EPTCS 378

Proceedings of the

14th Workshop on

Programming Language Approaches to Concurrency and Communication-cEntric Software

Paris, France, 22 April 2023

Edited by: Ilaria Castellani and Alceste Scalas

Published: 13th April 2023
DOI: 10.4204/EPTCS.378
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii
Ilaria Castellani and Alceste Scalas

Keynote Talk: VerCors & Alpinist: Verification of Optimised GPU Programs iv
Marieke Huisman

Keynote Talk: Thirthy Years of Session Types v
Vasco T. Vasconcelos

Presentations of Preliminary or Already-Published Work vi

Kind Inference for the FreeST Programming Language 1
Bernardo Almeida, Andreia Mordido and Vasco T. Vasconcelos

A Declarative Validator for GSOS Languages 14
Matteo Cimini

A Logical Account of Subtyping for Session Types 26
Ross Horne and Luca Padovani

Communicating Actor Automata - Modelling Erlang Processes as Communicating Machines 38
Dominic Orchard, Mihail Munteanu and Paulo Torrens

Choreographic Programming of Isolated Transactions 49
Ton Smeele and Sung-Shik Jongmans

Preface

Ilaria Castellani
INRIA Sophia Antipolis Méditerranée, FR
ilaria.castellani@inria.fr

Alceste Scalas
Technical University of Denmark, DK
alcsc@dtu.dk

This volume contains the proceedings of PLACES 2023, the 14th edition of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software. The workshop is scheduled to take place in Paris on 22 April 2023, as a satellite event of ETAPS, the European Joint Conferences on Theory and Practice of Software.

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

The Programme Committee of PLACES 2023 consisted of:

- Marco Carbone, IT University of Copenhagen, DK
- Elias Castegren, Uppsala University, SE
- Silvia Crafa, Università di Padova, IT
- Francisco Ferreira, Royal Holloway, University of London, UK
- José Fragoso Santos, Universidade de Lisboa and INESC-ID, PT
- Paola Giannini, Università del Piemonte Orientale, IT
- Andrew K. Hirsch, State University of New York at Buffalo, US
- Sung-Shik Jongmans, Open University of the Netherlands, NL
- Luc Maranget, INRIA Paris, FR
- Andreia Mordido, Universidade de Lisboa and LASIGE, PT
- Violet Ka I Pun, Western Norway University of Applied Sciences, NO
- Emilio Tuosto, Gran Sasso Science Institute, IT
- Laura Voinea, University of Glasgow, UK

After a thorough reviewing process, the Programme Committee has accepted five research papers (out of seven submitted for review): such papers are published in this volume. The Programme Committee has also accepted seven talk proposal on preliminary or already-published work: the titles and abstracts of such talks are also listed in this volume (except for one, because the authors had to cancel their presentation). Each submission (research paper or talk proposal) was reviewed by three Programme Committee members and then discussed on the Easychair platform.

We would like to thank everyone who contributed to PLACES 2023: this includes the authors of submissions, the Programme Committee members, the ETAPS 2023 organisers, the Easychair and EPTCS

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop
on Programming Language Approaches to Concurrency and
Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. ii–iii, doi:10.4204/EPTCS.378.0

© Ilaria Castellani and Alceste Scalas
This work is licensed under the
Creative Commons Attribution License.

administrators. We would also like to thank Marieke Huisman and Vasco T. Vasconcelos for accepting our invitation to give a keynote talk. Finally, a special thank you goes to the Steering Committee of PLACES, consisting of Simon Gay, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida.

4 April 2023

Ilaria Castellani and Alceste Scalas

Keynote Talk:
VerCors & Alpinist: Verification of Optimised GPU Programs

Marieke Huisman
University of Twente, NL
m.huisman@utwente.nl

The VerCors verifier is a tool set for the verification of parallel and concurrent software. Its main characteristics are (i) that it can verify programs under different concurrency models, written in high-level programming languages, such as for example in Java, OpenCL and OpenMP; and (ii) that it can reason not only about race freedom and memory safety, but also about functional correctness. In this talk I will first give an overview of the VerCors verifier, and how it has been used for the verification of many different parallel and concurrent algorithms.

In the second part of my talk I will zoom in on verification of GPU programs, as they are widely used in industry. To obtain the best performance, a typical development process involves the manual or semi-automatic application of optimizations prior to compiling the code. To avoid the introduction of errors, we can augment GPU programs with (pre- and postcondition-style) annotations to capture functional properties. However, keeping these annotations correct when optimizing GPU programs is labor-intensive and error-prone.

In my talk I introduce Alpinist, an annotation-aware GPU program optimizer. It applies frequently-used GPU optimizations, but besides transforming code, it also transforms the annotations. We evaluate Alpinist, in combination with the VerCors program verifier, to automatically optimize a collection of verified programs and reverify them.

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop
on Programming Language Approaches to Concurrency and
Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. iv–iv, doi:10.4204/EPTCS.378.0.1

© Marieke Huisman
This work is licensed under the
Creative Commons Attribution License.

Keynote Talk: Thirthy Years of Session Types

Vasco T. Vasconcelos
University of Lisbon, PT
vmvasconcelos@ciencias.ulisboa.pt

1993. Kohei Honda publishes *Types for Dyadic Interaction*. In the course of five years two further papers shaped a field that was to became known as Session Types. Session types discipline interactive behaviour in the same way that functional types govern applicative behaviour. What are session types? What are they good for? What sort of applications benefit from the discipline imposed by such types? What are the challenges ahead?

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. v–v, doi:10.4204/EPTCS.378.0.2

© Vasco T. Vasconcelos
This work is licensed under the Creative Commons Attribution License.

Presentations of Preliminary and Already-Published Work

Ilaria Castellani
INRIA Sophia Antipolis Méditerranée, FR
ilaria.castellani@inria.fr

Alceste Scalas
Technical University of Denmark, DK
alcsc@dtu.dk

PLACES 2023 welcomed the submissions of talk proposals (describing preliminary or already-published work) that could spark interesting discussion during the workshop. This is the list of all accepted talk proposals.

Concurrent Symbolic Execution with Trace Semantics in Coq

Åsmund Aqissiaq Arild Kløvstad — Department of Informatics, University of Oslo, NO.

Symbolic Execution is a technique for program analysis using symbolic expressions to abstract over program state, thereby covering many program states simultaneously. Symbolic execution has been used since the mid 70's in both testing and analysis, but its formal aspects have only recently begun to be explored and unified. We present a model of symbolic execution with trace semantics in a concurrent setting in Coq, utilizing syntactic contexts to succinctly deal with parallelism.

MAGπ: Types for Failure-Prone Communication

Matthew Alan Le Brun and Ornela Dardha — University of Glasgow, UK.

This talk proposal is based on work accepted for publication at ESOP 2023. We introduce MAGπ — Multiparty, Asynchronous and Generalised π-calculus — an extension of generalised session type theory into a calculus capable of modelling non-Byzantine faults, for various physical topologies and network assumptions. Our contributions are: (1) a calculus and type-system enriched with timeouts and message loss semantics — capable of modelling the widest set of non-Byzantine faults; (2) a novel and most general definition of reliability, allowing MAGπ to model physical topologies of distributed systems; (3) a generalised theory capable of specifying assumptions of underlying network protocols; and (4) type properties that lift the benefits of generalised MPST into our realm of failure-prone communication.

Functions as Processes: The Non-Deterministic Case

Joseph Paulus — University of Groningen, NL.
Daniele Nantes-Sobrinho — Imperial College London, UK.
Jorge A. Pérez — University of Groningen, NL.

Milner's seminal work on encodings of the lambda-calculus into the pi-calculus ("functions-as-processes") explains how interaction in pi subsumes evaluation in lambda. His work opened a research strand on formal connections between sequential and concurrent calculi, covering untyped and typed regimes.

In this talk, we review a recent series of works in which we extend "functions-as-processes"; by considering calculi in which computation is non-deterministic and may lead to failures - two relevant features in programming models. On the functional side, we consider a resource lambda-calculus with non-determinism and failure, equipped with non-idempotent intersection types; on the concurrent side, we

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop
on Programming Language Approaches to Concurrency and
Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. vi–viii, doi:10.4204/EPTCS.378.0.3

© Ilaria Castellani and Alceste Scalas
This work is licensed under the
Creative Commons Attribution License.

consider a session-typed pi-calculus in which non-determinism and failure are justified by logical foundations ("propositions-as-sessions"). We have developed correct encodings of the former into the latter; they describe how typed session protocols can codify sequential evaluation in which absence/excess of resources leads to failures.

Our work reveals a new connection between two different mechanisms for enforcing resource awareness in programming calculi, namely intersection types and session types. Our talk shall elaborate on the challenges involved in connecting these different type disciplines, and also how our encodings allow us to study confluent and non-confluent forms of non-determinism in the typed setting.

Polymorphic Sessions and Sequential Composition of Types

Diogo Poças, Diana Costa, Andreia Mordido, Vasco T. Vasconcelos — LASIGE, Faculdade de Ciéncias, Universidade de Lisboa, PT.

Session types equipped with a sequential composition operator are known as context-free session types. The sequential composition operator poses new challenges not present in traditional, tail recursive types. The foremost challenge is probably deciding type equivalence. This problem has been studied in increasingly expressive systems, from first-order systems (where only base types may be exchanged), to higher-order systems; from Damas-Milner polymorphism to System F; and, more recently in the higher-order polymorphic lambda calculus. In all these systems, however, polymorphic types are of a functional nature, meaning that types cannot be exchanged on messages. We introduce polymorphic session types in a language of higher-order context free sessions and show that type equivalence is still decidable.

Language Support for Implementing Algorithms on Low Level Hardware Components

Mads Rosendahl, Maja H. Kirkeby, Mathias Larsen, Martin Sundman — Roskilde University, DK.
Tjark Petersen, Martin Schoeberl — Technical University of Denmark, DK.

Future optimizations of algorithms will include hardware implementations targeting a field-programmable gate array (FPGA). However, describing hardware in a hardware description language like VHDL or Verilog is cumbersome compared to describing an algorithm in a software language like C or Java. An alternative is to use High-level synthesis to convert programs in C into hardware design.

We explore language extensions that can assist programmers in designing algorithms for FPGA components and be integrated into existing hardware designs. The aim is to give the programmer control over the parallelism while retaining the algorithmic aspects in the development process. We compare hardware designs generated using the language extensions with designs written directly in hardware description languages.

Multiparty Session Types Meet Message Sequence Charts

Felix Stutz — Max Planck Institute for Software Systems, DE.

Implementing communication protocols is a routine task for distributed software. However, verifying that a protocol is implemented correctly in an asynchronous setting is challenging. The implementability problem asks if a (global) protocol can be implemented locally and has been studied from two perspectives. On the one hand, multiparty session types (MSTs) provide a type-theoretic approach that restricts the expressiveness of protocols. Its projection operator is a partial function that, given a protocol, attempts to compute a correct-by-construction implementation. As a best-effort technique, it is very

efficient but rejects implementable protocols. On the other hand, high-level message sequence charts (HMSCs) do not impose any restrictions on the protocols, yielding undecidability of the implementability problem for HMSCs. Consequently, model-checking can easily diverge but also suffers from high complexity. Our research aims to bridge the gap between both approaches. In this talk, we report on recent results from this endeavour. I will first visually explain classical MST projection operators and exemplify their shortcomings, showcasing sources of incompleteness for the classical MST projection approach. Then, I will elaborate on our decidability result for MST implementability. For this, we exploit our formal encoding from MSTs to HMSCs, generalise results for the latter, and prove that any implementable MST falls into a class of HMSCs with decidable implementability. Last, I will showcase techniques from the HMSC domain that become applicable in the MST setting with these results.

Kind Inference for the FreeST Programming Language

Bernardo Almeida Andreia Mordido
Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{bpdalmeida,afmordido,vmvasconcelos}@ciencias.ulisboa.pt

We present a kind inference algorithm for the FREEST programming language. The input to the algorithm is FREEST source code with (possibly part of) kind annotations replaced by kind variables. The algorithm infers concrete kinds for all kind variables. We ran the algorithm on the FREEST test suite by first replacing kind annotation on all type variables by fresh kind variables, and concluded that the algorithm correctly infers all kinds. Non surprisingly, we found out that programmers do not choose the most general kind in 20% of the cases.

1 Introduction

Software systems usually handle resources such as files and communication channels. The correct usage of such resources generally follows a protocol that describes valid patterns of interactions. For example a file should be opened and eventually closed, after which no read or write operations should ever be performed. The case for communication channels is similar: channels are opened, messages are exchanged, channels may eventually be closed, after which no more messages should be exchanged. Session types [6, 7, 15] allow expressing elaborate protocols (for files and channels, for example) guaranteeing that protocols are obeyed by programs.

FREEST [1, 2, 3] is a concurrent functional programming language based on System F where processes communicate via heterogeneously typed-channels governed by context-free session types [16]. Context-free session types allow describing protocols such as the serialization of arithmetic expressions. Consider the following datatype for arithmetic expressions.

```
1 data Exp = Lit Int | Plus Exp Exp | Times Exp Exp
```

An Exp is either a literal with an integer (Lit Int), a sum of two sub-expressions (Plus Exp Exp) or the product of two sub-expressions (Times Exp Exp). To serialise a value of type Exp we use a session type such as the following.

```
2 type ExpC = ⊕{LitC: !Int, PlusC: ExpC;ExpC, TimesC: ExpC;ExpC}
```

The abbreviation ExpC defines the type of a channel as seen from the point of view of the writer. A channel of type ExpC offers a set of options LitC, PlusC and TimesC. If the first option is chosen, an integer must be sent (!Int), while, in the others, two (sub-) expressions are expected to be sent.

Now, suppose that serialise is a function that serialises an Exp on a channel ExpC.

```
3 serialise : Exp → ExpC;a → a
```

The function expects a channel whose initial part is of type ExpC and then behaves as a: serialise is thus polymorphic on a. It consumes the front of the channel (of type ExpC) and returns the unused part of the channel (of type a).

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. 1–13, doi:10.4204/EPTCS.378.1

© B. Almeida, A. Mordido & V.T. Vasconcelos
This work is licensed under the
Creative Commons Attribution License.

As simple as it may seem, the above code is not valid in the current version of FREEST. The actual code requires further annotations allowing to distinguish functional from session types as well as linear from unrestricted types. The distinction is materialised by classifying types with kinds.

In FREEST kinds are composed of a multiplicity and a basic kind. Multiplicities control the number of times a value may be used: exactly once (linear, 1) or zero or more (unrestricted, $*$). Basic kinds distinguish functional types (T) from session types (S). The reason why FREEST requires kinds lies on polymorphism. If !**Int**;?**Int** is undoubtedly a session type and **Int** $\rightarrow$ **Bool** a functional type, the same does not apply to the polymorphic variable a. Is it a session type or a functional type? The answer depends on the base kind of a: if S or then it is a session type, if T then it is a functional type. Kinds are thus necessary to decide whether the types such as a;!**Int** are well-formed.

The datatype defined in line 1 is currently written in annotated form as follows.

```
4 data Exp:*T = Lit Int | Plus Exp Exp | Times Exp Exp
```

The kind annotation $*$T, says that the datatype is functional. As for the multiplicity, we chose the unrestricted usage so that it may be used as often as required. Notwithstanding, one may declare Exp of kind 1T, in which case serialise must become a linear function (of type Exp $\rightarrow$ ExpC;a 1$\rightarrow$ a).

Expanding the abbreviation and annotating the datatype in line 2 we get the following type.

```
5 type ExpC:1S = rec a:1S . ⊕{LitC: !Int, PlusC: a;a, TimesC: a;a}
```

ExpC defines a recursive type that is well-formed when the kind of its body, the external choice ($\oplus$), is a subkind of the kind for the recursion variable. In this case, the recursion variable ExpC is annotated with 1S, given that its body is itself a linear session.

Finally, the function serialise is currently written as follows.

```
6 serialise : ∀a:1S . Exp → ExpC;a → a
```

The polymorphic variable a stands for the continuation channel; it must be a linear session. Annotating a with the unrestricted session $*$S would dictate that it can only be instantiated with Skip, the only unrestricted session type.

Even if kinds are necessary in the underlying theory of the FREEST language, they clutter the code. The code in lines 1–3 is easier to understand and quicker to write; programmers need not fight the subtleties of each kind. Note that once kinds are inferred, the prenex occurrences of $\forall$ can be omitted. The algorithm that we present in this paper annotates all type variables with their kinds, converting the code in lines 1–3 to that in lines 4–6.

The works more closely related to FREEST are Quill [9], Affe [13], Alms [17], F° [8], $FuSe^{\{\}}$ [11] and Linear Haskell [4]. All these languages feature substructural type systems for dealing with linear, functional and affine types (in the case of Affe).

