac{\mathsf{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x\colon T, \Gamma_2 \vdash x\colon T} \qquad \frac{\Gamma \vdash v\colon S \quad S <\colon T}{\Gamma \vdash v\colon T}$$

[T-Unit] [T-True] [T-False] [T-Var] [T-Subt]

Typing rules for branches, $\Gamma \vdash M\colon U$

$$\frac{\Gamma_1 \vdash v\colon S \qquad \Gamma_2 \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash l^{!}v.P\colon l^{!}S.T} \qquad \frac{\Gamma, x\colon S \vdash P}{\Gamma \vdash l^{?}x.P\colon l^{?}S.T} \qquad \text{[T-Out] [T-In]}$$

Typing rules for processes, $\Gamma \vdash P$

$$\frac{q_1(\Gamma_1 \circ \Gamma_2) \quad \Gamma_1 \vdash x\colon q_2\sharp\{l_i^{\star}S_i.T_i\}_{i\in I} \quad \Gamma_2 + x\colon T_j \vdash l_j^{\star}v_j.P_j\colon l_j^{\star}S_j.T_j \quad \{l_j^{\star}\}_{j\in J} = \{l_i^{\star}\}_{i\in I}}{\Gamma_1 \circ \Gamma_2 \vdash q_1 x \sum_{j\in J} l_j^{\star}v_j.P_j} \quad \text{[T-Choice]}$$

$$\frac{\mathsf{un}(\Gamma)}{\Gamma \vdash \mathbf{0}} \qquad \frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash P \mid Q} \qquad \frac{\Gamma_1 \vdash v\colon \mathsf{bool} \quad \Gamma_2 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 \circ \Gamma_2 \vdash \mathsf{if}\, v\, \mathsf{then}\, P\, \mathsf{else}\, Q} \qquad \frac{S \perp T \quad \Gamma, x\colon S, y\colon T \vdash P}{\Gamma \vdash (\nu xy)P}$$

[T-Inact] [T-Par] [T-If] [T-Res]

Figure 8: Mixed session types: un and lin predicates, context split and update, and typing

Syntactic forms

$$
\begin{array}{lll}
P & ::= \ldots & \text{Processes:} \\
 & \quad x!v.P & \text{output} \\
 & \quad qx?x.P & \text{input} \\
 & \quad x \lhd l.P & \text{selection} \\
 & \quad x \rhd \{l_i\colon P_i\}_{i\in I} & \text{branching} \\
T & ::= \ldots & \text{Types:} \\
 & \quad q \star T.T & \text{communication} \\
 & \quad q\sharp\{l_i\colon T_i\}_{i\in I} & \text{choice}
\end{array}
$$

Reduction rules, $P \to P$, (plus [R-RES] [R-PAR] [R-STRUCT] from Figure 6)

$$(\nu xy)(x!v.P \mid \mathsf{lin}\, y?z.Q \mid R) \to (\nu xy)(P \mid Q[v/z] \mid R) \qquad \text{[R-LINCOM]}$$

$$(\nu xy)(x!v.P \mid \mathsf{un}\, y?z.Q \mid R) \to (\nu xy)(P \mid Q[v/z] \mid \mathsf{un}\, y?z.Q \mid R) \qquad \text{[R-UNCOM]}$$

$$\frac{j \in I}{(\nu xy)(x \lhd l_j.P \mid y \rhd \{l_i\colon Q_i\}_{i\in I} \mid R) \to (\nu xy)(P \mid Q_j \mid R)} \qquad \text{[R-CASE]}$$

Subtyping rules, $T <: T$

$$\frac{T <: S \qquad S' <: T'}{q!S.S' <: q!T.T'} \qquad \frac{S <: T \qquad S' <: T'}{q?S.S' <: q?T.T'}$$

$$\frac{J \subseteq I \qquad S_j <: T_j}{q \oplus \{l_i\colon S_i\}_{i\in I} <: q \oplus \{l_j\colon T_j\}_{j\in J}} \qquad \frac{I \subseteq J \qquad S_i <: T_i}{q\&\{l_i\colon S_i\}_{i\in I} <: q\&\{l_j\colon T_j\}_{j\in J}}$$

Type duality rules, $T \perp T$

$$\frac{S <: T \qquad T <: S \qquad S' \perp T'}{q?S.S' \perp q!T.T'} \qquad \frac{S_i \perp T_i}{q \oplus \{l_i\colon S_i\}_{i\in I} \perp q\&\{l_i\colon T_i\}_{i\in I}}$$

Typing rules, $\Gamma \vdash P$, (plus [T-INACT] [T-PAR] [T-RES] from Figure 7)

$$\frac{\Gamma_1 \vdash x\colon q!T.U \qquad \Gamma_2 \vdash v\colon T \qquad \Gamma_3 + x\colon U \vdash P}{\Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!v.P} \qquad \text{[T-TOUT]}$$

$$\frac{q_1(\Gamma_1 \circ \Gamma_2) \qquad \Gamma_1 \vdash x\colon q_2?T.U \qquad (\Gamma_2 + x\colon U), y\colon T \vdash P}{\Gamma_1 \circ \Gamma_2 \vdash q_1 x?y.P} \qquad \text{[T-TIN]}$$

$$\frac{\Gamma_1 \vdash x\colon q\&\{l_i\colon T_i\}_{i\in I} \qquad \Gamma_2 + x\colon T_i \vdash P_i \qquad \forall i \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \rhd \{l_i\colon P_i\}_{i\in I}} \qquad \text{[T-BRANCH]}$$

$$\frac{\Gamma_1 \vdash x\colon q\oplus\{l_i\colon T_i\}_{i\in I} \qquad \Gamma_2 + x\colon T_j \vdash P \qquad j \in I}{\Gamma_1 \circ \Gamma_2 \vdash x \lhd l_j.P} \qquad \text{[T-SEL]}$$

Figure 9: Classical session types

Fluent Session Programming in C#

Shunsuke Kimura Keigo Imai
Gifu University, Japan
kimura@ct.info.gifu-u.ac.jp keigoi@gifu-u.ac.jp

We propose `SessionC#`, a lightweight session typed library for safe concurrent/distributed programming. The key features are (1) the improved fluent interface which enables writing communication in chained method calls, by exploiting C#'s out variables, and (2) amalgamation of session delegation with async/await, which materialises session cancellation in a limited form, which we call *session intervention*. We show the effectiveness of our proposal via a Bitcoin miner application.

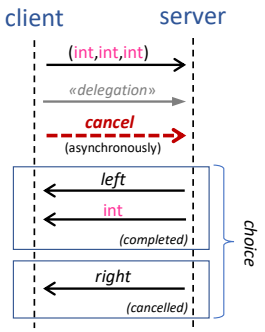
1 Introduction

Session types [8] are a theoretical framework for statically specifying and verifying communication protocols in concurrent and distributed programs. Session types guarantee that a well-typed program follows a *safe* communication protocol free from reception errors (unexpected messages) and deadlocks.