Quill [9] is a language with linear types and a syntax similar to that of Haskell. Quill features a novel design that combines linear and functional types. Contrarily to FREEST, Quill does not use kind mechanisms to distinguish between linear and functional types, instead it uses type predicates (or, qualified types) to reason about linearity. Furthermore, Quill does not support subkinding. Quill also has a type inference algorithm which was proven sound and complete. Affe [13] is an ML-like language with support to linear, affine and unrestricted types. Like Quill, Affe uses kinds and constrained types to distinguish between linear and affine types. Affe supports subkinding and it is equipped with full principal type inference. Like Affe, Alms [17] is an ML-like language but is based on System $F^{\omega}_{<:}$, the higher-order polymorphic λ-calculus with subtyping. Alms supports affine and unrestricted types. It features a rich kind system with dependent kinds, unions, and intersections. Moreover, Alms supports ML modules, allows to expose unrestricted types as affine which gives flexibility to library programmers

$$
\begin{array}{rcll}
m & ::= & * \mid \mathtt{1} \mid \varphi & \text{Multiplicity} \\
\upsilon & ::= & \mathtt{S} \mid \mathtt{T} & \text{Prekind} \\
\kappa & ::= & m\upsilon \mid \chi & \text{Kind} \\
\sharp & ::= & ! \mid ? & \text{Polarity} \\
\star & ::= & \oplus \mid \& & \text{View} \\
(\!|\cdot|\!) & ::= & \{\cdot\} \mid \langle\cdot\rangle & \text{Record} \\
T & ::= & \mathsf{Skip} \mid \mathsf{End} \mid \sharp T \mid \star(\!|\ell\colon T_\ell|\!)_{\ell\in L} \mid T;T \mid ()_m & \text{Type} \\
 & & \mid T\ m\!\rightarrow T \mid (\!|\ell\colon T_\ell|\!)_{\ell\in L} \mid \forall a^\kappa.T \mid \mu\, a^\kappa.T \mid a & \\
e & ::= & ()_m \mid x \mid \lambda_m x\colon T.e \mid \Lambda a^\kappa.v \mid e\ e \mid \{\ell{=}e_\ell\}_{\ell\in L} \mid \mathsf{let}\ \{\ell{=}x_\ell\}_{\ell\in L} = e\ \mathsf{in}\ e & \text{Expression} \\
 & & \mid \ell\ e \mid \mathsf{let}\ ()_m = e\ \mathsf{in}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L} \mid e[T] \mid \mathsf{match}\ e\ \mathsf{with}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L} &
\end{array}
$$

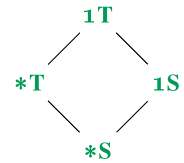
Figure 1: The syntax of kinds and types with support for kind inference

and it is equipped with local type inference. F$^\circ$ [8] is an extension of System F that uses kinds to distinguish between linear and unrestricted types. Similarly to Affe and Alms, it supports subkinding. Similarly to FreeST, but unlike Affe, F$^\circ$ does not support quantification over kinds. The work closest to FreeST in terms of context-free session types is FuSe$^{\{\}}$ [11]. Padovani proposed an alternative formulation of context-free session types in which code and types are aligned via extra annotations, something we decided to avoid in FreeST. Linear Haskell [4] is a proposal to bring linear types to Haskell. In Linear Haskell functions $T \rightarrow U$ and $T \multimap U$ describe how the arguments of the function are used. The latter form, inspired by linear logic [5], uses the argument T exactly once. In FreeST, annotated arrows $T *\!\!\rightarrow U$ or $T\ \mathtt{1}\!\!\rightarrow U$ describe how the function is used (unbounded usage or exactly once). FreeST kinding system differentiates session from functional types. It also classifies types according to their usage, linear or unrestricted. Other systems consider these notions separately (or only one of them). The ideas behind our inference algorithm are similar to Quill and Affe, but the details are quite different since we do not use type qualifiers to reason about linearity.

2 The Syntax of Kinds, Types and Expressions

This section briefly introduces the notions of kinds, types and expressions; we refer the interested reader to previous work for details [1]. FreeST relies on a base set for type variables (denoted by a, b, c) and another for labels (denoted by k, ℓ). For the purpose of kind inference, we further use *multiplicity variables* (denoted by φ) and *kind variables* (denoted by χ). The syntax of kinds, types and expressions is in fig. 1.

Multiplicities are used to indicate the number of times a value can be used. They are either unrestricted ($*$), which denotes an arbitrary number of usages, linear ($\mathtt{1}$), indicating precisely one usage, or a multiplicity variable (φ). The kinding system relies on two base kinds: $\mathtt{S}$ for session types and $\mathtt{T}$ for arbitrary types. *Kinds* are either the combination of a base kind and a multiplicity or a kind variable χ. Since a value of an unrestricted type may be used zero or more times, and one with a linear type must be used exactly once, it should be clear that an unrestricted value can used where



a linear one is expected. Similarly, the interpretation of base kinds should be such that a session type ($*\mathbf{S}$, $\mathbf{1S}$) can be used in place of an arbitrary type ($\mathbf{1T}$). The subkind relation for non variables (denoted $\kappa <: \kappa$) forms a lattice, as exhibited in the diagram.

Session types include Skip indicating no communication, End representing channels ready to be closed, output ($!T$) and input ($?T$) messages, internal ($\&\{\ell\colon T_\ell\}_{\ell\in L}$) and external choices ($\oplus\{\ell\colon T_\ell\}_{\ell\in L}$) and sequential composition ($T;U$). *Functional types* are composed of linear $()_{\mathbf{1}}$ and unrestricted unit types $()_*$, linear $T \;\mathbf{1}\!\!\rightarrow U$ and unrestricted $T \;*\!\!\rightarrow U$ functions, records $\{\ell\colon T_\ell\}_{\ell\in L}$, variants $\langle\ell\colon T_\ell\rangle_{\ell\in L}$ and universal types $\forall a^\kappa.T$. Recursive types $\mu a^\kappa.T$ are either session or functional depending on κ. Type variables a may refer to recursion variables in recursive types or to polymorphic variables in universal types. A function capturing in its body a free linear variable must itself be linear.

Expressions include variables x, term abstraction $\lambda_m x\colon T.e$ and application $e\,e$, type abstraction $\Lambda a^\kappa.v$ and application $e[T]$, record $\{\ell{=}e_\ell\}_{\ell\in L}$ and record elimination $\mathsf{let}\ \{\ell{=}x_\ell\}_{\ell\in L} = e\ \mathsf{in}\ e$, unit $()_m$ and unit elimination $\mathsf{let}\ ()_m = e\ \mathsf{in}\ e$, injection in a variant $\ell\,e$ and variant elimination $\mathsf{case}\ e\ \mathsf{of}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L}$. The expressions for channel operations include channel creation, $\mathsf{new}\ T$, and branching on a choice, $\mathsf{match}\ e\ \mathsf{with}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L}$. The remaining operations on channels—namely new, send, $\mathsf{receive}$ and $\mathsf{select}\ \ell$—are all understood as constants (pre-defined variables).

Given that our goal is to infer kind annotations, the reader may wonder why we allow them in the source language, namely in polymorphic types $\forall a^\kappa.T$, in recursive types $\mu a^\kappa.T$ and in type abstractions $\Lambda a^\kappa.v$. Programmers may, if they so wish, provide kind annotations in the source code. Such annotations are passed to the algorithm. For those omitted, a fresh kind variable χ is generated in its place.

3 Kind Inference

Our approach to kind inference follows the established two-step process, wherein the first gathers constraints and the second resolves the constraints. The constraint generation step produces constraints in two forms: $\kappa <: \kappa$ and $\varphi = \bigsqcup_{\ell\in L}\mathtt{mult}(\kappa_\ell)$. The first form represents subkinding constraints, while the second represents equalities between multiplicity variables and the least upper bound of a given set of multiplicities. To enhance readability, we use shorthand notation $\varphi = \mathtt{mult}(\kappa)$ for $\varphi = \bigsqcup\mathtt{mult}(\kappa)$ and use $\bigsqcup$ in infix format for binary sets.

Constraint Generation from Types Kind and multiplicity constraints are captured by judgement $\Delta_{\text{in}} \vdash T_{\text{in}} : \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}$. The judgement states that type T has kind κ under kinding context Δ (a map from type variables to kinds), producing constraint set $\mathscr{C}$. To clarify the distinction between input and output, we use the subscript **in** for parameters and **out** for results.

We explain a core subset of the constraint generation rules, those in fig. 2 (the complete set is in fig. 4). Rule CG-Var reads the kind for type variable a (recursive or polymorphic) from the kinding context, generating no additional restrictions. Rule CG-Rec governs recursive types which can either be session or functional. The kind of the recursion variable is copied to the kinding context when analysing type T. A constraint $\kappa' <: \kappa$ is generated to ensure that the kind κ' of the body of the recursive type is a subkind of the kind κ of the recursion variable. Rule CG-Arrow, deals with functions $T\ m\!\!\rightarrow U$. It applies the algorithm recursively to T and U, and assigns the kind $m\mathbf{T}$ to the function type, where m comes from the arrow annotation. Rule CG-Rcd builds kinds and constraints for all elements in the record. It generates a new fresh multiplicity variable φ. The result is kind $\varphi\mathbf{T}$ and the constraint set is composed of the union of $\mathscr{C}_\ell$ for all $\ell \in L$ and a new constraint associating variable φ to the least upper bound of the multiplicities of κ_ℓ. In order to ensure that φ gets the expected multiplicity, all elements

$\boxed{\Delta_{\text{in}} \vdash T_{\text{in}} : \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}}$

CG-VAR
$$\Delta, a\colon \kappa \vdash a\colon \kappa \Rightarrow \varnothing$$

CG-REC
$$\frac{\Delta, a\colon \kappa \vdash T\colon \kappa' \Rightarrow \mathscr{C}}{\Delta \vdash \mu\, a^{\kappa}.T\colon \kappa' \Rightarrow \mathscr{C} \cup \{\kappa' <: \kappa\}}$$

CG-ARROW
$$\frac{\Delta \vdash T\colon \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \vdash U\colon \kappa_2 \Rightarrow \mathscr{C}_2}{\Delta \vdash T\ m\!\rightarrow U\colon m\mathbf{T} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2}$$

CG-RCD
$$\frac{\Delta \vdash T_\ell\colon \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \varphi \text{ fresh} \qquad (\forall \ell \in L)}{\Delta \vdash \{\ell\colon T_\ell\}_{\ell \in L}\colon \varphi\mathbf{T} \Rightarrow \bigcup_{\ell \in L} \mathscr{C}_\ell \cup \{\varphi = \bigsqcup_{\ell \in L} \mathtt{mult}(\kappa_\ell), \kappa_\ell <: \varphi\mathbf{T}\}}$$

CG-TABS
$$\frac{\Delta, a\colon \kappa \vdash T\colon \kappa' \Rightarrow \mathscr{C} \qquad \varphi \text{ fresh}}{\Delta \vdash \forall a^{\kappa}.T\colon \varphi\mathbf{T} \Rightarrow \mathscr{C} \cup \{\varphi = \mathtt{mult}(\kappa')\}}$$

Figure 2: Constraint generation from types

must be subkinds of the kind of the record, that is $\varphi\mathbf{T}$. Thus, if at least one entry in the record is linear, then φ is also constrained to be linear. Rule CG-TAbs adds the kind of the polymorphic variable to the typing context when checking the body T. It then assigns kind $\varphi\mathbf{T}$ to the incoming type $\forall a^{\kappa}.T$, where the fresh multiplicity variable φ denotes the multiplicity of the kind of type T.

Type operator `mult` is fully resolved only after analysing expressions. At this point it can only be partially resolved. When applied to a kind of the form $m\upsilon$ operator `mult` rewrites into multiplicity m, that is, $\mathtt{mult}(m\upsilon) = m$.

As an example, let us consider the function that extracts the first element of a pair.

$$\mathtt{fst}\colon \forall a^{\chi_a}.\forall b^{\chi_b}.\{\mathsf{fst}\colon a, \mathsf{snd}\colon b\} \ast\!\rightarrow a$$

The application of the rules in fig. 2, yields the constraint set $\{\varphi_1 = \mathtt{mult}(\varphi_2\mathbf{T}), \varphi_2 = \mathtt{mult}(\ast\mathbf{T}), \varphi_3 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b)\}$. Solving the constraint set one obtains $\{\varphi_1 = \ast, \varphi_2 = \ast, \varphi_3 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b)\}$. We resolve the indeterminacy of kind variables χ_a and χ_b by assuming that they both are $1\mathbf{T}$, the maximum of the kind lattice. The solution would then be $\{\varphi_1 = \ast,\ \varphi_2 = \ast,\ \varphi_3 = 1,\ \chi_a = 1\mathbf{T}, \chi_b = 1\mathbf{T}\}$.

We argue that assigning $1\mathbf{T}$ (the maximum) to χ_a and χ_b is the preferred solution, since it is the less restrictive of all solutions. If we were to choose another kind, such as $\ast\mathbf{T}$, then it would be impossible to call function `fst` on linear values (of types with kind $1\mathbf{T}$). We would, undesirably, be ruling out some perfectly well-behaved programs.

But is $1\mathbf{T}$ the best kind for variables χ_a and χ_b? The answer depends on the definition of `fst`.

$$\mathtt{fst} = \Lambda a^{\chi_a}.\Lambda b^{\chi_b}.\lambda_\ast p\colon \{\mathsf{fst}\colon a, \mathsf{snd}\colon b\}.\,\mathsf{let}\ \{\mathsf{fst} = x, \mathsf{snd} = y\} = p\ \mathsf{in}\ x$$

An examination of expression $\mathsf{let}\ \{\mathsf{fst} = x, \mathsf{snd} = y\} = p\ \mathsf{in}\ x$ reveals that the second element of the pair, y, is discarded. Hence, χ_b must be unrestricted. Would $\chi_b = 1\mathbf{T}$ be chosen, then FREEST would complain about a linearity violation when type checking the function. In other words, constraint $\chi_b <: \ast\mathbf{T}$ must be added to the constraint set, but an inspection of the type of `fst` alone does not provide enough information to generate such a constraint. In the following, we present rules that allow generating constraints such as $\chi_b <: \ast\mathbf{T}$ by inspecting variable usage in expressions.