The major gaps between session types and "mainstream" programming language type system are the absence of the two key features: (1) *duality* for checking the communication protocol realises reciprocal communication actions between two peers, and (2) *linearity* ensuring that each peer is exactly following the protocol, in the way that channel variables are exclusively used from one site for the exact number of times. Various challenges have been made for incorporating them into general-purpose languages including Java [12], Scala [28], Haskell [25, 14, 19, 23], OCaml [24, 13] and Rust [15, 16].

We observe that the above-mentioned gaps in session-based programming can be *narrowed* further by the recent advancement of programming languages, which is driven by various real-world programming issues. In particular, C# [1] is widely used in areas ranging from Windows and web application platforms to gaming (e.g. Unity), and known to be eagerly adopting various language features including async/await, reifiable generics, named parameters, `out` variables and extension methods.

In this paper, we propose **`SessionC#`** – a library implementation of session types on top of the rich set of features in C#, and show its usefulness in concurrent/distributed programming, aiming for *practicality*. Namely, (1) it has an improved *fluent interface* (i.e., method calls can be chained) via C#'s `out` variables, reducing the risk of linearity violation in an *idiomatic* way. Furthermore, (2) it enables *session cancellation* in a limited form — which we call *session intervention* — by utilising amalgamation of C#'s async/await and *session delegation* in thread-based concurrency.



We illustrate the essential bits of `SessionC#` where a *cancellable* computation is guided by session types, by a use-case where a C# thread calculates a cancellable `tak` function which is designed to have a long running time [20]. The figure on the right depicts the overall communication protocol, which can be written in `SessionC#` as a ***protocol specification*** describing a client-server communication protocol from the client's viewpoint, as follows:

```
var prot = Send(Val<(int,int,int)>, Deleg(chan:Recv(Unit,End),
                                          Offer(left:Recv(Val<int>, End), right:End)));
```

S. Balzer, L. Padovani (Eds.): Programming Language Approaches
to Concurrency- & Communication-cEntric Software (PLACES 2020)
EPTCS 314, 2020, pp. 61–75, doi:10.4204/EPTCS.314.6

```
1  var cliCh = prot.ForkThread(srvCh => {
2   var srvCh2 =
3    srvCh.Receive(out int x, out int y, out int z)
4          .DelegRecv(out var cancelCh);
5   cancelCh.ReceiveAsync(out Task cancel).Close();
6   try
7   {
8    var result = Tak(x, y, z);  // compute tak
9    srvCh2.SelectLeft().Send(result).Close();
10  }
11  catch (OperationCanceledException)
12  {
13   srvCh2.SelectRight().Close(); // if cancelled
14  }
15  int Tak(int a, int b, int c) {
16   if (cancel.IsCompleted)
17    throw new OperationCanceledException();
18   return a <= b ? b :
19    Tak(Tak(a-1,b,c),Tak(b-1,c,a),Tak(c-1,a,b));
20  }});
```

Figure 1: A Cancellable `tak` [20] Implementation in `SessionC#`

From the above, C# compiler can *statically* derive both the client and the server's *session type* which is *dual* to each other, ensuring the safe interaction between the two peers. The client starts with an output (`Send`) of a triple of integer values (`Val<(int,int,int)>`) as arguments to `tak` function, which continues to a *session delegation* (`Deleg`) where a channel with an input capability (`Recv(Unit,End)`) is passed — annotated by a *named parameter* *`chan`* — from the client to the server so that the server can get notified of *cancellation*. `Offer` specifies the client offering a binary choice between *`left`* option with a reception (`Recv`) of the resulting `int` value, and *`right`* option with an immediate closing (`End`), in case `tak` is cancelled.

The `tak` server's ***endpoint implementation*** in Figure 1 enjoys *compliance* to the protocol `prot` above, by starting itself using `ForkThread` method of protocol `prot`. It runs an anonymous function communicating on a channel `srvCh` passed as an object with the exact communication API methods prescribed by the session type, which is derived from the protocol specification. Note that the channel `cliCh` returned by `ForkThread` has the session type enforcing the client as well (which will be shown by Figure 5 of § 2.2.4).

Notably, the use of ***improved fluent interface*** in Line 3-4 enhances protocol compliance, where the consecutive *input* actions (`Receive`) are realised as the *chained* method calls in a row, promoting *linear* use of the returned session channels. The `out` keywords in Line 3 are the key for this; they declare the three variables `x`, `y` and `z` *in-place*, and upon delivery of the integer triple, the received values are bound to these variables (as their references are passed to the callee). In Line 4, `DelegRecv` accepts a delegation from the client, binding it to `cancelCh`. The protocol for `cancelCh` is inferred via `var` keyword as `Recv(Unit,End)` specifying the reception of a cancellation. The continuation is then assigned to `srvCh2`.

In addition, our design of the fluent interface also takes advantage of the modern *programming environment* like Visual Studio, via ***code completion***: The code editor suggests the correct communication primitive to the programmer, guided by the session types. The screenshot in Figure 2 is such an example in Visual Studio where the two alternatives in a `Select` branch are suggested right after the symbol '.' (dot) is typed.

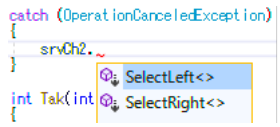


Figure 2: Completion

Furthermore, we claim that the ***session intervention pattern*** emerging from Line 5 is a novel intermix of C#'s async/await and session types in `SessionC#`, where a control flow of a session is affected by a delegated session. The delegated session can be seen as a *cancellation token* in folks, modelled by session-types. Line 5 schedules *asynchronous* reception of cancellation on `cancelCh` (`ReceiveAsync`), immediately returning from the method call (i.e., non-blocking) and binding the variable **`cancel`** to a `Task`. The task **`cancel`** gets *completed* when a `Unit` value of cancellation is delivered. The following `Close` leaves the reception incomplete. The task is checked inside `Tak` function (Line 16), raising `OperationCanceledException` if cancellation is delivered before finishing the calculation, which is caught by the outer `catch` block in Lines 11-14. Lines 8-9 in a `try` block calls the `Tak` function and sends (`Send`) the result back to the client

after selecting the left option (`SelectLeft`). If a cancellation is signalled by an exception, in `catch` block (Line 13) the server selects the right option (`SelectRight`) and closes the session.

See that all interactions in Figure 1 are deadlock-free, albeit binary session type systems like [8] and its successors used in many libraries, including ours, do not prevent deadlocks in general[1]. In the case above, operations on the delegated session `cancelCh` are non-blocking, and the session type on `srvCh` guarantees progress, provided that the client respects the *dual* session types as well, which is the case in the client in Figure 5 shown later (§ 2.2.4). Note that, however, in general, the program using *blocking* operations of `Task` may cause a deadlock.

Notes on Nondeterminism. The session intervention above is *nondeterministic*, as the client can send a cancellation *at any time* after it receives `cancelCh`. There is a possibility where the server may *disregard* the cancellation. For example, the server will *not* cancel the calculation if the client outputs on `cancelCh` *after* the server check `cancel` on Line 16, and after the result has been computed (i.e., no recursive call is made at Lines 18-19). In this case, the cancellation is still delivered to the server, and silently *ignored*. Note that the channel `cancelCh` is still used faithfully according to its protocol `Recv(Unit,End)` and *session fidelity* is maintained, as the reception is already "scheduled" by `ReceiveAsync` on Line 5. Note also that there is *no* confusion that the client consider that the calculation is cancelled, as the client must check the result of a cancellation via `Offer`, which we will revisit in § 2.2.4.

Notes on Linearity. C# does not have linear types, as in the most of mainstream languages. Thus there are two risks of linearity violations which are not checked statically: (1) use of a channel more than once and (2) channels discarded without using. For (1), we have implemented dynamic checking around it, which raises `LinearityViolationException` when a channel is used more than once. Regarding (2), although current `SessionC#` does not have capability to check it, we are planning to implement it around the *destructor* of a channel which is still not optimal, however better than nothing, as the check is delayed to the point when the garbage collector is invoked.

Notes on Session Cancellation. There are a few literature on session cancellation, such as Fowler et al.'s EGV [7], which we do not follow for now. Instead, The session intervention pattern above uses branching (`Select`/`Offer`) to handle a cancellation. There are a few issues on session cancellation in this form: (a) The cancellation handler clutters the protocol as the number of interaction increases, as mentioned in [7, § 1.2]. Although the branching-based solution is suitable for a short protocol like the above, there is a criticism by a reviewer specifically to `SessionC#` that (b) it lacks an exception handling mechanism, such as *crashing* (e.g. unhandled exceptions) and *disconnecting* (e.g. TCP connection failure). While we are yet to implement exception handling mechanisms, the distributed version of `SessionC#` equips `SessionCanceller` which handles session disconnection in terms of [7, 16].

Based on the key features and notes above, in the following sections, we explore the design space of modern session programming in `SessionC#`, showing the effectiveness of our proposal. The rest of this paper is structured as follows: In § 2, we describe the basic design of `SessionC#`, and show a few application in § 3. We conclude with remarks in § 4. Appendix § A describes implementation detail of the protocol combinators, and Appendix § B discusses more details on delegating a recursive session. Appendix § C includes more examples using `SessionC#`, including distributed implementation. The `SessionC#` is available at the following URL:

`https://github.com/curegit/session-csharp/`

[1] Exceptions are GV [30, 18] and its successors (e.g. EGV [7]), and Links language [5].

Session type	Synopsis	Combinator*	Return type*
`Send<V,S>`	Output `V` then do `S`	`Send(v,p)`$^{(1),(2)}$	`Dual<Send<V,S>,Recv<V,T>>`
`Recv<V,S>`	Input `V` then do `S`	`Recv(v,p)`$^{(1),(2)}$	`Dual<Recv<V,S>,Send<V,T>>`
`Select<SL,SR>`	Internal choice between `SL` and `SR`	`Select(left:`p_L`,right:`p_R`)`$^{(3)}$	`Dual<Select<`S_L`,`T_L`>,Offer<`S_R`,`T_R`>>`
`Offer<SL,SR>`	External choice between `SR` and `SR`	`Offer(left:`p_L`,right:`p_R`)`$^{(3)}$	`Dual<Offer<`S_L`,`T_L`>,Select<`S_R`,`T_R`>>`
`Eps`	End of the session	`End`	`Dual<Eps,Eps>`
`Goto0`	Jump to the beginning	`Goto0`	`Dual<Goto0,Goto0>`
`Deleg<S0,T0,S>`	Delegate `S0` then do `S` (where `T0` is dual of `S0`)	`Deleg(chan:`p_0`,p)`$^{(2),(3)}$	`Dual<Deleg<`S_0`,`T_0`,S>,DelegRecv<`S_0`,T>>`
`DelegRecv<S0,S>`	Accept delegation `S0` then do `S`	`DelegRecv(chan:`p_0`,p)`$^{(2),(3)}$	`Dual<DelegRecv<`S_0`,S>,Deleg<`S_0`,`T_0`,T>>`

*Note: The right half of the table assume that (1) variable `v` has type `Val<V>`, (2) variable `p` has type `Dual<S,T>`, (3) variable p_i has type `Dual<`S_i`,`T_i`>` for $i \in \{L, R, 0\}$.

Figure 3: Session Types and Protocol Combinators

2 Design of `SessionC#`

In this section, we show the design of `SessionC#` which closely follows Honda et al.'s binary session types [8]. § 2.1 introduces protocol combinators, by following Pucella and Tov's approach [25, § 5.2] with a few extensions including recursion and delegation. § 2.2 introduces the improved fluent API, taking inspiration from Scribble [2, 10] and process calculi's literature. § 2.3 discusses an encoding of mutually recursive sessions with less notational overhead.

2.1 Session Types, Protocol Combinators and Duality

Duality is the key to ensure that a pair of session types realise a safe series of interaction. Before introducing protocol combinators, we summarise session types in `SessionC#` in the left half of Figure 3. Type `Send<V,S>` and `Recv<V,S>` are output and input of value of type `V`, respectively, which continues to behave according to the session type `S`. `Select<SL,SR>` means that a process internally *decides* whether to behave according to `SL` or `SR`, by sending either label of `left` or `right`, which is called as *internal choice*. `Offer<SL,SR>` is an *external choice* where a process offers to its counterpart two possible behaviours `SL` and `SR`. `Eps` is the end of a session. `Goto0` specifies transition to the beginning of the session, which makes a limited form of *recursive session*. Later on, we extend this to mutual recursion by having more than one session types in a C# type and accessing them via an index, which is why we annotate `0` as the suffix to `Goto`. `Deleg<S0,T0,S>` is a *delegation* of session `S0` which continues to `S`, where the additional parameter `T0` is the dual of `S0`.

Note that it is possible to implement delegation *without* `Deleg` and `DelegRecv`, but with `Send` and `Recv` instead. The sole purpose of this distinction is the parameter `T0`, which is used by `DelegNew`, which we will develop later, to give the dual session type to the freshly created channel without further protocol annotation. `DelegRecv<S0,S>` is an acceptance of delegation of session `S0` which continues to `S`.