Constraint Generation from Expressions Constraints for expressions are derived from judgement $\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}}\colon T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}$. The judgement states that expression e has type T under kinding context Δ and typing context Γ. It generates a constraint set $\mathscr{C}$ and a usage context Σ. Typing contexts map term variables x to types T; usage contexts map term variables x to the kind κ of their types. Usage

$$\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}}$$

Inf-Var
$$\frac{\Delta \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma, x : T \vdash x : T \Rightarrow \mathscr{C} \mid \{x : \kappa\}}$$

Inf-Abs
$$\frac{\Delta \vdash T_1 : \kappa \Rightarrow \mathscr{C}_1 \quad \Delta \mid \Gamma, x : T_1 \vdash e : T_2 \Rightarrow \mathscr{C}_2 \mid \Sigma \quad \mathscr{C}_3 = \text{if } \mathtt{isAbs}\ e \text{ then } \{\kappa <: m\mathbf{T}\} \text{ else } \varnothing}{\Delta \mid \Gamma \vdash \lambda_m x : T_1 . e : T_1 \ m{\rightarrow}\ T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathtt{Weaken}(\Sigma, x, \kappa) \mid \Sigma \setminus x}$$

Inf-App
$$\frac{\Delta \mid \Gamma \vdash e_1 : T_1 \ m{\rightarrow}\ T_2 \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 : T_1 \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T_1 \ m{\rightarrow}\ T_2 : \kappa \Rightarrow \mathscr{C}_3}{\Delta \mid \Gamma \vdash e_1\ e_2 : T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathtt{Merge}(\Sigma_1, \Sigma_2) \mid \Sigma_1 \cup \Sigma_2}$$

Inf-RcdElim
$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 : \{\ell : T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma, (x_\ell : T_\ell)_{\ell \in L} \vdash e_2 : T \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T : \kappa \Rightarrow \mathscr{C}_3 \\ \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathscr{C}_\ell \cup \mathtt{Merge}(\Sigma_1, \Sigma_2) \cup \mathtt{Weaken}(\Sigma_2, x_\ell, \kappa_\ell) \qquad (\forall \ell \in L)\end{array}}{\Delta \mid \Gamma \vdash \text{let } \{\ell{=}x_\ell\}_{\ell \in L} = e_1 \text{ in } e_2 : T \Rightarrow \mathscr{C} \mid (\Sigma_1 \cup \Sigma_2) \setminus \{x_\ell\}_{\ell \in L}}$$

Figure 3: Constraint generation from expressions

contexts enable reasoning about variable usage: if the variable is used exactly once, it may be linear, otherwise it must be unrestricted. Next, we define functions `Weaken` and `Merge`. The former checks whether variables are used in expressions. If a variable is not used, then the set with constraint $\kappa <: *\mathbf{T}$ is returned. The latter checks whether a variable is used more than once: if it appears in multiple usage contexts, it must also be unrestricted.

$$\mathtt{Weaken}(\Sigma, x, \kappa) = \begin{cases} \varnothing & \text{if } x \in \Sigma \\ \{\kappa <: *\mathbf{T}\} & \text{otherwise} \end{cases} \qquad \mathtt{Merge}(\Sigma_1, \Sigma_2) = \{\kappa <: *\mathbf{T} \mid x : \kappa \in \Sigma_1 \cap \Sigma_2\}$$

We are now in a position to explain the rules for expressions, in fig. 3 (the complete set is in fig. 5). Rule Inf-Var is used to assign a type to a variable in a given typing context. The rule requires the type context Γ to contain an entry $x : T$. The constraints pertaining to type T are gathered in $\mathscr{C}$. To reflect the usage of x, the rule returns a singleton map $x : \kappa$, where κ is the kind of T. Rule Inf-Abs deals with abstractions $\lambda_m x : T_1 . e$. It recursively calls the judgments on T_1 and on e to collect constraint sets $\mathscr{C}_1, \mathscr{C}_2$ and usage context Σ. The rule uses a new predicate, $\mathtt{isAbs}\ e$, which holds when e is an abstraction. Then, if e is a closure the kind of T_1 must be a subkind of $m\mathbf{T}$, where m is the multiplicity of the abstraction. This restriction ensures that unrestricted abstractions do not close over linear values. The result is type $T_1\ m{\rightarrow}\ T_2$ together with a constraint set composed of the union of $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_3$ and the result of `Weaken`. The `Weaken` function checks whether a variable is unused at the end of its scope. In this case, the lambda abstraction introduces term variable x and therefore, at the end of the scope, we have to check its usage. Rule Inf-App states that if e_1 has type $T_1\ m{\rightarrow}\ T_2$ and e_2 has type T_1, then the expression $e_1\ e_2$ has type T_2. The constraints $\mathscr{C}$ and usage context Σ are computed by combining the results of the kind inference of e_1, e_2 and T. The final constraint set is the union of Σ_1, Σ_2, Σ_3, and the result of the `Merge` function which imposes that any variable found in both Σ_1 and Σ_2 must be unrestricted. The final usage context is $\Sigma_1 \cup \Sigma_2$. Rule Inf-RcdElim combines all previously discussed concepts: it evaluates expressions e_1 and e_2, collecting $\mathscr{C}_1, \mathscr{C}_2$ and Σ_1, Σ_2. The result is the type of e_2, a constraint set $\mathscr{C}$, which is the union of

$\mathscr{C}_1, \mathscr{C}_2, \mathscr{C}_3$, the result of `Merge` on Σ_1 and Σ_2, and the application of `Weaken` on Σ_2 for all $x_\ell : \kappa_\ell$ to check for unused variables. The resulting usage context is the combination of Σ_1 and Σ_2 with all entries for x_ℓ removed.

When analysing constraint generation from the type for function `fst`, we intuitively concluded that the second element in the pair must be unrestricted because it is discarded. The application of rules in fig. 3, yield the constraint set $\{\chi_b <: *\mathbf{T}, \chi_a <: \varphi_1\mathbf{T}, \chi_b <: \varphi_1\mathbf{T}, \chi_a <: \varphi_0\mathbf{T}, \chi_b <: \varphi_0\mathbf{T}, \varphi_0 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b), \varphi_1 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b)\}$. A solution for this set is $\{\varphi_0 = \mathtt{1}, \varphi_1 = \mathtt{1}, \chi_a = \mathtt{1T}, \chi_b = *\mathbf{T}\}$. The kind variable χ_b is set to $*\mathbf{T}$ as we predicted. The constraint set is computed by combining the constraint sets generated resulting from applying the judgement to all sub-expressions and the result of functions `Merge` and `Weaken`. First, we examine the `Merge` function: it takes contexts $\{p : \kappa_p\}$ and $\{x : \chi_a\}$ as input and calculates the intersection of the two contexts, adding a constraint $\kappa <: *\mathbf{T}$ for each element in the intersection. This process ensures that any variable that is used in both contexts is unrestricted. The `Weaken` function is used to verify if any newly introduced variable is eventually discarded. In our example, `Weaken` is applied to $x : \chi_a$ and $y : \chi_b$ against usage context $\{p : \kappa_p, x : \chi_a\}$. For $y : \chi_b$ function `Weaken` proceeds as follows: since y is not present in the context, a new constraint $\{\chi_b <: *\mathbf{T}\}$ is added. On the other hand, since x is already in the context, no constraint is created.

Constraint Solving We now describe an algorithm to solve constraint sets.

1. Initialise all kind variables χ to the maximum of the kind lattice, $\mathtt{1T}$. Likewise initialize all multiplicity variables φ to the maximum of multiplicities, $\mathtt{1}$. Store them in σ.
2. Iterate over each constraint in the set:
   (a) If the constraint is of the form $\chi <: \kappa$, then update the entry for χ in σ with the greatest lower bound of κ and $\sigma(\chi)$. For example, if $\sigma = [\chi \mapsto \mathtt{1T}]$ and we are analysing constraint $\chi <: *\mathbf{T}$, then the value for χ in σ must be updated to $\mathtt{1T} \sqcap *\mathbf{T} = *\mathbf{T}$. After this step, we would have $\sigma = [\chi \mapsto *\mathbf{T}]$.
   (b) If the constraint is of the form $\kappa <: \chi$, then check whether κ and the kind for χ in σ is in the subkind relation; if not then fail. For example, if $\sigma = [\chi \mapsto \mathtt{1T}]$ and we are analysing constraint $*\mathbf{T} <: \chi$, then we find that it is in the subkind relation since $*\mathbf{T} <: \mathtt{1T}$. A failure would happen with $\sigma = [\chi \mapsto \mathtt{1T}]$.
   (c) If the constraint is of the form $\kappa_1 <: \kappa_2$ and neither of the elements is a kind variable, then check whether $\kappa_1 <: \kappa_2$ is in the subkind relation; if not then fail. If not fail, then remove constraint $\kappa_1 <: \kappa_2$ from the constraint set.
   (d) If the constraint is a multiplicity constraint $\varphi = \bigsqcup_{\ell \in L} \mathtt{mult}(\kappa_\ell)$, then compute the least upper bound of the multiplicities. If any κ_ℓ is a kind variable (χ) or a base kind with a multiplicity variable ($\varphi\mathbf{T}$), we get its kind from σ (recall that all variables are in σ as per step 1). If the thus obtained kind is more restrictive than that for φ in σ (e.g. $*$ against $\sigma(\varphi) = \mathtt{1}$), then store it in σ. If $\varphi = *$, then remove the constraint from the set.
3. Repeat the process in step 2 until there are no further updates to be made.
4. If all constraints have been satisfied, then return the solution σ. Otherwise, the constraint set is unsatisfiable.

In the case of function `fst`, the constraints gathered by the rules in fig. 3 are as follows.

$$\chi_1 <: \varphi_0\mathbf{T}, \chi_0 <: \varphi_0\mathbf{T}, \chi_1 <: \varphi_1\mathbf{T}, \chi_0 <: \varphi_1\mathbf{T}, \chi_1 <: *\mathbf{T},$$
$$\varphi_0 = \mathtt{mult}(\chi_0) \sqcup \mathtt{mult}(\chi_1), \varphi_1 = \mathtt{mult}(\chi_0) \sqcup \mathtt{mult}(\chi_1)$$

Category of annotation	Number of annotations in the source code	Number of more general annotations generated
Datatypes	129	0
Type abbreviations	206	7
Universal types	282	94
Explicit recursive types	23	10
Type abstractions	30	25
Total	670	136

Table 1: Distribution of annotations

We start with $\sigma = [\chi_0 \mapsto \mathbf{1T}, \chi_1 \mapsto \mathbf{1T}, \varphi_0 \mapsto \mathbf{1}, \varphi_1 \mapsto \mathbf{1}]$. Next we pick constraint $\chi_1 <: \varphi_0\mathbf{T}$ and use item 2(a). We have, $\chi_1 <: \mathbf{1T}$ since $\sigma(\varphi_0) = \mathbf{1}$. Given that $\sigma(\chi_0)$ is equal to $\mathbf{1T}$, and subkinding is reflexive, $\sigma(\varphi_0)$ remains as $\mathbf{1T}$. The process for the second constraint, $\chi_1 <: \varphi_0\mathbf{T}$, is similar. We analyse the constraint $\chi_1 <: \mathbf{1T}$ since $\sigma(\varphi_0) = \mathbf{1}$. Also in this case item 2(a) does not change σ. The next two constraints, $\chi_1 <: \varphi_1\mathbf{T}$ and $\chi_0 <: \varphi_1\mathbf{T}$, are also handled by item 2(a). Once again, σ is subject to no update. Now we pick constraint $\chi_1 <: *\mathbf{T}$. Under item 2(a) the algorithm computes the greatest lower bound of $*\mathbf{T}$ and $\mathbf{1T}$, which is $*\mathbf{T}$, so σ is updated accordingly. For the last two constraints we use item 2(d). We read the values of χ_0 and χ_1 from σ and compute the least upper bound of $\mathtt{mult}(\mathbf{1T})$ and $\mathtt{mult}(*\mathbf{T})$ which yields $\mathbf{1}$. Both entries for χ_0 and χ_1 are already $\mathbf{1}$ and therefore no update to σ is done. Since we analysed all constraints and σ was updated in this iteration of the algorithm, the fixed-point is not reached yet and so we go through each constraint once again. This time no update is made and therefore we terminate with $\sigma = [\chi_0 \mapsto \mathbf{1T}, \chi_1 \mapsto *\mathbf{T}, \varphi_0 \mapsto \mathbf{1}, \varphi_0 \mapsto \mathbf{1}]$.

The algorithm iteratively updates the values of the kind and multiplicity variables until no further updates can be made, that is, until a fixed point is reached. Since the kind lattice is finite, any sequence of updates must eventually converge to a fixed point. For the same reason, each constraint can only be updated a finite number of times. Therefore, the algorithm terminates after a finite number of iterations.

The running time of the constraint generation algorithm is linear on the size of the input expression; that of the constraint satisfaction algorithm is quadratic. In the worst case scenario the number of constraints is equal to the size of the expression. Each constraint can only update σ twice (when a more restrictive solution is found). The worst case happens when a different constraint performs an update in each iteration, forcing the algorithm to analyse all the constraints in each iteration. A sensible optimization removes the constraints from the constraint set also in items 2(a) and 2(b), after concluding that they cannot update σ to a more restrictive solution. Since the update can only be performed a constant number of times, the algorithm becomes linear on the size of the input expression.

Evaluation We implemented the algorithm and incorporated it in the FreeST interpreter. Then we conducted an evaluation to check the behaviour of the algorithm when used on FreeST source code. The evaluation consisted of replacing all the 670 kind annotations by fresh kind variables in the 232 valid programs in the FreeST test suite and standard library (total of 9131 lines of code), running the algorithm and checking whether the algorithm infers the annotations back.

Kind annotations are spread over datatypes, type abbreviations, universal types, recursive types, and type abstractions. The distribution of annotations is as in table 1. The small number of annotations in recursive types and type abstractions comes from the fact that they are usually introduced implicitly, either via type abbreviations (as in the code in line 2) or through compiler elaboration introducing type

abstractions $\Lambda a^{\kappa}.v$ for functions accompanied by their signatures.

We concluded that the algorithm correctly inferred all annotations and found that 136 of the 670 annotations (that is, 20%) were too specific and could be relaxed to a more general kind. The largest number of more general annotations found by the algorithm come from universal types. We attribute this to the conservative nature of programmers: if we are developing Church encodings (heavy on polymorphism), why would one require linear type variables? The algorithm did not improve the kind for datatypes: datatypes are usually used in an unrestricted manner in programs. Moreover, in the test suite, they usually appear as the first argument (to be pattern-matched) of functions with unrestricted closures and therefore they cannot be linear.

For an example where the algorithm suggests a more general kind, consider function composition.

```
dot : ∀ a:*T b:*T c:*T . (b → c) → (a → b) → a → c
dot f g x = f (g x)
```

If we only provide unrestricted arguments to dot, then there is no reason why the polymorphic variables a, b and c could not have kind ∗T. However, we would be ruling out programs that apply dot to linear arguments. Consider the following program.

```
dot : (b → c) → (a → b) → a → c
dot f g x = f (g x)

g : ?Int;End → Int
g c = let (x, c) = receive c in close c ; x

main : Int
main =
  let (w,r) = new () in
  fork (\_ 1→ let w = send 5 w in close w);
  dot id g r
```

This program would be flagged as untypable because we instantiate the polymorphic variable a with the linear session type ?**Int**;**End**. Since there is no reason why a, b and c should be unrestricted, the algorithm assigns kind 1T to the three polymorphic variables.

4 Future Work

There are several avenues for future work. The most immediate is to prove the correctness of the algorithm with respect to the typing system. Then, equipped with kind inference, we may think of introducing a third base kind, that for session types that must be eventually closed (that reach type End). In this case we would require the kind of the argument to function new to be of the newly introduced kind. We further plan to study the possibility of quantifying over kinds or multiplicities for extra flexibility in programming.

Acknowledgements We thank the anonymous reviewers for their detailed comments that greatly contributed to improve the paper. This work was supported by FCT through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, and the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

References

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

[2] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST, a concurrent programming language with context-free session types*. https://freest-lang.github.io. Last accessed 2023.

[3] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST: Context-free Session Types in a Functional Language*. In: PLACES, EPTCS 291, pp. 12–23, doi:10.4204/EPTCS.291.2.

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

[5] Jean-Yves Girard (1987): *Linear Logic*. Theor. Comput. Sci. 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

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

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

[8] Karl Mazurak, Jianzhou Zhao & Steve Zdancewic (2010): *Lightweight linear types in system fdegree*. In Andrew Kennedy & Nick Benton, editors: Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010, ACM, pp. 77–88, doi:10.1145/1708016.1708027.

[9] J. Garrett Morris (2016): *The best of both worlds: linear functional programming without compromise*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016, ACM, pp. 448–461, doi:10.1145/2951913.2951925.

[10] Martin Odersky, Christoph Zenger & Matthias Zenger (2001): *Colored local type inference*. In Chris Hankin & Dave Schmidt, editors: Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001, ACM, pp. 41–53, doi:10.1145/360204.360207.

[11] Luca Padovani (2019): *Context-Free Session Type Inference*. ACM Trans. Program. Lang. Syst. 41(2), pp. 9:1–9:37, doi:10.1145/3229062.

[12] Benjamin C. Pierce & David N. Turner (2000): *Local type inference*. ACM Trans. Program. Lang. Syst. 22(1), pp. 1–44, doi:10.1145/345099.345100.

[13] Gabriel Radanne, Hannes Saffrich & Peter Thiemann (2020): *Kindly bent to free us*. Proc. ACM Program. Lang. 4(ICFP), pp. 103:1–103:29, doi:10.1145/3408985.

[14] John C. Reynolds (1974): *Towards a theory of type structure*. In Bernard J. Robinet, editor: Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974, Lecture Notes in Computer Science 19, Springer, pp. 408–423, doi:10.1007/3-540-06859-7_148.

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

[16] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-free session types*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016, ACM, pp. 462–475, doi:10.1145/2951913.2951926.

[17] Jesse A. Tov & Riccardo Pucella (2011): *Practical affine types*. In Thomas Ball & Mooly Sagiv, editors: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, ACM, pp. 447–458, doi:10.1145/1926385.1926436.

[18] J. B. Wells (1994): *Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable*. In: *LICS*, IEEE Computer Society, pp. 176–185, doi:10.1109/LICS.1994.316068.