We illustrate our protocol combinators in the right half of Figure 3, making them *prove* duality of two types by restricting the constructors of `Dual<S,T>` to them having `S` and `T` to be dual to each other. In the "Combinator" column of Figure 3, the intuitive meaning of each protocol combinator can be understood

Creating a session

```
var cliCh = prot.ForkThread(srvCh => stmts)
```

Communication

Session type	Method
`Send<V,S>`	`ch.Send(v)`
`Recv<V,S>`	`ch.Receive(out V x)`, `ch.ReceiveAsync(out Task<V> task)`
`Eps`	`ch.Close()`
`Goto0`	`ch.Goto0()`
`Select<SL,SR>`	`ch.SelectLeft()`, `ch.SelectRight()`
`Offer<SL,SR>`	`ch.Offer(left:leftFunc, right:rightFunc)`
`Deleg<S0,T0,S>`	`ch.Deleg(ch2)`, `ch.DelegNew(out Session<T0,T0> ch2)`
`DelegRecv<S0,S>`	`ch.DelegRecv(out Session<S0,S0> ch2)`

Figure 4: The Communication API of `SessionC#`

as the session type in the same row of the left half specifying the *client side*'s behaviour. The "Return type" column establishes duality, by pairing each session type in the first parameter for the client with the reciprocal behaviours in the second one for the server. The type `Val<V>` is the placeholder for payload types of `Send` and `Recv`. For example, `Send(v,p)` with type `Dual<Send<V,S>,Recv<V,T>>` establishes the duality between two session types `Send<V,S>` and `Recv<V,T>` provided that `S` and `T` are dual to each other, which is ensured by the nested protocol object `p`. We defer the actual method signatures of protocol combinators to Appendix § A.

2.2 A Fluent Communication API

In Figure 4, we show the communication API of `SessionC#` which we develop in this subsection. The first column of the figure specifies the session type of the method in the second column. The fluent interface contributes to reducing the risk of linearity violation, by returning the channel with a continuation session type which increase the opportunity to chain the method call. An exception is `Offer` which takes two functions *`leftFunc`* and *`rightFunc`* taking a channel with different continuation session type for selection labels *`left`* and *`right`*, respectively.

2.2.1 Channels and Threads Maintaining Duality

The *channel type* `Session<S,E>` plays the key role in maintaining a session's evolution in the recursive type structure, where the type parameter `S` is the session type assigned to the channel, while `E` is the *session environment* of a channel which serves as a table for recursive calls (`Goto`) to look up the *next* behaviour. In other words, the `S`-part progresses when the interaction occurs on that channel, while the `E`-part *persists* (i.e., remains unchanged) during a session, maintaining the global view of a session. Thus, for example, in a method call `ch.Send(v)`, the channel `ch` must have type `Session<Send<V,S>,E>`, which returns `Session<S,E>`. We explain how the recursive structure is maintained later in § 2.3.

Based on the duality established by the protocol combinators, the `ForkThread` method ensures *safe* communication on `Session<S,E>` channels between the main thread and the forked threads. Concretely, provided `prot` has `Dual<S,T>` saying `S` and `T` are dual to each other, a method call `prot.ForkThread(ch => stmt)`

forks a new server thread, running `stmt` with channel `ch` of type `Session<T,T>`, returning the other end of channel `Session<S,S>`. The `ForkThread` is defined in the following way:

```
class Dual<S,T> {  static Session<S,S> ForkThread(Func<Session<T,T>> fun) { ... }  }
```

Note that the part `<S,S>` (and `<T,T>`) requires the beginning of a session being the same as in the session environment, maintaining the recursive structure by specifying `Goto0` going back to `S` (and `T` resp.).

2.2.2 Protocol Compliance via Extension Methods

The communication API enforces *compliance* to the type parameters in `Session<S,E>`, via *extension methods* of that type which can have additional constraints on type parameters. An extension method is the one which can be added to the existing class without modifying the existing code. For example, the following method declaration adds a method to `List<T>` class in the standard library:

```
static void AddInt(this List<int> intList, int x) { ... }
```

The `this` keyword in the first parameter specifies the method as an extension method, where the possible type of `obj` is restricted to `List<int>`. In this way, we declare the fluent API of output `ch.Send(v)`, for example, as follows:

```
static Session<S,E> Send<V,S,E>(this Session<Send<V,S>,E> ch, V v) { ... }
```

2.2.3 Binders as `out` Parameters, and Async/Await Integration

One of the central ideas of the fluent API in `SessionC#` is to exploit C#'s `out` method parameter to increase chances for method chaining. This is mainly inspired by Scribble [2, 10] implemented in Java, however, thanks to the `out` parameter in C#, there is no need to explicitly passing a *buffer* to receive an input value as in Java, keeping the session-typed program more concise and readable. `Receive` and the acceptance of delegation `DelegRecv` are implemented similarly, in the following way:

```
static Session<S,E> Receive<V,S,E>(this Session<Recv<V,S>,E> ch, out V v) { ... }
static Session<S,E> DelegRecv<S0,S,E>(this Session<DelegRecv<S0,S>,E> ch,
                                      out Session<S0,S0> ch2) { ... }
```

More interestingly, the `out` parameter in the method call `obj.Receive(out var x)` resembles *binders* in process calculi, like an input prefix $a(x).P$ in the π-calculus. By expanding this observation to name restriction $(\nu x)P$ in the π-calculus and other constructs in literature, we crystallise a few useful communication patterns of process calculi in `SessionC#`; namely (1) *bound output* and (2) *delayed input*, where the latter is implemented using async/await.

Bound output is a form of channel-passing where the freshly-created channel is passed immediately through another channel, which is written in the π-calculus as $(\nu x)\overline{a}x.P$, and $\overline{a}(x).P$ in short. As it leaves the *other end* of a channel at the sender's side, we need the *dual* of the carried (delegated) session type, which is why we have both carried type `S0` and its dual `T0` in a delegation type `Deleg<S0,T0,S>`. Thus, delegation `Deleg` and its bound-output variant `DelegNew` is defined as follows:

```
static Session<S,E> Deleg<S0,T0,S,E>(this Session<Deleg<S0,T0,S>,E> ch, Session<S0,S0> ch2) { ... }
static Session<S,E> DelegNew<S0,T0,S,E>(this Session<Deleg<S0,T0,S>,E> ch,
                                        out Session<T0,T0> ch2) { ... }
```

See that `DelegNew` declares the `out` parameter in the second one, where it binds the dual type `T0` of the delegated type `S0`.

```
1 var cliCh2 = cliCh.Send((16, 3, 2))
2                   .DelegNew(out var cancelCh);
3 Task.Delay(10000).ContinueWith(_ => {
4   cancelCh.Send().Close(); });
5 cliCh2.Offer(
6   left: cliCh3 => {
7     cliCh3.Receive(out var ans).Close();
8     Console.WriteLine("Tak(16,3,2) = " + ans);
9 }, right: cliCh3 => {
10    cliCh3.Close();
11    Console.WriteLine("Cancelled");
12 });
```

Figure 5: A `tak` Client with a Timeout

The `ReceiveAsync` is a form of delayed input in the π-calculus literature [21, § 9.3], also inspired by Scribble's *future* [9, § 13.4]. The delayed input *asynchronously* inputs a value i.e., the execution progresses without waiting for delivery of an input value, which blocks at the place it uses the input variable. This is realised by method call `ch.ReceiveAsync(out Task<V> task)` which binds a fresh *task* to variable `task` which completes when the value is delivered. We illustrate the signature of `ReceiveAsync` in the following[2]:

```
static Session<S,E> ReceiveAsync<V,S,E>(this Session<Recv<V,S>,E> ch, out Task<V> v) { ... }
```

Note that the implementation adheres the communication pattern specified in a session type, as the subsequent communication on the same channel does not take place until the preceding reception occurs.

2.2.4 A `tak` Client Example

Based on the communication API shown in this section, including `Offer` and `DelegNew`, we show an implementation of `tak` client in Figure 5, with a timeout. Line 1 sends the three arguments `(16,3,2)` to the server, and Line 2 freshly creates a channel `cancelCh` and send it to the server using bound output `DelegNew`, for later termination request. Lines 3-4 arranges an output of a termination request in 10 seconds (`10000` milliseconds). `Offer` on Line 5 makes an external choice on a channel. The *`left`* case on Lines 6-8 handles the successful completion of the calculation, where the client receives the result `ans` and print it on the screen. The *`right`* case (Lines 9-12) immediately closes the channel and prints `"Cancelled"` on the console.

As we also noted in Introduction, the cancellation request may be disregarded by the server if she has already finished the calculation. Also note that the delegated channel `cancelCh` must be used according to the linearity constraint (of which dynamic checking in `SessionC#` is yet to be implemented though), even if the client does not wish to cancel the calculation. In that case, the client can send a dummy cancellation request *after* it receives the result.

2.3 Recursive Sessions, Flatly

To handle mutually-recursive structure of a session, we extend the session environment to have more than one session type. We extend the notion of duality to the tuple of session types, and provide the protocol combinator `Arrange(p1,p2,..)`, where `p1`, `p2`, ... refers to them each other via `Goto1`, `Goto2`, ... For example, a protocol specification which alternately sends and receives an integer is written as follows:

```
var prot = Arrange(Send(Val<int>, Goto2), Recv(Val<int>, Goto1));
```

Note that the indices origin from one, to avoid confusion in a session environment with the single-cycled sessions using `Goto0`.

[2] Figure 1 uses the overloaded version where payload type `V` is fixed to `Unit`, having `Task` instead of `Task<V>` in the second argument.

The main difference from the one by Pucella and Tov [25] is that, to avoid notational overhead, we stick on *flat* tuple-based representation `(S0,S1,..)` rather than a nested cons-based list `Cons<S0,Cons<S1,..>>`. This also elides *manual* unfolding of a recursive type from $\mu\alpha.T$ to $T[\mu\alpha.T/\alpha]$ encoded as `enter` in [25], resulting in a less notational overhead in recursive session types than [25]. This ad-hoc encoding comes at a cost; the number of cycles in a recursive session is limited because the size of the tuple is limited, we must overload methods since we do not have a structural way to manipulate tuple types – although the maximum size of tuples of 8-9 seems enough for a tractable communication program. Keeping this in mind, the duality proof, `DualEnv<SS,TT>`, which states the duality between the tuple of session types `SS` and `TT`, as well as `ForkThread` and `Goto` methods for mutually recursive sessions are implemented as follows:

```
static DualEnv<(S1,S2),(T1,T2)> Arrange<S1,S2,T1,T2>(Dual<S1,T1> p1, Dual<S2,T2> p2) { ... } ...
static Session<S1,(S1,S2)> ForkThread<S1,S2,T1,T2>(this DualEnv<(S1,S2),(T1,T2)> prot,
                                                   Func<Session<T1,(T1,T2)>> fun) { ... }
static Session<S1,(S1,S2)> Goto1<S1,S2>(this Session<Goto1,(S1,S2)> ch) { ... }
static Session<S2,(S1,S2)> Goto2<S1,S2>(this Session<Goto2,(S1,S2)> ch) { ... }
```

The overloaded versions up to 8-ary is defined in similar way.

Notes on Structural Recursion in C#. A reviewer mentioned that there should be an encoding using recursive generic types in C#. For example, it would be possible to declare the following session type in C#, embodying a recursive session where a sequence of integer is received, and then the sum of them is sent back:

```
class SumSrv : Recv<int, Offer<SumSrv, Send<int, End>>> { ... }
```

Although it is possible to declare such *session types* like above, what we need is a *duality witness* (proof) encoded in C#. Consider a duality relation defined as a class, stating that `Recv<V, S>` is a dual of `Send<V, T>` if `S` is a dual of `T`:

```
class DualRecv<V, Cont> : Dual<Recv<V, ...>, Dual<Send<V, ...>>> { ... }
```

We must refer to the two components of `Cont` in the two ellipsis parts ..., which would look like the following pseudo-code:

```
// pseudo-C# code!
class DualRecv<V, Cont> : Dual<Recv<V, Cont.S>, Dual<Send<V, Cont.T>>> { ... }
```

which is not possible in C# for now. One might recall *traits* or *type members* in C++ and Scala, and *associated types* in Haskell [4]. There exists an encoding from C#'s F-bounded polymorphism to *family polymorphism* [26], at the cost of much boilerplate code. That said, the use of recursive generic types seems promising, and we are currently seeking a better design for recursive protocol combinators.

3 Application

As a more interesting application of `SessionC#`, we show a *Bitcoin miner*, where a collection of threads *iteratively* try to find a *nonce* of the specified *block*. The protocol for the Bitcoin miner is the following:

```
var prot = Select(left: Send(Val<Block>, Deleg(chan:Recv(Unit,End),
                           Offer(left:Recv(Val<uint>,Goto0), right:Goto0))),
                 right: End);
```

The endpoint implementation is in Figure 6. The `Parallel` method runs multiple threads in parallel, by passing the anonymous function a pair of the server channel `srvCh` and an extra argument `id` which is extracted from the array of parameters `ids`. The client asks (`Select`) a server thread to start the calculation

```
1  var cliChs = prot.Parallel(ids, (srvCh, id) => {
2   for (var loop = true; loop;) {
3    srvCh.Offer(left: cont => {
4     var srvCh2 = cont.Receive(out var block)
5                      .DelegRecv(out var stopCh);
6     stopCh.ReceiveAsync(out Task stop).Close();
7     var miner = new Miner(block, id);
8     while (true) {
9      if (miner.TestNextNonce(out var nonce)) {
10       srvCh = srvCh2.SelectLeft()
11                     .Send(nonce).Goto0();
12       break; // back to Offer() again
13      } else if (stop.IsCompleted) {
14       srvCh = srvCh2.SelectRight().Goto0();
15       break; // back to Offer() again
16      } else { continue; }}},
17    right: end =>
18     { end.Close(); loop = false; });}});
```