[19] Andrew K. Wright (1995): *Simple Imperative Polymorphism*. *LISP Symb. Comput.* 8(4), pp. 343–355.

$$\boxed{\Delta_{\text{in}} \vdash T_{\text{in}} : \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}}$$

CG-UNIT
$$\Delta \vdash ()_m : m\mathbf{T} \Rightarrow \varnothing$$

CG-VAR
$$\Delta, a : \kappa \vdash a : \kappa \Rightarrow \varnothing$$

CG-SKIP
$$\Delta \vdash \mathsf{Skip} : *\mathbf{S} \Rightarrow \varnothing$$

CG-END
$$\Delta \vdash \mathsf{End} : \mathbf{1S} \Rightarrow \varnothing$$

CG-MSG
$$\frac{\Delta \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \vdash \sharp T : \mathbf{1S} \Rightarrow \mathscr{C}}$$

CG-CH
$$\frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad (\forall \ell \in L)}{\Delta \vdash \star\{\!|\ell : T_\ell|\!\}_{\ell \in L} : \mathbf{1S} \Rightarrow \bigcup_{\ell \in L} \mathscr{C}_\ell \cup \{\kappa_\ell <: \mathbf{1S}\}}$$

CG-SEQ
$$\frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \vdash U : \kappa_2 \Rightarrow \mathscr{C}_2 \qquad \varphi \text{ fresh}}{\Delta \vdash T;U : \varphi\mathbf{S} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{\kappa_1 <: \mathbf{1S}, \kappa_2 <: \mathbf{1S}, \varphi = \mathtt{mult}(\kappa_1) \sqcup \mathtt{mult}(\kappa_2)\}}$$

CG-REC
$$\frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C}}{\Delta \vdash \mu\, a^\kappa . T : \kappa' \Rightarrow \mathscr{C} \cup \{\kappa' <: \kappa\}}$$

CG-ARROW
$$\frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \vdash U : \kappa_2 \Rightarrow \mathscr{C}_2}{\Delta \vdash T \; m\!\rightarrow U : m\mathbf{T} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2}$$

CG-RCD
$$\frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \varphi \text{ fresh} \qquad (\forall \ell \in L)}{\Delta \vdash \{\ell : T_\ell\}_{\ell \in L} : \varphi\mathbf{T} \Rightarrow \bigcup_{\ell \in L} \mathscr{C}_\ell \cup \{\varphi = \bigsqcup_{\ell \in L} \mathtt{mult}(\kappa_\ell), \kappa_\ell <: \varphi\mathbf{T}\}}$$

CG-TABS
$$\frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C} \qquad \varphi \text{ fresh}}{\Delta \vdash \forall a^\kappa . T : \varphi\mathbf{T} \Rightarrow \mathscr{C} \cup \{\varphi = \mathtt{mult}(\kappa')\}}$$

Figure 4: Constraint generation from types (complete set of rules)

$$\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}}$$

INF-CONST
$$\frac{\Delta \vdash \mathtt{typeof}(c) \colon \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma \vdash c \colon \mathtt{typeof}(c) \Rightarrow \mathscr{C} \mid \varnothing}$$

INF-VAR
$$\frac{\Delta \vdash T \colon \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma, x \colon T \vdash x \colon T \Rightarrow \mathscr{C} \mid \{x \colon \kappa\}}$$

INF-ABS
$$\frac{\Delta \vdash T_1 \colon \kappa \Rightarrow \mathscr{C}_1 \qquad \Delta \mid \Gamma, x \colon T_1 \vdash e \colon T_2 \Rightarrow \mathscr{C}_2 \mid \Sigma \qquad \mathscr{C}_3 = \text{if } \mathtt{isAbs}\ e \text{ then } \{\kappa <: m\mathrm{T}\} \text{ else } \varnothing}{\Delta \mid \Gamma \vdash \lambda_m x \colon T_1 . e \colon T_1 \ m\!\!\rightarrow T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathtt{Weaken}(\Sigma, x, \kappa) \mid \Sigma \setminus \{x \colon \kappa\}}$$

INF-APP
$$\frac{\Delta \mid \Gamma \vdash e_1 \colon T_1 \ m\!\!\rightarrow T_2 \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 \colon T_1 \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T_1 \ m\!\!\rightarrow T_2 \colon \kappa \Rightarrow \mathscr{C}_3}{\Delta \mid \Gamma \vdash e_1\ e_2 \colon T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathtt{Merge}(\Sigma_1, \Sigma_2) \mid \Sigma_1 \cup \Sigma_2}$$

INF-TABS
$$\frac{\Delta, a \colon \kappa \mid \Gamma \vdash v \colon T \Rightarrow \mathscr{C}_1 \mid \Sigma \qquad \Delta \vdash T \colon \kappa' \Rightarrow \mathscr{C}_2}{\Delta \mid \Gamma \vdash \Lambda a^{\kappa} . v \colon \forall a^{\kappa} . T \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \mid \Sigma}$$

INF-TAPP
$$\frac{\Delta \vdash T \colon \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \mid \Gamma \vdash e \colon \forall a^{\kappa_2} . U \Rightarrow \mathscr{C}_2 \mid \Sigma}{\Delta \mid \Gamma \vdash e[T] \colon U[T/a] \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \mid \Sigma}$$

INF-RCDELIM
$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 \colon \{\ell \colon T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma, (x_\ell \colon T_\ell)_{\ell \in L} \vdash e_2 \colon T \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T \colon \kappa \Rightarrow \mathscr{C}_3 \\ \Delta \vdash T_\ell \colon \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathtt{Merge}(\Sigma_1, \Sigma_2) \cup \mathtt{Weaken}(\Sigma_2, x_\ell, \kappa_\ell) \qquad (\forall \ell \in L)\end{array}}{\Delta \mid \Gamma \vdash \text{let } \{\ell = x_\ell\}_{\ell \in L} = e_1 \text{ in } e_2 \colon T \Rightarrow \mathscr{C} \mid (\Sigma_1 \cup \Sigma_2) \setminus \{x_\ell \colon \kappa_\ell\}_{\ell \in L}}$$

INF-RCD
$$\frac{\Delta \mid \Gamma \vdash e_\ell \colon T_\ell \Rightarrow \mathscr{C}_\ell \mid \Sigma_\ell \qquad \Delta \vdash T_\ell \colon \kappa_\ell \Rightarrow \mathscr{C}'_\ell \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \{\ell = v_\ell\}_{\ell \in L} \colon \{\ell \colon T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_\ell \cup \mathscr{C}'_\ell \cup \mathtt{Merge}(\Sigma_\ell) \mid \bigcup_{\ell \in L} \Sigma_\ell}$$

INF-VARIANT
$$\frac{\Delta \mid \Gamma \vdash e \colon T_k \Rightarrow \mathscr{C}_1 \mid \Sigma \qquad \Delta \vdash T_\ell \colon \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad k \in L \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash k\ e \colon \langle \ell \colon T_\ell \rangle_{\ell \in L} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_\ell \mid \Sigma}$$

INF-CASE
$$\frac{\Delta \mid \Gamma \vdash e \colon \langle \ell \colon T_\ell \rangle_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_\ell \colon T_\ell \ m\!\!\rightarrow T \Rightarrow \mathscr{C}_\ell \mid \Sigma_\ell \qquad \Delta \vdash T_\ell \colon \kappa_\ell \Rightarrow \mathscr{C}'_\ell \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \text{case } e \text{ of } \{\ell \rightarrow x_\ell\}_{\ell \in L} \colon T \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_\ell \cup \mathscr{C}'_\ell \mid \Sigma_1 \cup \Sigma_\ell}$$

INF-SEL
$$\frac{\Delta \vdash T_\ell \colon \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \Delta \mid \Gamma \vdash e_\ell \colon T_\ell \ m\!\!\rightarrow T \Rightarrow \mathscr{C}'_\ell \mid \Sigma_\ell \qquad k \in L \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \text{select } k \colon \oplus (\!| \ell \colon T_\ell |\!)_{\ell \in L} \ m\!\!\rightarrow T_k \Rightarrow \mathscr{C}_\ell \cup \mathscr{C}'_\ell \mid \bigcup_{\ell \in L} \Sigma_\ell}$$

INF-NEW
$$\frac{\varnothing \vdash T \colon \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma \vdash \text{new } T \colon \{\text{fst} \colon T, \text{snd} \colon \overline{T}\} \Rightarrow \mathscr{C} \mid \emptyset}$$

Figure 5: Constraint generation from expressions (complete set of rules)

A Declarative Validator for GSOS Languages

Matteo Cimini
University of Massachusetts Lowell
Lowell, MA, USA
matteo_cimini@uml.edu

Rule formats can quickly establish meta-theoretic properties of process algebras. It is then desirable to identify domain-specific languages (DSLs) that can easily express rule formats. In prior work, we have developed LANG-N-CHANGE, a DSL that includes convenient features for browsing language definitions and retrieving information from them. In this paper, we use LANG-N-CHANGE to write a validator for the GSOS rule format, and we augment LANG-N-CHANGE with suitable macros on our way to do so. Our GSOS validator is concise, and amounts to a few lines of code. We have used it to validate several concurrency operators as adhering to the GSOS format. Moreover, our code expresses the restrictions of the format declaratively.

1 Introduction

After creating a process algebra, the job of a language designer is not finished yet. Ideally, the language designer would strive to prove that the process algebra at hand affords the desired properties. Depending on the process algebra, it may be interesting to establish whether bisimilarity is a congruence, whether the language is deterministic, or whether some of its operators satisfy certain algebraic laws such as commutativity and associativity, to name a few.

The field of *language validation* aims at developing methods and tools that take a language definition as input, apply a static analysis on it, and establish whether some property holds for the language. In the context of concurrency theory, language validation has been best expressed with *rule formats* [34] over Structural Operational Semantics (SOS) specifications [36]. Rule formats state that if the rules that have been used to write the SOS specification of a language conform to some syntactic restrictions then some semantic property is guaranteed to hold. This approach has been applied to automatically derive the congruence of strong bisimilarity [12, 25, 26, 40], of weak bisimilarity [11, 21, 23], and to establish algebraic laws of operators [4, 5, 18, 35], as well as deriving global properties such as determinism [1] and bounded nondeterminism [22], and has also been applied to probabilistic transitions [9, 19, 28] and contexts with binders [6, 20, 41], to name a few applications.

It is then desirable to identify suitable domain-specific languages (DSLs) that make it easier for designers of rule formats to express their formats and automatically test them. This allows them to quickly test their new ideas, do so on a suite of several process algebras at once, and have a path to rapidly prototyping new formats. These tests are helpful for debugging a new rule format while designers are still crafting a theoretical result.

Unfortunately, literature does not offer any DSL that has been specifically designed for expressing rule formats. In this paper, we focus on an existing DSL called LANG-N-CHANGE [30, 32], which has been created for purposes other than language validation but whose operations can be repurposed to write rule formats. LANG-N-CHANGE is a DSL for expressing language transformations, that is, the input is a language definition and transformation instructions, and the output is a modification of the language

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop
on Programming Language Approaches to Concurrency and
Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. 14–25, doi:10.4204/EPTCS.378.2

© M. Cimini
This work is licensed under the
Creative Commons Attribution License.

given as input. Consider the typing rule of function application below on the left and its version with subtyping on the right.

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2}\ (\text{T-APP}) \quad \Longrightarrow \quad \frac{\Gamma \vdash e_1 : T_{11} \to T_2 \qquad \Gamma \vdash e_2 : T_{12} \qquad T_{12} <: T_{11}}{\Gamma \vdash e_1\, e_2 : T_2}\ (\text{T-APP'})$$

(T-APP') is a transformation of (T-APP). LANG-N-CHANGE provides linguistic features to express such a transformation. For example, LANG-N-CHANGE can express the test that detects that T_1 has been used twice in (T-APP) and assigns new distinct variables T_{11} and T_{12} to those occurrences. LANG-N-CHANGE also includes the operation for creating the premise $T_{12} <: T_{11}$. LANG-N-CHANGE has been applied to several case studies that automatically transform SOS specifications, such as adding subtyping [16, 30], pattern-matching [32], references [32], gradual typing [31], as well as automatically deriving big-step semantics from small-step style [30], and CK machines [16].

To express language transformations, LANG-N-CHANGE offers fine-grained operations for browsing SOS rules, their premises, and other components of the language. In this paper, we explore the idea of using LANG-N-CHANGE to express rule formats. The idea is to devise a language transformation that ultimately *does not* modify the language in input, but that uses the operations of LANG-N-CHANGE to test the syntactic constraints prescribed by the rule formats, and throw a runtime error when they are not met. We have written a checker for the GSOS format [12] using LANG-N-CHANGE. This is a well-known rule format, which can establish the congruence of bisimilarity of many process algebras with common operators. To better express some of the tests that are prescribed by this format, we have augmented LANG-N-CHANGE with suitable macros. These are not extensions to the core language, nor to its evaluator. These are parsed away into ordinary operations of LANG-N-CHANGE.

We have used our tool to validate common concurrency operators that are known to adhere to the GSOS format, such as the CCS parallel operator [29], the synchronous parallel composition from CSP [27], and the projection operator of ACP [10]. (Section 4 provides a complete list of our tests.) In total, we have validated 18 concurrency operators. We have also performed a series of negative tests. More specifically, we have given to our tool languages as input that do not adhere to the GSOS format, and we confirm that our tool rejects them, indeed.

Our GSOS validator amounts to 6 lines of code, which makes for a very concise validator. Also, our code expresses the GSOS syntactic restrictions declaratively. The work in this paper provides some evidence that LANG-N-CHANGE can be a useful tool for expressing rule formats.

The paper is organized as follows. Section 2 provides an overview of LANG-N-CHANGE. Section 3 presents our new macros and our GSOS validator. Section 4 discusses our evaluation. Section 5 discusses related work, and Section 6 concludes the paper.

2 Overview on LANG-N-CHANGE

We repeat the relevant background on LANG-N-CHANGE [30, 32] in this section. Fig. 1 shows the tool pipeline of LANG-N-CHANGE. The input consists of two elements: A language definition and a language transformation. The output is either a language definition, or an error message.

What Language Definitions? LANG-N-CHANGE works with languages defined in SOS. The input is a textual representation of transition system specifications for SOS (with negative transitions) [13]. The

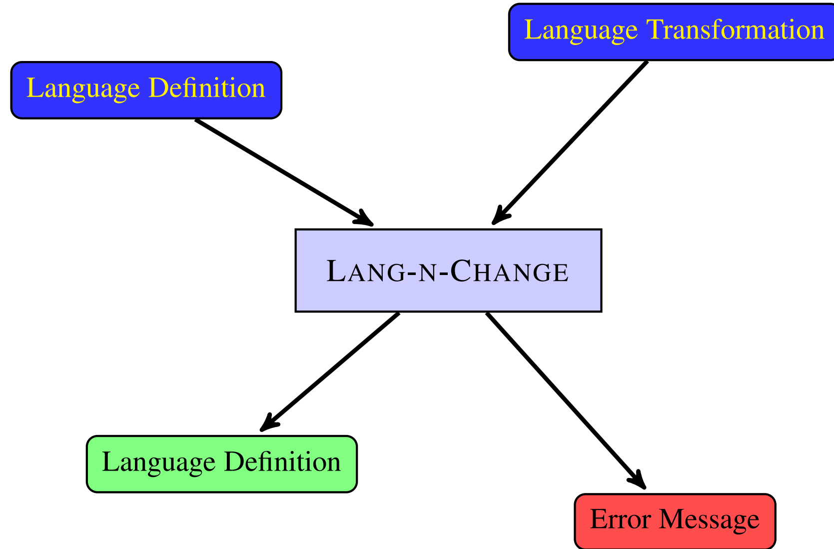


Figure 1: Tool pipeline of LANG-N-CHANGE

following is an example language that is input to LANG-N-CHANGE: A process algebra with the prefix operator, the interleaving operator, and the sequence operator.

```
Label L ::= (a) | (b) | (c)
Process P ::= (null) | (a P) | (b P) | (c P) | (par P P) | (sequence P P)

(a P) --(a)--> P.

(par P1 P2) --(a)--> (par P1' P2) <== P1 --(a)--> P1'.
(par P1 P2) --(a)--> (par P1 P2') <== P2 --(a)--> P2'.

(sequence P1 P2) --(a)--> (sequence P1' P2) <== P1 --(a)--> P1'.
(sequence P1 P2) --(a)--> P2' <== P2 --(a)--> P2' /\ P1 -/-(a)-->
                                                  /\ P1 -/-(b)-->
                                                  /\ P1 -/-(c)-->.

 ...  rest of the rules (same as the rules above but for the other labels)
```

That is, a grammar declares processes and labels, and a series of inference rules define labeled transitions. Intuitively, <== means "provided that", and the formulae after that symbol are the premises of the rule. A formula such as "`P2 -/-(a)-->`" means that P_2 does not perform an a-transition. This syntax does not present any novelty compared to other textual representations for SOS, and is indeed inspired by the syntax employed in the Ott tool [38].

Some remarks on our example language: The GSOS format only works with a finite set of labels [12]. For simplicity, we have chosen the set of actions $\{a,b,c\}$. Moreover, the uniform setting for operators in GSOS is that of a function symbol applied to processes. This is typically accommodated as shown above: A prefix operator for each action.

What Language Transformations? The following is the subset of the syntax of LANG-N-CHANGE that is relevant to this paper[1].

$$\begin{array}{llll}
\textsf{Expression} & e & ::= & x \mid str \mid t \mid [e\ \ldots\ e] \mid \texttt{head}\, e \mid \texttt{tail}\, e \mid e@e \mid e - e \mid \texttt{map}(e,e) \mid e(e) \\
& & & \mid \texttt{rules} \mid \texttt{premises} \mid \texttt{conclusion} \mid \texttt{self} \\
& & & \mid e[p] :\ e \mid \texttt{uniquefy}(e) \Rightarrow (x,x) : e \mid \texttt{getVars}(e) \\
& & & \mid \texttt{if}\, b\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{skip} \mid \texttt{error}\, str \\
\textsf{Boolean Expr.} & b & ::= & e = e \mid \texttt{isVar}(e) \mid b\, \texttt{and}\, b \mid b\, \texttt{or}\, b \mid \texttt{not}\, b \\
\textsf{Pattern} & p & ::= & x \mid [p\ \ldots\ p] \mid predname\ p \mid opname\ p \mid x\ p
\end{array}$$

We assume a set of operator names OPNAME ranged over by *opname*. OPNAME contains elements such as `par`, and `sequence`, for example. We also assume a set of predicate names PREDNAME ranged over by *predname*. PREDNAME contains elements such as $\longrightarrow$ and $\not\longrightarrow$. LANG-N-CHANGE accommodates formulae uniformly in abstract syntax $(predname\ arg_1\ \ldots\ arg_n)$, as it does not make assumptions on the language. Yet, the tool still reads transitions such as $P \longrightarrow^{a} P'$ with syntax `P --(a)--> P'` for the convenience of users (see the example language above).

Expression is the main syntactic category. Given an SOS specification, i.e., a language $\mathscr{L}$, and given an expression e, e contains the operations that will be applied to $\mathscr{L}$. Expressions can be variables, strings (str), terms (t) (such as `(par P1 P2)` and `(sequence P1 P2)`), and lists with ordinary operations for extracting the head and the tail of lists, appending two lists (e @ e), and performing list difference ($e - e$). Expressions can also be maps $\texttt{map}(e_1, e_2)$, where e_1 and e_2 are lists. The first element of e_1 is the key of the first element of e_2, and so on for the rest of the elements[2]. Given a map m, $m(k)$ retrieves the value in m associated with the key k.

LANG-N-CHANGE includes the special keywords `rules`, `premises`, `conclusion`, and `self`. The keyword `rules` returns a list with all the inference rules of the language given as input. We shall describe the other keywords in the context of the following operator.

The *selector operator* $e_1[p] :\ e_2$ selects one by one the elements of the list e_1 that satisfy the pattern p and executes the body e_2 for each of them. The selector returns a list with the values produced by each evaluation of e_2. The keyword `self`, when used in e_2, returns the element of the list e_1 that has been selected at that iteration. A *pattern* p can be a variable, can attempt to match a list (pattern $[p\ \ldots\ p]$), to match a formula that uses a specific predicate name (pattern *predname p*), to match a term with a specific top-level operator (pattern *opname p*), or can attempt to match a formula or term with an unspecified top-level name (pattern *x p*). As typical with pattern-matching, the variables that are used in the pattern p are bound in e_2, and are instantiated at runtime. To make an example, let us consider $\texttt{rules}[P \longrightarrow^{L} P'] :\ e_2$ being executed for the example language above (with prefix, `par`, and `sequence`). `rules` evaluates to a list with all the rules of the language. When the list contains rules, the pattern of the selector is applied to match the conclusions of these rules. Therefore, the pattern $P \longrightarrow^{L} P'$ selects all the reduction rules. The first iteration of e_2 is executed with $P =$ `(a P)`, $L =$ `(a)`, and $P' =$ `P`, and so on. (Notice that there is no clash between pattern variables and the metavariables of rules, as they are separate in LANG-N-CHANGE.) For the convenience of programmers, simply writing $\texttt{rules}[\longrightarrow] :\ e$ selects the rules that define $\longrightarrow$ without specifying a full pattern. Also, $e[p]$ is a shorthand for $e[p] : \texttt{self}$, i.e., a list of the elements selected by the pattern. Therefore, $\texttt{rules}[\longrightarrow]$ simply selects all reduction rules.

[1] We refer the reader to [30] and [32] for the syntax of LANG-N-CHANGE.

[2] This schema is motivated in [30]. For example, it quickly maps T_i to T_i' from conclusions of subtyping rules such as the conclusions $T_1 \to T_2 <: T_1' \to T_2'$ and $T_1 \times T_2 <: T_1' \times T_2'$.

The keyword `premises` can be used when the selector operator works on rules, and returns the list of premises of the selected rule. For example, `rules`$[\longrightarrow]$: `premises` returns a list where each element is the list of premises of a rule such as [[], [`P1 --(a)--> P1'`], [`P2 --(a)--> P2'`], ...] in our example process algebra. Similarly, the keyword `conclusion` returns the conclusion of the selected rule.

`uniquefy`$(e_1) \Rightarrow (x,y) : e_2$ takes a list e_1 of formulae, and returns a version of these formulae where multiple occurrences of a metavariable have been assigned distinct metavariables. The computation continues by executing e_2. The list of new formulae is passed to e_2 as x. `uniquefy` also computes a map that summarizes the changes that have been made to the original list of formulae e_1. This map is passed to e_2 as y. This operation is useful for transformations such as that of (T-APP) into (T-APP') that we have described in Section 1. Suppose that l contains the list of premises of (T-APP), then `uniquefy`$(l) \Rightarrow (\mathit{newPremises}, \mathit{mapOfChanges}) : e$ will execute e where *newPremises* is the list $[\Gamma \vdash e_1 : T_{11} \to T_2, \Gamma \vdash e_2 : T_{12}]$ and *mapOfChanges* is the map $\{T_1 \mapsto [T_{11}, T_{12}]\}$, which denotes that the occurrences of T_1 have been split into T_{11} and T_{12}[3].

`getVars`(e) returns the list of metavariables that are used in e after it has been evaluated. We also have an if-statement, and a `skip` operation that does not perform any operation. When `if` has no `else` branch, as in `if` b `then` e, it means `if` b `then` e `else skip`. Error `error` throws a runtime error and carries a string as error message. The boolean conditions of the if-statement can check for syntactic equality, whether e is a metavariable with `isVar`(e), and can combine these checks with boolean operations.

We do not discuss here the type checker of LANG-N-CHANGE, which has been presented in [30] and can reject, for example, $e[p] : e'$ when e is not a list, and other type errors.

3 A GSOS Validator

The GSOS Rule Format We recall the GSOS format [12]. The following is the shape for GSOS rules:

$$\frac{\{x_i \longrightarrow^{l_{ij}} y_{ij} \mid i \in I, 1 \leq j \leq m_i\} \cup \{x_j \not\longrightarrow^{l'_{jk}} \mid j \in J, 1 \leq k \leq n_j\}}{(op\; x_1 \ldots x_h) \longrightarrow^{l} t}$$

Notice that xs and ys are metavariables for metavariables, so that some relation can be stated among different metavariables. In other words, xs and ys all denote metavariables such as P, P_1, P_2, and so on. We have that x_i and y_i are all distinct. I and J are subsets of $\{1, \ldots, h\}$, that is, xs in the premises come from the conclusion, and each of them can be the subject of positive premises multiple times, as well as the subject of negative premises multiple times. The metavariables that occur in t can only come from xs and ys, Finally, labels ls are constants.

A rule that conforms to these restrictions is a GSOS rule. For example, all the rules of the example process algebra in the previous section (with prefix, `par`, and `sequence`) are GSOS rules.

Consider the following rule, which defines the behavior of the replication operator.

$$\frac{(P \mid !P) \longrightarrow^{a} P'}{!P \longrightarrow^{a} P'}$$

This rule is not a GSOS rule because the source of the premise is $(P \mid !P)$ rather than a variable.

The following is a classic result of the meta-theory of SOS: If all the rules of the language are GSOS rules then bisimilarity is a congruence for the language [12].

[3] `uniquefy` can be used in a more fine-grained style, as shown in [30], but we do not need that style in this paper.

3.1 New Macros for LANG-N-CHANGE

We define the following macros in LANG-N-CHANGE.

- e `must match` $p_1 \mid p_2 \mid \ldots \mid p_n$ `otherwise` $e' \triangleq$ `if not`$((e - e[p_1] - e[p_2] \ldots - e[p_n]) = [])$ `then` e'.
  Here, e is a list. The idea is that each element of e must match one of the patterns $p_1, p_2, \ldots, p_n$, otherwise we execute e'. To do that, we progressively subtract from e its sublists filtered by the patterns and check that the resulting list is empty. This macro is useful to check that premises and conclusions have the correct shape. We use this same "empty list"-test to check whether a list is a sublist of another with: e `sublistOf` $e' \triangleq (e - e') = []$. This macro is useful to check that the metavariables being used in some part of the rule all come from the correct list of metavariables.
- `match` e `with` $p \to e'$ `otherwise` $e'' \triangleq$ `if` $[e][p] = []$ `then` e'' `else` e'.

  Here, we check that e matches the pattern p and, if that is the case, we execute e', otherwise we execute e''. To do so, we create the list with only one element $[e]$ and use the selector to filter it by pattern p. If the resulting list is empty then the pattern p does not succeed for e[4]. When we omit "$\to e'$" in this macro, it means "$\to$ `skip`". When we omit "`otherwise` e''", it means "`otherwise skip`". This macro is useful to check a pattern for one element, as opposed to a list as above, and to specify a `then`- versus `otherwise`-reaction.
- The following macros are useful to quickly access sources and targets of transition formulae:

$$
\begin{aligned}
\texttt{premises.LTsources} &\triangleq (\texttt{premises}[P \texttt{ --}L\texttt{--> } P'] : P) \ @ \ (\texttt{premises}[P \texttt{ -/-}L\texttt{-->}] : P)\\
\texttt{premises.LTtargets} &\triangleq \texttt{premises}[P \texttt{ --}L\texttt{--> } P'] : P'\\
\texttt{conclusion.LTsource} &\triangleq \texttt{head}\ ([\texttt{conclusion}][P \texttt{ --}L\texttt{--> } P'] : P)\\
\texttt{conclusion.LTtarget} &\triangleq \texttt{head}\ ([\texttt{conclusion}][P \texttt{ --}L\texttt{--> } P'] : P')
\end{aligned}
$$

  Notice that `premises.LTsources` extracts the sources of both positive and negative labeled transition formulae. "LT" in `LTsources` stands for labeled transition. We also have introduced the analogous macros for (unlabeled) transitions $P \longrightarrow P$ such as `Tsources`, `Ttargets`, and so on. We do not think that these are ad-hoc macros in the context of language design. Labeled and (unlabeled) transitions are so common that it is reasonable to have operations that say, for example, "*handle this premise as a labeled transition formula and return its source*". When a formula of another shape is given, `premises.LTsources` and `premises.LTtargets` return an empty list, and `head` fails at runtime for `conclusion.LTsource` and `conclusion.LTtarget`.
- `distinctVars`(e) `otherwise` $e' \triangleq$

  `uniquefy`$([(\mathit{pname}\ e)]) \Rightarrow (\mathit{new}, m)$: `if not`$(m = \texttt{map}([],[]))$ `then` e'

  Here, e is a list of metavariables. We create the formula $(\mathit{pname}\ e)$ with an unused predicate name *pname* just so we can pass it to `uniquefy`. If m is the empty map `map`$([],[])$ then `uniquefy` did not detect any metavariable as being used more than once, i.e., all metavariables in e are distinct. `distinctVars` executes e' otherwise.

3.2 A GSOS Validator in LANG-N-CHANGE

We now use LANG-N-CHANGE to write a GSOS validator. We divide our task into 5 parts. These are 5 checks that are meant to be performed in the order they appear, i.e., first Part 1, then Part 2, and so on.

[4] [31] used this method for a simpler version of this macro.

They all return skip if their corresponding check succeeds, otherwise they throw a runtime error. Thanks to the operations of LANG-N-CHANGE and our macros, these checks are easy to read, and we may omit commenting on some of them. Below, we use *math* font for LANG-N-CHANGE pattern variables.

- Part 1: All premises are positive or negative transition formulae, and they use constant labels.

```
rules[-->]: premises must match P --(op [])--> P' | P -/-(op [])-->
            otherwise error msg
```

  where *msg* = "Premises must be either positive labeled transitions or negative labeled transitions, and their label must be a constant". A constant is a term with a top-level operator and an empty list as arguments.

- Part 2: All conclusions are transition formulae that use a constant label, and are defined for an operator applied to metavariables as arguments. (Part 4 will check later that these metavariables are distinct, as they also need to be distinct from *ys*.)

```
rules[-->]: match conclusion with (op1 Ps)--(op2 [])--> P' ->
                  Ps[P]: if not(isVar(P)) then error msg1
            otherwise error msg2
```

  where msg_1 = "The operator that is the subject of the conclusion must have all metavariables as arguments", and msg_2 = "Conclusion formulae must be positive labeled transitions with a constant label and must apply to an operator".

- Part 3: Sources of premises must come from *xs* of the conclusion, and *ys* must be metavariables.

```
rules[-->]:
  if not(premises.LTsources sublistOf getVars(conclusion.LTsource))
  then error msg1
  else premises.LTtargets[P]: if not(isVar(P)) then error msg2
```

  where msg_1 = "Sources of premises must be arguments of the operator in the source of the conclusion", and msg_2 = "Targets of premises must be metavariables". Here, premises.LTsources, conclusion.LTsource, and premises.LTtargets are used after Part 1 and Part 2 have checked that we do have labeled transition formulae.

- Part 4: *xs* in the source of the conclusion, and *ys* in the premises must all be distinct.

```
rules[-->]:
  distinctVars (getVars(conclusion.LTsource) @ premises.LTtargets)
  otherwise error msg
```

  where *msg* = "The arguments of the operator in the source of the conclusion and the targets of the premises must all be distinct metavariables".

- Part 5: Metavariables in the target of the conclusion come from *xs* and *ys*.

```
rules[-->]:
 if not(getVars(conclusion.LTtarget)
              sublistOf
         (getVars(conclusion.LTsource) @ premises.LTtargets))
 then error msg
```

where `msg` = "The metavariables in the target of the conclusion must come from the source of the conclusion or from the targets of premises".

4 Evaluation

We have implemented the macros that we have described in this paper [14][5]. More precisely, we have not changed the core language of LANG-N-CHANGE, but we have added those constructors to the surface language for the convenience of programmers. These constructors are simply *parsed away*.

We have created a collection of test cases for our GSOS validator. We have defined a base language definition with only the prefix operator $l.P$. This language serves as a base to which we have added other features. Starting from this, we have created one language for each of the following concurrency operators: the interleaving parallel operator of CCS [29] (without process communication), the full parallel operator with communication of CCS [29], the synchronous parallel composition from CSP [27], the external choice of CCS, the internal choice of CSP, projection of ACP [10], hiding of CSP, left merge operator, the rename operator of CCS, the restriction operator of CCS, the "hourglass" operator from [2], signaling [8], the disrupt operator, the interrupt operator, the sequence operator ;, the priority operator, and a while-loop operator. These operators are known to satisfy the GSOS restrictions.

Our repo contains 18 concurrency operators, and we confirm that our GSOS validator successfully executes the checks of Section 3.2 (Part 1–5) on all these languages. That is, our tool validates the above-mentioned operators as adhering to the GSOS format. We have also performed a series of negative tests. Specifically, we have created languages that do not conform to the GSOS restrictions described in Section 3. We confirm that our GSOS validator fails in these cases and provide the corresponding error message.

Most of the checks of Section 3.2 can be written in one line despite having presented them in multiple lines for readability. Overall, we could write a GSOS validator in 6 lines of LANG-N-CHANGE code. This is a remarkably concise validator. Moreover, we believe that our code expresses the syntactic restrictions of the GSOS format declaratively. The website of our tool reports on all our tests [14].

5 Related Work

We are not aware of domain-specific languages that have been designed to express rule formats.

There are only a few tools that validate rule formats. PREG Axiomatizer [3] includes a checker for the GSOS format in around four hundred lines of Maude code[6]. Besides the format checks, this part implements methods for retrieving information from languages. For example, it implements functions for retrieving rules, searching premises, and obtaining the variables used in formulae, to name a few. As it turns out, these are functionalities that most language tools [15, 17, 24, 37, 39] need to use. It appears that each tool makes use of a specific programming language, stores languages as a data type of such programming language, and reimplements these retrieval functions. This is an issue that LANG-N-CHANGE can alleviate by providing a DSL for expressing them concisely and declaratively. It would be interesting to embed LANG-N-CHANGE into programming languages so that implementors can call its primitives.