Figure 6: A Bitcoin Miner Server

by selecting `left` label, and then it sends a bitcoin `Block` and a cancellation channel in a row. Dually, after the server enters the main loop in Line 2, it offers a binary choice in Line 3, receives the block and a channel in Line 4-5, and then schedules asynchronous reception of cancellation in Line 6. After that, the server starts the calculation in Lines 7-9, entering the loop. Meanwhile, the client waits for the server (`Offer`), and if it sees `left` label, then it receives a nonce of an unsigned integer (`uint`). The corresponding behaviour in the server is found in Lines 10-12, where the server goes back to Line 3. In case another thread finds the nonce, the client asynchronously sends cancellation to the server, which is observed by the server in Lines 13-15, notifying the `right` label back to the client. In the both case, the client returns to the beginning (`Goto0`). If nonce is not found and cancellation is not asked, in Line 16, the server tries the next iteration without interacting with the client. By selecting `right` label at the top, the client can ask the server to terminate, where the server closes the session and assigns `false` to `loop` variable in Line 18, exiting the outer loop.

4 Concluding Remarks

We proposed `SessionC#`, a session-typed communication library for C#. The mainstream languages like C# has not been targeted as a platform implementing session-typed library, where one of the reasons is that the type system of the language is not suitable to implement them — they are less capable than other languages like Haskell, Scala, F# and OCaml, in the sense of having richer type inference or type-classes or implicits. Another reason would be that the type system of C# is considered quite similar to Java's one. We proclaim that the *language features* like `out` variables (and closures) also matters for establishing a safe, usable session communication pattern on top of it, including session intervention, as we have shown in the several examples in this paper.

The typestate approach taken by StMungo by Kouzapas et al. [17] equips session types on top of programming front-end Mungo. Gerbo and Padovani [17] also implements session types in typestate-based encoding via code generation using Java's annotation, enabling concurrent type-state programming a concise manner, at the cost that the protocol conformance is checked dynamically. On the other hand, type-states are sometimes *manually* maintained via variable (re)assignment in `SessionC#`, which weakens the static conformance checking. However, we hope that sticking to the library-based implementation with dynamic linearity checking competes to the aforementioned tools by providing the idiomatic usage of fluent interface.

The techniques and patterns incorporated in improved fluent interface in `SessionC#` is orthogonal to tool support, and we see opportunities to build them in combination with other proposals like Scribble, resulting in a concise multiparty session programming environment. Notably, we see that the session

intervention pattern is also effective in multiparty setting. We observe several instances of the fluent interface in Scribble family, albeit without `out` parameters, in Java [2, 10, 11], Scala [27], Go [3], and F# [22], providing *multiparty session types*. Code completion shown in the Introduction is also available in various implementation in Scribble, and most notably, the work by Neykova et al. [22] integrates Scribble with *Type Provider* in F#. SJ by Hu et al. [12] extends Java with session primitives, and also studies the protocol for session delegation in a distributed setting.

The protocol combinators are highly inspired from Pucella and Tov's encoding of duality [25]. To the author's knowledge, the addition of delegation and recursion to [25] is new. We believe the simplification of recursion adds more readability to programs using protocol-combinator based implementations. Scalas et al. [28] and Padovani [24] implements binary session types based on duality encoded in linear i/o types by Dardha et al. [6]. While it does not require any intermediate object like protocol combinators, we see the encoded session types sometimes have type readability issue, as it makes a nested, flipping sequence inside i/o type constructors, as mentioned in [13, § 6.2]. Imai et al. [13] solved this readability issues via *polarised session types*, at the cost of having polarity in types. Albeit the lack of session type inference in `SessionC#`, we also see the *explicit* approach taken by protocol combinators is not a big obstacle, as it is also the case in C# to declare method signatures explicitly.

Acknowledgements We thank the reviewers for thorough review and helpful comments for improving this paper. This work is partially supported by KAKENHI 17K12662 from JSPS, Japan, and by Grants-in-aid for Promotion of Regional Industry-University-Government Collaboration from Cabinet Office, Japan.

References

[1] *C# documentation*. `https://docs.microsoft.com/dotnet/csharp/`.

[2] *Scribble*. `http://www.scribble.org/`.

[3] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng & Nobuko Yoshida (2019): *Distributed Programming Using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures*. *Proc. ACM Program. Lang.* 3(POPL), doi:10.1145/3290342.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones & Simon Marlow (2005): *Associated types with class*. In Jens Palsberg & Martín Abadi, editors: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, ACM, pp. 1–13, doi:10.1145/1040305.1040306.

[5] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2006): *Links: Web Programming Without Tiers*. In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, pp. 266–296, doi:10.1007/978-3-540-74792-5_-12.

[6] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session types revisited*. *Inf. Comput.* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.

[7] Simon Fowler, Sam Lindley, J. Garrett Morris & Sára Decova (2019): *Exceptional Asynchronous Session Types: Session Types Without Tiers*. *Proc. ACM Program. Lang.* 3(POPL), pp. 28:1–28:29, doi:10.1145/3290341.

[8] 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.

[9] Raymond Hu (2017): *Distributed Programming Using Java APIs Generated from Session Types*. In [29, Chapter 13].

[10] Raymond Hu & Nobuko Yoshida (2016): *Hybrid Session Verification through Endpoint API Generation*. In: *19th International Conference on Fundamental Approaches to Software Engineering*, *LNCS* 9633, Springer, pp. 401–418, doi:10.1007/978-3-662-49665-7_24.

[11] Raymond Hu & Nobuko Yoshida (2017): *Explicit Connection Actions in Multiparty Session Types*. In Marieke Huisman & Julia Rubin, editors: *Fundamental Approaches to Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 116–133, doi:10.1007/978-3-662-54494-5_7.

[12] Raymond Hu, Nobuko Yoshida & Kohei Honda (2008): *Session-Based Distributed Programming in Java*. In: *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, pp. 516–541, doi:10.1007/978-3-540-70592-5_22.

[13] Keigo Imai, Nobuko Yoshida & Shoji Yuen (2018): *Session-ocaml: a Session-based Library with Polarities and Lenses*. *Sci. Comput. Program.* 172, pp. 135–159, doi:10.1016/j.scico.2018.08.005.

[14] Keigo Imai, Shoji Yuen & Kiyoshi Agusa (2010): *Session Type Inference in Haskell*. In: *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010.*, pp. 74–91, doi:10.4204/EPTCS.69.6.