[5]The flagship implementation of LANG-N-CHANGE is that of [32] but it uses a syntax that is more verbose than the one firstly proposed in [30]. We have then implemented a lightweight evaluator of the LANG-N-CHANGE DSL. The flagship implementation of LANG-N-CHANGE remains that of [32].

[6]We are thankful to Eugen-Ioan Goriac for kindly providing this estimation in private communications via email, and also for clarifying that such a GSOS checker is originally a part of PREG Axiomatizer rather than Meta SOS. Notice that if we did not count the code for parsing, the checker is estimated to be around 150 lines of Maude code.

Meta SOS [7] implements rule formats other than the GSOS format, hence a direct comparison is not possible. The tool of Mousavi and Reniers [33] provides a GSOS validator in Maude, and adopts a different implementation approach than [3]. Process algebras are provided as Maude rewriting rules. The tool then makes use of Maude introspective reflective features to explore the shape of rules, premises, and so on. Language designers can certainly use this approach to express their next rule formats, but it requires familiarity with Maude, with its reflective library, and with a very particular style of meta-programming that can have a steep learning curve. Some practitioners may find the linguistic features of LANG-N-CHANGE more intuitive and accessible.

6 Conclusion

Rule formats can quickly establish meta-theoretic properties of process algebras. It is then desirable to identify DSLs that can easily express rule formats and automatically test them. In this paper, we have observed that LANG-N-CHANGE offers convenient operations to interrogate operational semantics. We have created macros on top of LANG-N-CHANGE to better express some of the checks that often occur in rule formats. We have then used LANG-N-CHANGE and our macros to implement the GSOS rule format. Overall, we have written a full GSOS validator with only 6 lines of code, and we have used it to validate several concurrency operators. Moreover, our code expresses the GSOS restrictions declaratively.

In the future, we would like to apply our approach to other rule formats [34]. Several formats, including tyft, ntyft, path and panth [25, 26, 40], check for distinct variables, impose a specific shape for transition formulae, and retrieve sources and targets for analysis. We believe that LANG-N-CHANGE and our macros can be useful in those cases. However, there are possibly challenging aspects of our approach. For example, there may be operations that LANG-N-CHANGE does not implement yet, and whose need may be discovered at the attempt of capturing other rule formats. We have encountered an instance of this scenario when using LANG-N-CHANGE to automatically add references to certain pure functional languages [32]. This endeavor requires lifting the shape of reduction rules from $e \longrightarrow e$ to $e;\mu \longrightarrow e;\mu$, where μ is the heap. In that occasion we have extended LANG-N-CHANGE with an operator called `lift` to specify the change of shape for relations. That is, we have added a new operator that LANG-N-CHANGE lacked. (This operator turned out to be useful also for automating gradual typing in [31], and it is therein described.) Another challenge is that LANG-N-CHANGE presents some limitations. For example, a characteristic of the GSOS rule format is that its checks remain "local" and confined to the rule that has been selected for analysis. Some rule formats, instead, need to maintain a more global view on the process algebra at hand. An example is the rule format that establishes the commutativity of operators (modulo bisimilarity) [35], which compares multiple rules at the same time. LANG-N-CHANGE seems to be best suited to select one rule at a time. We will explore developing other DSLs, if other linguistic designs are more suitable.

Our tool is publicly available, and all our tests are documented at its GitHub repo [14].

References

[1] Luca Aceto, Arnar Birgisson, Anna Ingolfsdottir, MohammadReza Mousavi & Michel A. Reniers (2009): *Rule Formats for Determinism and Idempotence*. In: *Proceedings of the Third IPM International Conference on Fundamentals of Software Engineering*, FSEN'09, Springer-Verlag, Berlin, Heidelberg, pp. 146–161, doi:10.1007/978-3-642-11623-0_8.

[2] Luca Aceto, Bard Bloom & Frits Vaandrager (1994): *Turning SOS Rules into Equations*. Information and Computation 111(1), pp. 1–52, doi:10.1006/inco.1994.1040.

[3] Luca Aceto, Georgiana Caltais, Eugen-Ioan Goriac & Anna Ingolfsdottir (2011): *PREG Axiomatizer – A Ground Bisimilarity Checker for GSOS with Predicates*. In Andrea Corradini, Bartek Klin & Corina Cîrstea, editors: International Conference on Algebra and Coalgebra in Computer Science (CALCO 2011), Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 378–385, doi:10.1007/978-3-642-22944-2_27.

[4] Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Mohammad Reza Mousavi & Michel A. Reniers (2011): *SOS rule formats for zero and unit elements*. Theoretical Computer Science 412(28), pp. 3045–3071, doi:10.1016/j.tcs.2011.01.024.

[5] Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, MohammadReza Mousavi & Michel A. Reniers (2012): *Rule formats for distributivity*. Theoretical Computer Science 458, pp. 1–28, doi:10.1016/j.tcs.2012.07.036.

[6] Luca Aceto, Ignacio Fábregas, Álvaro García-Pérez, Anna Ingólfsdóttir & Yolanda Ortega-Mallén (2019): *Rule Formats for Nominal Process Calculi*. Logical Methods in Computer Science 15(4), doi:10.23638/LMCS-15(4:2)2019.

[7] Luca Aceto, Eugen-Ioan Goriac & Anna Ingólfsdóttir (2013): *Meta SOS - A Maude Based SOS Meta-Theory Framework*. In Johannes Borgström & Bas Luttik, editors: Proceedings Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics, EXPRESS/SOS 2013, Buenos Aires, Argentina, 26th August, 2013, EPTCS 120, pp. 93–107, doi:10.4204/EPTCS.120.8.

[8] J. C. M. Baeten & J. A. Bergstra (1992): *Process Algebra with Signals and Conditions*. In Manfred Broy, editor: Programming and Mathematical Method, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 273–323, doi:10.1007/978-3-642-77572-7_13.

[9] Falk Bartels (2002): *GSOS for probabilistic transition systems: (extended abstract)*. Electronic Notes in Theoretical Computer Science 65(1), pp. 29–53, doi:10.1016/S1571-0661(04)80358-X. CMCS'2002, Coalgebraic Methods in Computer Science (Satellite Event of ETAPS 2002).

[10] J.A. Bergstra & J.W. Klop (1984): *Process algebra for synchronous communication*. Information and Control 60(1), pp. 109–137, doi:10.1016/S0019-9958(84)80025-X.

[11] Bard Bloom (1995): *Structural operational semantics for weak bisimulations*. Theoretical Computer Science 146(1), pp. 25–68, doi:10.1016/0304-3975(94)00152-9.

[12] Bard Bloom, Sorin Istrail & Albert R. Meyer (1995): *Bisimulation Can't Be Traced*. Journal of the ACM 42(1), pp. 232–268, doi:10.1145/200836.200876.

[13] Roland Bol & Jan Friso Groote (1996): *The Meaning of Negative Premises in Transition System Specifications*. Journal of the ACM 43(5), pp. 863–914, doi:10.1145/234752.234756.

[14] Matteo Cimini (2022): *GSOS-Validator*. `https://github.com/mcimini/gsos-validator`.

[15] Matteo Cimini, Dale Miller & Jeremy G. Siek (2020): *Extrinsically typed operational semantics for functional languages*. In: Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020, pp. 108–125, doi:10.1145/3426425.3426936.

[16] Matteo Cimini & Benjamin Mourad (2021): *Language Transformations in the Classroom*. In Ornela Dardha & Valentina Castiglioni, editors: Proceedings Combined 28th International Workshop on Expressiveness in Concurrency and 18th Workshop on Structural Operational Semantics, EXPRESS/SOS 2021, and 18th Workshop on Structural Operational SemanticsParis, France (online event), 23rd August 2021, EPTCS 339, pp. 43–58, doi:10.4204/EPTCS.339.6.

[17] Matteo Cimini & Jeremy G. Siek (2016): *The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, Association for Computing Machinery, New York, NY, USA, pp. 443–455, doi:10.1145/2837614.2837632.

[18] Sjoerd Cranen, MohammadReza Mousavi & Michel A. Reniers (2008): *A Rule Format for Associativity*. In Franck van Breugel & Marsha Chechik, editors: *CONCUR 2008 - Concurrency Theory*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 447–461, doi:10.1007/978-3-540-85361-9_35.

[19] Pedro R. D'Argenio, Matias David Lee & Daniel Gebler (2015): *SOS rule formats for convex and abstract probabilistic bisimulations*. In Silvia Crafa & Daniel Gebler, editors: *Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS 2015, Madrid, Spain, 31st August 2015*, *EPTCS* 190, pp. 31–45, doi:10.4204/EPTCS.190.3.

[20] Marcelo Fiore & Sam Staton (2009): *A congruence rule format for name-passing process calculi*. *Information and Computation* 207(2), pp. 209–236, doi:10.1016/j.ic.2007.12.005.

[21] Wan J. Fokkink (2000): *Rooted Branching Bisimulation as a Congruence*. *Journal of Computer and System Sciences* 60(1), pp. 13–37, doi:10.1006/jcss.1999.1663.

[22] Wan J. Fokkink & Thuy Duong Vu (2003): *Structural operational semantics and bounded nondeterminism*. *Acta Informatica* 39, pp. 501–516, doi:10.1007/s00236-003-0111-1.

[23] Rob J. van Glabbeek (2005): *On Cool Congruence Formats for Weak Bisimulations*. In Dang Van Hung & Martin Wirsing, editors: *Theoretical Aspects of Computing – ICTAC 2005*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 318–333, doi:10.1007/11560647_21.

[24] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann & Mira Mezini (2015): *Type Systems for the Masses: Deriving Soundness Proofs and Efficient Checkers*. In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, ACM, New York, NY, USA, pp. 137–150, doi:10.1145/2814228.2814239.

[25] Jan Friso Groote (1993): *Transition system specifications with negative premises*. *Theoretical Computer Science* 118(2), pp. 263–299, doi:10.1016/0304-3975(93)90111-6.

[26] Jan Friso Groote & Frits Vaandrager (1992): *Structured operational semantics and bisimulation as a congruence*. *Information and Computation* 100(2), pp. 202–260, doi:10.1016/0890-5401(92)90013-6.

[27] C. A. R. Hoare (1978): *Communicating Sequential Processes*. *Communications of the ACM* 21(8), pp. 666–677, doi:10.1145/359576.359585.

[28] Ruggero Lanotte & Simone Tini (2009): *Probabilistic Bisimulation as a Congruence*. *ACM Transactions on Computational Logic* 10(2), pp. 9:1–9:48, doi:10.1145/1462179.1462181.

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

[30] Benjamin Mourad & Matteo Cimini (2020): *A Calculus for Language Transformations*. In: *46th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2020)*, Springer, pp. 547–555, doi:10.1007/978-3-030-38919-2_44.

[31] Benjamin Mourad & Matteo Cimini (2020): *A Declarative Gradualizer with Language Transformations*. In Olaf Chitil, editor: *IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020*, ACM, pp. 44–54, doi:10.1145/3462172.3462190.

[32] Benjamin Mourad & Matteo Cimini (2020): *System Description: Lang-n-Change - A Tool for Transforming Languages*. In Keisuke Nakano & Konstantinos Sagonas, editors: *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*, *Lecture Notes in Computer Science* 12073, Springer, pp. 198–214, doi:10.1007/978-3-030-59025-3_12.

[33] Mohammad Reza Mousavi & Michel A. Reniers (2006): *Prototyping SOS Meta-theory in Maude*. *Electronic Notes in Theoretical Computer Science* 156(1), pp. 135–150, doi:10.1016/j.entcs.2005.09.030. Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005).

[34] Mohammad Reza Mousavi, Michel A. Reniers & Jan F. Groote (2007): *SOS formats and meta-theory: 20 years after*. *Theoretical Computer Science* 373(3), pp. 238–272, doi:10.1016/j.tcs.2006.12.019.

[35] MohammadReza Mousavi, Michel Reniers & Jan Friso Groote (2005): *A syntactic commutativity format for SOS*. *Information Processing Letters* 93(5), pp. 217–223, doi:10.1016/j.ipl.2004.11.007.

[36] Gordon D. Plotkin (2004): *A structural approach to operational semantics*. *Journal of Logic and Algebraic Programming* 60-61, pp. 17–139, doi:10.1016/j.jlap.2004.05.001.

[37] Michael Roberson, Melanie Harries, Paul T. Darga & Chandrasekhar Boyapati (2008): *Efficient Software Model Checking of Soundness of Type Systems*. In Gail E. Harris, editor: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, Association for Computing Machinery, New York, NY, USA, pp. 493–504, doi:10.1145/1449764.1449803.

[38] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2010): *Ott: Effective tool support for the working semanticist*. *Journal of Functional Programming* 20(1), pp. 71–122, doi:10.1017/S0956796809990293.

[39] Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li & Grigore Roşu (2016): *Semantics-based program verifiers for all languages*. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pp. 74–91, doi:10.1145/2983990.2984027.

[40] C. Verhoef (1994): *A congruence theorem for structured operational semantics with predicates and negative premises*. In Bengt Jonsson & Joachim Parrow, editors: *CONCUR '94: Concurrency Theory*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 433–448, doi:10.1007/978-3-540-48654-1_32.

[41] Axelle Ziegler, Dale Miller & Catuscia Palamidessi (2006): *A Congruence Format for Name-passing Calculi*. *Electronic Notes in Theoretical Computer Science* 156(1), pp. 169–189, doi:10.1016/j.entcs.2005.09.032. Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005).

A Logical Account of Subtyping for Session Types

Ross Horne
University of Luxembourg

Luca Padovani
University of Camerino

We study the notion of subtyping for session types in a logical setting, where session types are propositions of multiplicative/additive linear logic extended with least and greatest fixed points. The resulting subtyping relation admits a simple characterization that can be roughly spelled out as the following lapalissade: every session type is larger than the smallest session type and smaller than the largest session type. At the same time, we observe that this subtyping, unlike traditional ones, preserves termination in addition to the usual safety properties of sessions. We present a calculus of sessions that adopts this subtyping relation and we show that subtyping, while useful in practice, is superfluous in the theory: every use of subtyping can be "compiled away" via a coercion semantics.

1 Introduction

Session types [12, 13, 15] are descriptions of communication protocols supported by an elegant correspondence with linear logic [23, 3, 16] that provides session type systems with solid logical foundations. As an example, below is the definition of a session type describing the protocol implemented by a mathematical server (in the examples of this section, $\&$ and $\oplus$ are n-ary operators denoting external and internal labeled choices, respectively):

$$B = \&\{\mathsf{end} : \bot, \mathsf{add} : \mathsf{Num}^\perp \parr \mathsf{Num}^\perp \parr \mathsf{Num} \otimes B\}$$

According to the session type B, the server first waits for a label – either end or add – that identifies the operation requested by the client. If the label is end, the client has no more requests and the server terminates. If the label is add, the server waits for two numbers, sends their sum back to the client and then makes itself available again offering the same protocol B. In this example, we write $\mathsf{Num}^\perp$ for the type of numbers being consumed and Num for the type of numbers being produced. A client of this server could implement a communication protocol described by the following session type:

$$A = \oplus\{\mathsf{add} : \mathsf{Num} \otimes \mathsf{Num} \otimes \mathsf{Num}^\perp \parr \oplus\{\mathsf{end} : \mathbf{1}\}\}$$

This client sends the label add followed by two numbers, it receives the result and then terminates the interaction with the server by sending the label end. When we connect two processes through a session, we expect their interaction to be flawless. In many session type systems, this is guaranteed by making sure that the session type describing the behavior of one process is the *dual* of the session type describing the behavior of its peer. *Duality*, often denoted by $\cdot^\perp$, is the operator on session types that inverts the *direction* of messages without otherwise altering the structure of protocol. In the above example it is clear that A is *not* the dual of B nor is B the dual of A. Nonetheless, we would like such client and such server to be declared compatible, since the client is exercising only a subset of the capabilities of the server. To express this compatibility we have to resort to a more complex relation between A and B, either by observing that B (the behavior of the server) is a *more accommodating* version of $A^\perp$ or by observing that A (the behavior of the client) is a *less demanding* version of $B^\perp$. We make these relations precise by means of a *subtyping relation* $\leqslant$ for session types. Subtyping enhances the applicability of

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop
on Programming Language Approaches to Concurrency and
Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. 26–37, doi:10.4204/EPTCS.378.3

© R. Horne and L. Padovani
This work is licensed under the
Creative Commons Attribution License.

type systems by means of the well-known substitution principle: an entity of type C can be used where an entity of type D is expected if C is a subtype of D. After the initial work of Gay and Hole [9] many subtyping relations for session types have been studied [4, 20, 17, 21, 10]. Such subtyping relations differ widely in the way they are defined and/or in the properties they preserve, but they all share the fact that subtyping is essentially defined by the branching structure of session types given by labels. To illustrate this aspect, let us consider again the session types A and B defined above. We have

$$B \leqslant \&\{\mathsf{add} : \mathsf{Num}^\perp \parr \mathsf{Num}^\perp \parr \mathsf{Num} \otimes \&\{\mathsf{end} : \perp\}\} = A^\perp \tag{1}$$

meaning that a server behaving as B can be safely used where a server behaving as $A^\perp$ is expected. Dually, we also have

$$A \leqslant \oplus\{\mathsf{end} : \mathbf{1}, \mathsf{add} : \mathsf{Num} \otimes \mathsf{Num} \otimes \mathsf{Num}^\perp \parr B^\perp\} = B^\perp \tag{2}$$

meaning that a client behaving as A can be safely used where a client behaving as $B^\perp$ is expected. Note how subtyping is crucially determined by the sets of labels that can be received/sent when comparing two related types. In (1), the server of type B is willing to accept any label from the set $\{\mathsf{end}, \mathsf{add}\}$, which is a *superset* of $\{\mathsf{add}\}$ that we have in $A^\perp$. In (2), the client is (initially) sending a label from the set $\{\mathsf{add}\}$, which is a subset of $\{\mathsf{end}, \mathsf{add}\}$ that we have in $B^\perp$. This co/contra variance of labels in session types is a key distinguishing feature of all known notions of subtyping for session types.[1]

In this work we study the notion of subtyping for session types in a setting where session types are propositions of $\mu\mathsf{MALL}^\infty$ [2, 6], the infinitary proof theory of multiplicative additive linear logic extended with least and greatest fixed points. Our investigation has two objectives. First, to understand whether and how it is possible to capture the well-known co/contra variance of behaviors when the connectives used to describe branching session types ($\&$ and $\oplus$ of linear logic) have fixed arity. Second, to understand whether there are criticial aspects of subtyping that become relevant when typing derivations are meant to be logically sound.