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

[16] Wen Kokke (2019): *Rusty Variation: Deadlock-free Sessions with Failure in Rust*. In Massimo Bartoletti, Ludovic Henrio, Anastasia Mavridou & Alceste Scalas, editors: *Proceedings 12th Interaction and Concurrency Experience, ICE 2019, Copenhagen, Denmark, 20-21 June 2019*, *EPTCS* 304, pp. 48–60, doi:10.4204/EPTCS.304.4.

[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*, pp. 146–159, doi:10.1145/2967973.2968595.

[18] Sam Lindley & J. Garrett Morris (2015): *A Semantics for Propositions as Sessions*. In: *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*, pp. 560–584, doi:10.1007/978-3-662-46669-8_23.

[19] Sam Lindley & J. Garrett Morris (2016): *Embedding Session Types in Haskell*. In: *Haskell 2016: Proceedings of the 9th International Symposium on Haskell*, ACM, pp. 133–145, doi:10.1145/2976002.2976018.

[20] J. McCarthy (1979): *An Interesting LISP Function*. *Lisp Bull.* (3), pp. 6–8, doi:10.1145/1411829.1411833.

[21] Massimo Merro & Davide Sangiorgi (2004): *On asynchrony in name-passing calculi*. *Mathematical Structures in Computer Science* 14(5), p. 715âĂŞ767, doi:10.1017/S0960129504004323.

[22] Rumyana Neykova, Raymond Hu, Nobuko Yoshida & Fahd Abdeljallal (2018): *A session type provider: compile-time API generation of distributed protocols with refinements in F#*. In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, ACM, pp. 128–138, doi:10.1145/3178372.3179495.

[23] Dominic Orchard & Nobuko Yoshida (2016): *Effects as sessions, sessions as effects*. In: *POPL 2016: 43th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 568–581, doi:10.1145/2837614.2837634.

[24] Luca Padovani (2016): *A Simple Library Implementation of Binary Sessions*. *Journal of Functional Programming* 27, p. e4, doi:10.1017/S0956796816000289.

[25] Riccardo Pucella & Jesse A. Tov (2008): *Haskell Session Types with (Almost) No Class*. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, ACM, New York, NY, USA, pp. 25–36, doi:10.1145/1411286.1411290.

[26] Chieri Saito & Atsushi Igarashi (2008): *The Essence of Lightweight Family Polymorphism*. Journal of Object Technology 7(5), pp. 67–99, doi:10.5381/jot.2008.7.5.a3.

[27] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. In: *ECOOP*, doi:10.4230/LIPIcs.ECOOP.2017.24.

[28] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight Session Programming in Scala*. In: *ECOOP 2016: 30th European Conference on Object-Oriented Programming*, *LIPIcs* 56, Dagstuhl, pp. 21:1–21:28, doi:10.4230/LIPIcs.ECOOP.2016.21.

[29] António Ravara Simon Gay, editor (2017): *Behavioural Types: from Theory to Tools*. River Publisher, doi:10.13052/rp-9788793519817.

[30] Philip Wadler (2014): *Propositions as sessions*. *J. Funct. Program.* 24(2-3), pp. 384–418, doi:10.1017/S095679681400001X.

A Protocol Combinators

The protocol combinators are implemented as the following C# static methods and fields. The ellipsis parts are obvious return statements like "`return new Dual<Send<V,S>,Recv<V,T>>()`":

```
class ProtocolCombinators {
  static Val<V> Val<V>() { return new Val<V>(); }
  static Dual<Eps,Eps> End = new Dual<Eps,Eps>;    static Dual<Goto0,Goto0> Goto0 = new Dual<Goto0,Goto0>;
  static Dual<Send<V,T>,Recv<V,T>>  Send<V,T> (Func<Val<V>> v, Dual<V,T> cont) {...}
  static Dual<Recv<V,T>,Send<V,T>>  Recv<V,T> (Func<Val<V>> v, Dual<V,T> cont) {...}
  static Dual<Select<SL,SR>,Offer<TL,TR>>  Select<SL,SR,T0,T1> (Dual<SL,TL> contL, Dual<SR,TR> contR) {...}
  static Dual<Offer<SL,SR>,Select<TL,TR>>  Offer<SL,SR,T0,T1> (Dual<SL,TL> contL, Dual<SR,TR> contR) {...}
  static Dual<Deleg<S0,T0,S>,DelegRecv<S0,T>>  Deleg<S0,T0,S,T> (Dual<S0,T0> deleg, Dual<S,T> cont) {...}
  static Dual<DelegRecv<S0,S>,Deleg<S0,T0,T>>  DelegRecv<S0,T0,S,T> (Dual<S0,T0> deleg, Dual<S,T> cont) {...}
}
```

Modifiers such as `public` are omitted. Also, there is a small hack: The use of `Func` in the payload of `Send` and `Recv` enables omitting the parenthesis `()` after `Val<V>` in protocol specifications (as in `prot` in § 1).

B More on Recursion and Delegation

Delegation is also extended to handle with mutual recursive sessions:

```
    static Session<S,E> DelegRecv<S1,S2,S,E>(this Session<DelegRecv<(S1,S2),S>,E> ch,
                                             out Session<S1,(S1,S2)> ch2) { ... }
    static Session<S,E> Deleg<S1,T1,S2,T2,S,E>(this Session<Deleg<(S1,S2),(T1,T2),S>,E> ch,
                                               Session<S1,(S1,S2)> ch2) { ... }
    static Session<S,E> DelegNew<S1,T1,S2,T2,S,E>(this Session<Deleg<(S1,S2),(T1,T2),S>,E> ch,
                                                  out Session<T1,(T1,T2)> ch2) { ... }
```

To cope with the delegation in the *middle* of the session, we further extend the communication API for delegation, as follows:

```
    static Session<S,E> DelegRecv<S0,S1,S,E>(this Session<DelegRecv<(S0,S1),S>,E> ch,
                                             out Session<S0,S1> ch2) { ... }
    static Session<S,E> Deleg<S0,T0,S1,T1,S,E>(this Session<Deleg<(S0,S1),(T0,T1),S>,E> ch,
                                               Session<S0,S1> ch2) { ... }
    static Session<S,E> DelegNew<S0,T0,S1,T1,S,E>(this Session<Deleg<(S0,S1),(T0,T1),S>,E> ch,
                                                  out Session<T0,T1> ch2) { ... }
```

It enables a session delegated in the middle of it by having different session types in a session environment, as in `Session<S0,S1>` above.

```
1  protCA.Listen(IPAddress.Any, 8888, srvCh => {
2   using var c = new SessionCanceller();
3   c.Register(srvCh);
4   for (var loop = true; loop;) {
5    srvCh.Offer(srvQuot =>
6     quote.Receive(out var dest).Send(90.00m)
7          .Offer(srvAcpt => {
8      var cliCh = protAS.Connect("1.1.1.1", 9999);
9      c.Register(cliCh);
10     cliCh.Send(dest)
11          .Receive(out var date).Close();
12     srvAcpt.Send(date).Close();
13     loop = false;
14    }, srvReject => {
15     srvCh = srvReject.Goto();
16    }), srvQuit => {
17     srvQuit.Close();
18     loop = false; });}});
```