At the core of our proposal is the observation that, when session types (hence process behaviors) are represented by linear logic propositions [23, 3, 16], it is impossible to write a process that behaves as $\mathbf{0}$ and it is very easy to write a process that behaves as $\top$. If we think of a session type as the set of processes that behave according to that type, this means that the additive constants $\mathbf{0}$ and $\top$ may serve well as the least and greatest elements of a session subtyping relation. Somewhat surprisingly, the subtyping relation defined by these properties of $\mathbf{0}$ and $\top$ allows us to express essentially the same subtyping relations that arise from the usual co/contra variance of labels. For example, following our proposal the session type of the client, previously denoted A, would instead be written as

$$C = \oplus\{\mathsf{end} : \mathbf{0}, \mathsf{add} : \mathsf{Num} \otimes \mathsf{Num} \otimes \mathsf{Num}^\perp \parr \oplus\{\mathsf{end} : \mathbf{1}, \mathsf{add} : \mathbf{0}\}\}$$

using which we can derive both

$$B \leqslant \&\{\mathsf{end} : \top, \mathsf{add} : \mathsf{Num}^\perp \parr \mathsf{Num}^\perp \parr \mathsf{Num} \otimes \&\{\mathsf{end} : \perp, \mathsf{add} : \top\}\} = C^\perp \qquad \text{as well as} \qquad C \leqslant B^\perp$$

without comparing labels and just using the fact that $\mathbf{0}$ is the least session type and $\top$ the greatest one. Basically, instead of *omitting those labels* that correspond to impossible continuations (*cf.* the missing

[1] Gay and Hole [9] and other authors [4, 20, 21] define subtyping for session types in such a way that the *opposite* relations of eqs. (1) and (2) hold. Both viewpoints are viable depending on whether session types are considered to be types of *channels* or types of *processes*. Here we take the latter stance, referring to Gay [8] for a comparison of the two approaches.

Process	$P,Q ::=$		$\mid (x)(P \mid Q)$	composition	
	$\mathsf{A}\langle\overline{x}\rangle$	invocation	$\mid \mathsf{fail}\ x$	failure	
	$\mid\ x().P$	signal input	$\mid x[]$	signal output	
	$\mid\ x(y).P$	channel input	$\mid x[y](P \mid Q)$	channel output	
	$\mid\ \mathsf{case}\ x\{P,Q\}$	choice input	$\mid x[\mathsf{in}_i].P$	choice output	$i \in \{0,1\}$

Table 1: Syntax of $\mu\mathsf{CP}^\infty$.

end and add in A), we use the uninhabited session type $\mathbf{0}$ or its dual $\top$ *as impossible continuations* (*cf.* C). It could be argued that the difference between the two approaches is mostly cosmetic. Indeed, it is easy to devise (de)sugaring functions to rewrite session types from one syntax to the other. However, the novel approach we propose allows us to recast the well-known subtyping relation for session types in a logical setting. A first consequence of this achievement is that the soundness of the type system *with subtyping* does not require an *ad hoc* proof, but follows from the soundness of the type system *without subtyping* through a suitable coercion semantics. In addition, we find out that the subtyping relation we propose preserves not only the usual *safety properties* – communication safety, protocol fidelity and deadlock freedom – but also *termination*, which is a *liveness property*.

Structure of the paper. In Section 2 we define $\mu\mathsf{CP}^\infty$, a session calculus of processes closely related to $\mu\mathsf{CP}$ [16] and CP [23]. In Section 3 we define the type language for $\mu\mathsf{CP}^\infty$ and the subtyping relation. In Section 4 we define the typing rules for $\mu\mathsf{CP}^\infty$ and give a coercion semantics to subtyping, thus showing that the type system of $\mu\mathsf{CP}^\infty$ is a conservative extension of $\mu\mathsf{MALL}^\infty$ [2, 6]. We wrap up in Section 5.

2 Syntax and semantics of $\mu\mathsf{CP}^\infty$

The syntax of $\mu\mathsf{CP}^\infty$ is shown in Table 1 and makes use of a set of *process names* A, B, ... and of an infinite set of *channels* x, y, z and so on. The calculus includes standard forms representing communication actions: $\mathsf{fail}\ x$ models a process failing on x; $x().P$ and $x[]$ model the input/output of a termination signal on x; $\mathsf{case}\ x\{P,Q\}$ and $x[\mathsf{in}_i].P$ model the input/output of a label in_i on x; $x(y).P$ and $x[y](P \mid Q)$ model the input/output of a channel y on x. Note that $x[y](P \mid Q)$ outputs a *new* channel y which is bound in P but not in Q. Free channel output can be encoded as shown in previous works [16]. The form $(x)(P \mid Q)$ models a session x connecting two parallel processes P and Q and the form $\mathsf{A}\langle\overline{x}\rangle$ models the invocation of the process named A with parameters $\overline{x}$. For each process name A we assume that there is a unique global definition of the form $\mathsf{A}(\overline{x}) \triangleq P$ that gives its meaning. Hereafter $\overline{x}$ denotes a possibly empty sequence of channels. The notions of free and bound channels are defined in the expected way. We identify processes up to renaming of bound channels and we write $\mathsf{fn}(P)$ for the set of free channels of P.

The operational semantics of $\mu\mathsf{CP}^\infty$ is shown in Table 2 and consists of a structural pre-congruence relation $\preccurlyeq$ and a reduction relation $\rightarrow$, both of which are fairly standard. We write $P \rightarrow$ if $P \rightarrow Q$ for some Q and we say that P is *stuck*, notation $P \not\rightarrow$, if not $P \rightarrow$.

Example 2.1. We can model client and server described in Section 1 as the processes below.

$$\mathsf{Client}(x) \triangleq x[\mathsf{in}_1].x[\mathsf{in}_0].x[] \qquad \mathsf{Server}(x,z) \triangleq \mathsf{case}\ x\{x().z[], \mathsf{Server}\langle x,z\rangle\}$$

For simplicity, we only focus on the overall structure of the processes rather than on the actual mathematical operations they perform, so we omit any exchange of concrete data from this model. ⌟

[S-PAR-COMM]	$(x)(P \mid Q) \preccurlyeq (x)(Q \mid P)$	
[S-PAR-ASSOC]	$(x)(P \mid (y)(Q \mid R)) \preccurlyeq (y)((x)(P \mid Q) \mid R)$	$x \in \mathsf{fn}(Q) \setminus \mathsf{fn}(R), y \notin \mathsf{fn}(P)$
[S-CALL]	$\mathsf{A}\langle \overline{x} \rangle \preccurlyeq P$	$\mathsf{A}(\overline{x}) \triangleq P$
[R-CLOSE]	$(x)(x[] \mid x().P) \to P$	
[R-COMM]	$(x)(x[y](P \mid Q) \mid x(y).R) \to (y)(P \mid (x)(Q \mid R))$	
[R-CASE]	$(x)(x[\mathsf{in}_i].P \mid \mathsf{case}\, x\{Q_0, Q_1\}) \to (x)(P \mid Q_i)$	$i \in \{0,1\}$
[R-PAR]	$(x)(P \mid R) \to (x)(Q \mid R)$	$P \to Q$
[R-STRUCT]	$P \to Q$	$P \preccurlyeq P' \to Q' \preccurlyeq Q$

Table 2: Structrual pre-congruence and reduction semantics of $\mu\mathsf{CP}^\infty$.

We conclude this section with the definitions of the properties ensured by our type system, namely *deadlock freedom* and *termination*. The latter notion is particularly relevant in our setting since termination preservation is a novel aspect of the subtyping relation that we are about to define.

Definition 2.1 (deadlock-free process)**.** We say that P is *deadlock free* if $P \Rightarrow Q \not\to$ implies that Q is not (structurally pre-congruent to) a process of the form $(x)(R_1 \mid R_2)$.

A deadlock-free process either reduces or it is stuck waiting to synchronize on some free channel.

Definition 2.2 (terminating process)**.** A *run* of a process P is a (finite or infinite) sequence $(P_0, P_1, \dots)$ of processes such that $P_0 = P$ and $P_i \to P_{i+1}$ whenever $i+1$ is a valid index of the sequence. We say that a run is maximal if either it is infinite or if the last process in it is stuck. We say that P is *terminating* if every maximal run of P is finite.

Note that a terminating process is not necessarily free of restrictions. For example, $(x)(\mathsf{fail}\, x \mid x[])$ is terminated but not deadlock free. It really is the conjunction of deadlock freedom and termination (as defined above) that ensure that a process is "well behaved".

3 Types and subtyping

The type language for $\mu\mathsf{CP}^\infty$ consists of the propositions of $\mu\mathsf{MALL}^\infty$ [2, 6, 1], the infinitary proof theory of multiplicative/additive linear logic extended with least and greatest fixed points. We start from the definition of *pre-types*, which are linear logic propositions built using type variables taken from an infinite set and ranged over by X and Y.

$$\textbf{Pre-type} \qquad A, B ::= X \mid \bot \mid \mathbf{1} \mid \top \mid \mathbf{0} \mid A \parr B \mid A \otimes B \mid A \binampersand B \mid A \oplus B \mid \nu X.A \mid \mu X.A$$

The usual notions of free and bound type variables apply. A *type* is a closed pre-type. We assume that type variables occurring in types are *guarded*. That is, we forbid types of the form $\sigma_1 X_1 \dots \sigma_n X_n . X_i$ where $\sigma_1, \dots, \sigma_n \in \{\mu, \nu\}$. We write $A^\perp$ for the *dual* of A, which is defined in the expected way with the proviso that $X^\perp = X$. This way of dualizing type variables is not problematic since we will always apply $\cdot^\perp$ to types, which contain no free type variables. As usual, we write $A\{B/X\}$ for the (pre-)type obtained by replacing every X occurring free in the pre-type A with the type B. Hereafter we let κ range over the constants $\mathbf{0}$, $\mathbf{1}$, $\bot$ and $\top$, we let $\star$ range over the connectives $\binampersand$, $\oplus$, $\parr$ and $\otimes$ and σ range over the binders μ and ν. Also, we say that any type of the form $\sigma X.A$ is a σ-type.

$$\text{[BOT]}\ \frac{}{\mathbf{0} \leqslant A} \qquad \text{[TOP]}\ \frac{}{A \leqslant \top} \qquad \text{[REFL]}\ \frac{}{\kappa \leqslant \kappa} \qquad \text{[CONG]}\ \frac{A \leqslant A' \qquad B \leqslant B'}{A \star B \leqslant A' \star B'} \qquad \text{[LEFT-}\sigma\text{]}\ \frac{A\{\sigma X.A/X\} \leqslant B}{\sigma X.A \leqslant B} \qquad \text{[RIGHT-}\sigma\text{]}\ \frac{A \leqslant B\{\sigma X.B/X\}}{A \leqslant \sigma X.B}$$

Table 3: Subtyping for session types.

We write $\preceq$ for the standard *sub-formula* relation on types. To be precise, the relation $\preceq$ is the least preorder on types such that $A \preceq \sigma X.A$ and $A_i \preceq A_1 \star A_2$. For example, consider $A \stackrel{\text{def}}{=} \mu X.\nu Y.(1 \oplus X)$ and its unfolding $A' \stackrel{\text{def}}{=} \nu Y.(1 \oplus A)$. We have $A \preceq 1 \oplus A \preceq A'$, hence A is a sub-formula of A'. Given a set $\mathscr{T}$ of types we write $\min \mathscr{T}$ for the $\preceq$-minimum type in $\mathscr{T}$ when it is defined.

Table 3 shows the inference rules for subtyping judgments. The rules are meant to be interpreted coinductively so that a judgment $A \leqslant B$ is derivable if it is the conclusion of a finite/infinite derivation. The rules [BOT] and [TOP] establish that $\mathbf{0}$ and $\top$ are respectively the least and the greatest session type; the rules [REFL] and [CONG] establish reflexivity and pre-congruence of $\leqslant$ with respect to all the constants and connectives; the rules [LEFT-σ] and [RIGHT-σ] allow fixed points to be unfolded on either side of $\leqslant$.

Example 3.1. Consider the types $A \stackrel{\text{def}}{=} \mathbf{0} \oplus (\mathbf{1} \oplus \mathbf{0})$ and $B \stackrel{\text{def}}{=} \nu X.(\bot \mathbin{\&} X)$ which, as we will see later, describe the behavior of Client and Server in Example 2.1. We can derive both $A \leqslant B^\bot$ and $B \leqslant A^\bot$ thus:

$$\dfrac{\dfrac{}{\mathbf{0} \leqslant \mathbf{1}}\text{[BOT]} \qquad \dfrac{\dfrac{\dfrac{}{\mathbf{1} \leqslant \mathbf{1}}\text{[REFL]} \qquad \dfrac{}{\mathbf{0} \leqslant B^\bot}\text{[BOT]}}{\mathbf{1} \oplus \mathbf{0} \leqslant \mathbf{1} \oplus B^\bot}\text{[CONG]}}{\mathbf{1} \oplus \mathbf{0} \leqslant B^\bot}\text{[RIGHT-}\mu\text{]}}{\dfrac{A \leqslant \mathbf{1} \oplus B^\bot}{A \leqslant B^\bot}\text{[RIGHT-}\mu\text{]}}\text{[CONG]}
\qquad
\dfrac{\dfrac{}{\bot \leqslant \top}\text{[TOP]} \qquad \dfrac{\dfrac{\dfrac{}{\bot \leqslant \bot}\text{[REFL]} \qquad \dfrac{}{B \leqslant \top}\text{[TOP]}}{\bot \mathbin{⅋} B \leqslant \bot \mathbin{⅋} \top}\text{[CONG]}}{B \leqslant \bot \mathbin{\&} \top}\text{[LEFT-}\nu\text{]}}{\dfrac{\bot \mathbin{⅋} B \leqslant A^\bot}{B \leqslant A^\bot}\text{[LEFT-}\nu\text{]}}\text{[CONG]}$$

The rules [LEFT-σ] and [RIGHT-σ] may look suspicious since they are applicable to either side of $\leqslant$ regardless of the intuitive interpretation of μ and ν as least and greatest fixed points. In fact, if subtyping were solely defined by the derivability according to the rules in Table 3, the two fixed point operators would be equivalent. For example, both $\mu X.(\mathbf{1} \oplus X) \leqslant \nu X.(\mathbf{1} \oplus X)$ and $\nu X.(\mathbf{1} \oplus X) \leqslant \mu X.(\mathbf{1} \oplus X)$ are derivable even though only the first relation seems reasonable. We will see in Example 4.2 that allowing the second relation is actually *unsound*, in the sense that it compromises the termination property enjoyed by well-typed processes. We obtain a sound subtyping relation by ruling out some infinite derivations as per the following (and final) definition of subtyping.