Figure 7: A Travel Agency (Agency Part)

```
1  // Client implementation (main thread)
2  foreach (var block in Block.GetSampleBlocks()) {
3    // Send a block to each thread
4    var ch2s = ch1s.Map(ch1 =>
5      ch1.SelectLeft().Send(block));
6    //  external choice
7    var (ch3s, cancelChs) = ch2s.Map(ch2 => {
8      var offer = ch2.DelegRecv(out var cancalCh)
9        .OfferAsync(some => {
10         var _ch3 = some.Receive(out var nonce);
11         return (_ch3.Goto(), nonce);
12       }, none => {
13         var ch3 = none.Goto();
14         return (ch3, default(uint?));
15       });
16     return (offer, cancalCh);
17   }).Unzip();
18   // Wait for any single thread to respond
19   await Task.WhenAny(ch3s);
20   // Send cancellation to each thread
21   cancelChs.ForEach(ch => ch.Send().Close());
22   // Get channels and results from future object
23   var (ch4s, results) =
24     ch3s.Select(c => c.Result).Unzip();
25   // Print results (omitted)
26   // Assign and recurse
27   ch1s = ch4s;
28 }
29 // No blocks to mine, finish channels
30 ch1s.ForEach(ch1 => ch1.SelectRight().Close());
```

Figure 8: A Bitcoin miner client

C More Examples

Figure 7 is an implementation of a Travel Agency from [12], which incorporates two sessions in a distributed setting. The *canceller* in Line 2 declared `using` modifier stops the *registered* sessions in Lines 3 and 9 when scoping out, which enables to propagate connection failure in one of underlying TCP connections to the other.

We leave a few more examples for curious readers. Figure 9 is a *parallel http downloader* from [3], which utilises `Parallel` method defined on the protocol specification object. Figure 8 is a client to Bitcoin miner shown in § 3. Figure 10 is an implementation of *parallel polygon clipping* from [25], where `Pipeline` creates a series of threads connected by two session-typed channels of which session type is described in a protocol specification.

```
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using Session;
using Session.Threading;
using static ProtocolCombinator;

public class Program {
  public static async Task Main(string[] args) {
    // Protocol specification
    var prot = Select(left: Send(Val<string>,
      Recv(Val<byte[]?>, Goto0)), right: End);

    var n = Environment.ProcessorCount;
    var ch1s = prot.Parallel(n, ch1 => {
      // Init http client
      var http = new HttpClient();

      // Work...
      for (var loop = true; loop;) {
        ch1.Offer(left => {
          var ch2 = left.Receive(out var url);
          var data = Download(url);
          ch1 = ch2.Send(data).Goto();
        }, right => {
          right.Close();
          loop = false;
        });
      }

      // Download function
      byte[]? Download(string url) {
        try {
          return http
            .GetByteArrayAsync(url).Result;
        } catch {
          return null;
        }
      }
    });

    // Pass jobs to each thread
    var (ch2s, ch1s_rest, args_rest) =
      ch1s.ZipWith(args, (ch1, arg) => {
        var ch3 = ch1.SelectLeft().Send(arg)
          .ReceiveAsync(out var data);
        return (ch3.Sync(), data);
      });

    // Close unneeded channels
    ch1s_rest
      .ForEach(c => c.SelectRight().Close());

    var (working, results) = ch2s.Unzip();
    var working_list = working.ToList();
    var result_list = results.ToList();

    // Wait for a single worker finish
    // and pass a new job
    foreach (var url in args_rest) {
      var finished =
        await Task.WhenAny(working_list);
      working_list.Remove(finished);
      var ch3 = (await finished).Goto()
        .SelectLeft().Send(url)
        .ReceiveAsync(out var data);
      working_list.Add(ch3.Sync());
      result_list.Add(data);
    }

    // Wait for still working threads
    while(working_list.Any())
    {
      var finished =
        await Task.WhenAny(working_list);
      working_list.Remove(finished);
      (await finished).Goto()
                      .SelectRight().Close();
    }

    // Save to files or something...
  }
}
```

Figure 9: Parallel HTTP Downloader [3]

```
using System;
using System.Collections.Generic;
using Session;
using Session.Threading;
using static ProtocolCombinator;

public class Program {
  public static void Main(string[] args) {
    // Input: clippee
    var vertices = new Vector[] {
      new Vector(2.0, 2.0),
      new Vector(2.0, 6.0),
      // ... and more points
    };

    // Input: clipper
    var clipper = new Vector[]
    {
      new Vector(1.0, 3.0),
      new Vector(3.0, 6.0),
      // ... and more points
    };

    // Split clipper each edges
    var edges =
      new (Vector, Vector)[clipper.Length];
    for (int i = 0; i < edges.Length; i++) {
      edges[i] = (clipper[i],
        clipper[(i + 1) % clipper.Length]);
    }

    // Protocol specification
    var prot = Select(left: Send(Val<Vector>,
      Goto0), right: End);

    var (in_ch, out_ch) = prot.Pipeline(edges,
      // Each thread
      (prev1, next1, edge) => {
        Vector? first = null;
        Vector from = default;
        Vector to = default;
        for (var loop = true; loop;) {
          prev1.Offer(left => {
            var prev2 = left
              .Receive(out var vertex);
            from = to;
            to = vertex;
            if (first == null) {
              first = to;
            } else {
              var clipped =
                Clip((from, to), edge);
              foreach (var v in clipped) {
                next1 = next1
                  .SelectLeft().Send(v).Goto();
              }
            }
            prev1 = prev2.Goto();
          }, right => {
            var clipped =
              Clip((to, first.Value), edge);
            foreach (var v in clipped) {
              next1 = next1.SelectLeft()
                           .Send(v).Goto();
            }
            next1.SelectRight();
            loop = false;
          });
        }
      }
    );

    // Main thread
    // Send vertices to pipeline
    foreach (var v in vertices) {
      in_ch = in_ch.SelectLeft().Send(v).Goto();
    }
    in_ch.SelectRight().Close();

    // Collect result from pipeline
    var result = new List<Vector>();
    for (var loop = true; loop;) {
      out_ch.Offer(left => {
        out_ch = left.Receive(out var vertex)
                     .Goto();
        result.Add(vertex);
      }, right => {
        right.Close();
        loop = false;
      });
    }

    // Print result
    for (int i = 0; i < result.Count; i++) {
      Console.WriteLine(result[i]);
    }
  }
}
```

Figure 10: Polygon Clipping Pipeline [25]