Definition 3.1 (subtyping)**.** We say that A is a *subtype* of B if $A \leqslant B$ is derivable and, for every infinite branch $(A_i \leqslant B_i)_{i \in \mathbb{N}}$ of the derivation, either (1) $\min\{C \mid \exists^\infty i : A_i = C\}$ is a μ-type or (2) $\min\{C \mid \exists^\infty i : B_i = C\}$ is a ν-type. Hereafter $\exists^\infty i$ means the existence of infinitely many i's with the stated property.

The clauses (1) and (2) of Definition 3.1 make sure that μ and ν are correctly interpreted as least and greatest fixed points. In particular, we expect the least fixed point to be subsumed by a greatest fixed point, but not vice versa in general. For example, consider once again the (straightforward) derivations for the aforementioned subtyping judgments $\mu X.(\mathbf{1} \oplus X) \leqslant \nu X.(\mathbf{1} \oplus X)$ and $\nu X.(\mathbf{1} \oplus X) \leqslant \mu X.(\mathbf{1} \oplus X)$. The first derivation satisfies both clauses (there is only one infinite branch, along which a μ-type is unfolded infinitely many times on the left hand side of $\leqslant$ and a ν-type is unfolded infinitely many times on the right hand side of $\leqslant$). The second derivation satisfies neither clause. Therefore, $\mu X.(\mathbf{1} \oplus X)$ is a

$$\frac{P \vdash \overline{x : A}}{\mathsf{A}\langle\overline{x}\rangle \vdash \overline{x : A}}\ \mathsf{A}(\overline{x}) \triangleq P\ [\text{CALL}] \qquad \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : B}{(x)(P \mid Q) \vdash \Gamma, \Delta}\ A \leqslant B^{\perp}\ [\text{SUB}] \qquad \frac{}{\mathsf{fail}\ x \vdash \Gamma, x : \top}\ [\top] \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}\ [\bot]$$

$$\frac{}{x[] \vdash x : \mathbf{1}}\ [\mathbf{1}] \qquad \frac{P \vdash \Gamma, y : A, x : B}{x(y).P \vdash \Gamma, x : A \parr B}\ [\parr] \qquad \frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{x[y](P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B}\ [\otimes] \qquad \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Gamma, x : B}{\mathsf{case}\ x\{P, Q\} \vdash \Gamma, x : A \mathbin{\&} B}\ [\&]$$

$$\frac{P \vdash \Gamma, x : A_i}{x[\mathsf{in}_i].P \vdash \Gamma, x : A_0 \oplus A_1}\ i \in \{0,1\}\ [\oplus] \qquad \frac{P \vdash \Gamma, x : A\{\sigma X.A/X\}}{P \vdash \Gamma, x : \sigma X.A}\ [\sigma]$$

Table 4: Typing rules for $\mu\mathsf{CP}^{\infty}$.

subtype of $\nu X.(\mathbf{1} \oplus X)$ but $\nu X.(\mathbf{1} \oplus X)$ is not a subtype of $\mu X.(\mathbf{1} \oplus X)$. As we will see in Section 4, the application of a subtyping relation $A \leqslant B$ can be explicitly modeled as a process *consuming* a channel of type A while *producing* a channel of type B. According to this interpretation of subtyping, we can see that clause (1) of Definition 3.1 is just a dualized version of clause (2).

In both clauses of Definition 3.1 there is a requirement that the type of the fixed point on each side of the relation is determined by the $\preceq$-minimum of the types that appear infinitely often on either side. This is needed to handle correctly alternating fixed points, by determining which one is actively contributing to the infinite path. To see what effect this has consider the types $A \stackrel{\text{def}}{=} \mu X.\nu Y.(\mathbf{1} \oplus X)$, $A' \stackrel{\text{def}}{=} \nu Y.(\mathbf{1} \oplus A)$, $B \stackrel{\text{def}}{=} \mu X.\mu Y.(\mathbf{1} \oplus X)$ and $B' \stackrel{\text{def}}{=} \mu Y.(\mathbf{1} \oplus B)$. Observe that A unfolds to A', A' unfolds to $\mathbf{1} \oplus A$, B unfolds to B' and B' unfolds to $\mathbf{1} \oplus B$. We have $A \leqslant B$ despite Y is bound by a *greatest* fixed point on the left and by a *least* fixed point on the right. Indeed, both A and A' occur infinitely often in the (only) infinite branch of the derivation for $A \leqslant B$, but $A \preceq A'$ according to the intuition that the $\preceq$-minimum type that occurs infinitely often is the one corresponding to the outermost fixed point. In this case, the outermost fixed point is μX which "overrides" the contribution of the inner fixed point νY. The interested reader may refer to the literature on $\mu\mathsf{MALL}^{\infty}$ [2, 6] for details.

Hereafter, unless otherwise specified, we write $A \leqslant B$ to imply that A is a subtype of B and not simply that the judgment $A \leqslant B$ is derivable. It is possible to show that $\leqslant$ is a preorder and that $A \leqslant B$ implies $B^{\perp} \leqslant A^{\perp}$. Indeed, as illustrated in Example 3.1, we obtain a derivation of $B^{\perp} \leqslant A^{\perp}$ from that of $A \leqslant B$ by dualizing every judgment and by turning every application of [LEFT-σ] (respectively [RIGHT-σ], [BOT], [TOP]) into an application of [RIGHT-$\sigma^{\perp}$] (respectively [LEFT-$\sigma^{\perp}$], [TOP], [BOT]).

4 Typing rules

In this section we describe the typing rules for $\mu\mathsf{CP}^{\infty}$. Typing judgments have the form $P \vdash \Gamma$ where P is a process and Γ is a typing context, namely a finite map from channels to types. We can read this judgment as the fact that P behaves as described by the types in the range of Γ with respect to the channels in the domain of Γ. We write $\mathsf{dom}(\Gamma)$ for the domain of Γ, we write $x : A$ for the typing context with domain $\{x\}$ that maps x to A, we write Γ, Δ for the union of Γ and Δ when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. The typing rules of $\mu\mathsf{CP}^{\infty}$ are shown in Table 4 and, with the exception of [CALL] and [SUB], they correspond to the proof

rules of $\mu\mathsf{MALL}^\infty$ [2, 6] in which the context is the sequent being proved and the process is (almost) a syntactic representation of the proof. The rules for the multiplicative/additive constants and for the connectives are standard. The rule $[\sigma]$ where $\sigma \in \{\mu,\nu\}$ simply unfolds fixed points regardless of their nature. The rule [CALL] unfolds a process invocation into its definition, checking that the invocation and the definition are well typed in the same context. Finally, [SUB] checks that the composition $(x)(P \mid Q)$ is well typed provided that A (the behavior of P with respect to x) is a subtype of $B^\perp$ (where B is the behavior of Q with respect to x). In this sense [SUB] embeds the substitution principle induced by $\leqslant$ since it allows a process behaving as A to be used where a process behaving as $B^\perp$ is expected. Note that the standard cut rule of $\mu\mathsf{MALL}^\infty$ is a special case of [SUB] because of the reflexivity of $\leqslant$.

Like in $\mu\mathsf{MALL}^\infty$, the rules are meant to be interpreted coinductively so that a judgment $P \vdash \Gamma$ is deemed derivable if there is an arbitrary (finite or infinite) derivation whose conclusion is $P \vdash \Gamma$.

Example 4.1. Let us show the typing derivations for the processes discussed in Example 2.1. To this aim, let $A \overset{\mathsf{def}}{=} \mathbf{0} \oplus (\mathbf{1} \oplus \mathbf{0})$ and $B \overset{\mathsf{def}}{=} \nu X.(\bot \,\&\, X)$ and recall from Example 3.1 that $A \leqslant B^\perp$. We derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x[] \vdash \mathbf{1}}\,[\mathbf{1}]}{x[\mathsf{in}_0].x[] \vdash \mathbf{1} \oplus \mathbf{0}}\,[\oplus]}{x[\mathsf{in}_1].x[\mathsf{in}_0].x[] \vdash x : A}\,[\oplus]}{\mathsf{Client}\langle x\rangle \vdash x : A}\,[\text{CALL}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z[] \vdash z : \mathbf{1}}\,[\mathbf{1}]}{x().z[] \vdash x : \bot, z : \mathbf{1}}\,[\bot] \quad \dfrac{\vdots}{\mathsf{Server}\langle x,z\rangle \vdash x : B, z : \mathbf{1}}}{\mathsf{case}\, x\{x().z[], \mathsf{Server}\langle x,z\rangle\} \vdash x : \bot \,\&\, B, z : \mathbf{1}}\,[\&]}{\mathsf{case}\, x\{x().z[], \mathsf{Server}\langle x,z\rangle\} \vdash x : B, z : \mathbf{1}}\,[\nu]}{\mathsf{Server}\langle x,z\rangle \vdash x : B, z : \mathbf{1}}\,[\text{CALL}]}{(x)(\mathsf{Client}\langle x\rangle \mid \mathsf{Server}\langle x,z\rangle) \vdash z : \mathbf{1}}\,[\text{SUB}]$$

We can obtain a similar typing derivation by swapping Client and Server and using the relation $B \leqslant A^\perp$. Note that Client and Server cannot be composed directly using a standard cut since $A \neq B^\perp$. So, the use of subtyping in the above typing derivation is important to obtain a well-typed composition. ⌟

It is a known fact that not every $\mu\mathsf{MALL}^\infty$ derivation is a valid one [2, 6, 1]. In order to characterize the valid derivations we need some auxiliary notions which we recall below.

Definition 4.1 (thread)**.** Let $\gamma = (P_i \vdash \Gamma_i)_{i\in\mathbb{N}}$ be an infinite branch in a typing derivation and recall that $P_{i+1} \vdash \Gamma_{i+1}$ is a premise of $P_i \vdash \Gamma_i$. A *thread* of γ is a sequence $(x_i)_{i\geq k}$ of channels such that $x_i \in \mathsf{dom}(\Gamma_i)$ and either $x_i = x_{i+1}$ or $P_i = x_i[x_{i+1}](P_{i+1} \mid Q)$ or $P_i = x_i(x_{i+1}).P_{i+1}$ for every $i \geq k$.

Intuitively, a thread is an infinite sequence of channel names $(x_i)_{i\geq k}$ that are found starting from some position k in an infinite branch $(P_i \vdash \Gamma_i)_{i\in\mathbb{N}}$ and that pertain to the same session. For example, consider the derivation in Example 4.1 and observe that there is only one infinite branch, the rightmost one. The sequence $(x,x,x,\dots)$ is a thread that starts right above the conclusion of the derivation.

Definition 4.2 (ν-thread)**.** Given a branch $\gamma = (P_i \vdash \Gamma_i)_{i\in\mathbb{N}}$ and a thread $t = (x_i)_{i\geq k}$ of γ, we write $\mathsf{inf}(\gamma,t) \overset{\mathsf{def}}{=} \{A \mid \exists^\infty i \geq k : \Gamma_i(x_i) = A\}$. We say that t is a ν-*thread* of γ if $\min \mathsf{inf}(\gamma,t)$ is a ν-type.

Given a branch $\gamma = (P_i \vdash \Gamma_i)_{i\in\mathbb{N}}$ and a thread $t = (x_i)_{i\geq k}$ of γ, the thread identifies an infinite sequence $(\Gamma_i(x_i))_{i\geq k}$ of types. The set $\mathsf{inf}(\gamma,t)$ is the set of those types that occur infinitely often in this sequence and $\min \mathsf{inf}(\gamma,t)$ is the $\preceq$-minimum among these types (it can be shown that the minimum of any set $\mathsf{inf}(\gamma,t)$ is always defined [6]). We say that t is a ν-thread if such minimum type is a ν-type. In Example 4.1, the thread $t = (x,x,x,\dots)$ identifies the sequence $(B, B, \bot \,\&\, B, B, \dots)$ of types in which both B and $\bot \,\&\, B$ occur infinitely often. Since $B \preceq \bot \,\&\, B$ and B is a ν-type we conclude that t is a ν-thread.

$$\left[\!\!\left[\frac{}{\mathbf{0} \leqslant A}\right]\!\!\right]_{x,y} \triangleq \mathsf{fail}\ x \qquad \left[\!\!\left[\frac{}{\mathbf{1} \leqslant \mathbf{1}}\right]\!\!\right]_{x,y} \triangleq x().y[]$$

$$\left[\!\!\left[\frac{\pi_1 :: A \leqslant A' \qquad \pi_2 :: B \leqslant B'}{A \oplus B \leqslant A' \oplus B'}\right]\!\!\right]_{x,y} \triangleq \mathsf{case}\ x\{y[\mathsf{in}_0].\,[\![\pi_1]\!]_{x,y}\,, y[\mathsf{in}_1].\,[\![\pi_2]\!]_{x,y}\}$$

$$\left[\!\!\left[\frac{\pi_1 :: A \leqslant A' \qquad \pi_2 :: B \leqslant B'}{A \otimes B \leqslant A' \otimes B'}\right]\!\!\right]_{x,y} \triangleq x(u).y[v]([\![\pi_1]\!]_{u,v} \mid [\![\pi_2]\!]_{x,y}) \quad (u \text{ and } v \text{ fresh})$$

$$\left[\!\!\left[\frac{\pi :: A\{\sigma X.A/X\} \leqslant B}{\sigma X.A \leqslant B}\right]\!\!\right]_{x,y} \triangleq [\![\pi]\!]_{x,y} \qquad \left[\!\!\left[\frac{\pi :: A \leqslant B\{\sigma X.B/X\}}{A \leqslant \sigma X.B}\right]\!\!\right]_{x,y} \triangleq [\![\pi]\!]_{x,y}$$

Table 5: Coercion semantics of subtyping (selected equations).

Definition 4.3 (valid branch)**.** Let $\gamma = (P_i \vdash \Gamma_i)_{i\in\mathbb{N}}$ be an infinite branch of a typing derivation. We say that γ is *valid* if there is a ν-thread $(x_i)_{i\geq k}$ of γ such that $[\nu]$ is applied to infinitely many of the x_i.

Definition 4.3 establishes that a branch is valid if it contains a ν-thread in which the ν-type occurring infinitely often is also unfolded infinitely often. This happens in Example 4.1, in which the $[\nu]$ rule is applied infinitely often to unfold the type of x. The reader familiar with the $\mu\mathsf{MALL}^\infty$ literature may have spotted a subtle difference between our notion of valid branch and the standard one [2, 6]. In $\mu\mathsf{MALL}^\infty$, a branch is valid only provided that the ν-thread in it is not "eventually constant", namely if the greatest fixed point that defines the ν-thread is unfolded infinitely many times. This condition is satisfied by our notion of valid branch because of the requirement that there must be infinitely many applications of $[\nu]$ concerning the names in the ν-thread. Now we can define the notion of valid typing derivation.

Definition 4.4 (valid derivation)**.** A typing derivation is *valid* if so is every infinite branch in it.

Following Pierce [22] we provide a *coercion semantics* to our subtyping relation by means of two translation functions, one on derivations of subtyping relations $A \leqslant B$ and one on typing derivations $P \vdash \Gamma$ that make use of subtyping. The first translation is (partially) given in Table 5. The translation takes a derivation π of a subtyping relation $A \leqslant B$ – which we denote by $\pi :: A \leqslant B$ – and generates a process $[\![\pi]\!]_{x,y}$ that transforms (the protocol described by) A into (the protocol described by) B. The translation is parametrized by the two channels x and y on which the transformation takes place: the protocol A is "consumed" from x and reissued on y as a protocol B. In Table 5 we show a fairly complete selection of cases, the remaining ones being obvious variations. It is easy to establish that $[\![\pi]\!]_{x,y} \vdash x : A^\perp, y : B$ if $A \leqslant B$. In particular, consider an infinite branch $\gamma \stackrel{\mathsf{def}}{=} ([\![\pi_i]\!]_{x_i,y_i} \vdash x_i : A_i^\perp, y : B_i)_{i\in\mathbb{N}}$ in the typing derivation of the coercion where $A_0 = A$ and $B_0 = B$. This branch corresponds to an infinite branch $(A_i \leqslant B_i)_{i\in\mathbb{N}}$ in $\pi :: A \leqslant B$. According to Definition 3.1, either clause (1) or clause (2) holds for this branch. Suppose, without loss of generality, that clause (1) holds. Then $\min\{C \mid \exists^\infty i \in \mathbb{N} : A_i = C\}$ is a μ-type. According to Table 5 we have that $(x_i)_{i\in\mathbb{N}}$ is a ν-thread of γ, hence γ is a valid branch. Note that in general $[\![\pi]\!]_{x,y}$ is (the invocation of) a recursive process.

Concerning the translation of typing derivations, it is defined by the equation

$$\left[\!\!\left[\frac{\pi_1 :: P \vdash \Gamma, x : A \qquad \pi_2 :: Q \vdash \Gamma, x : B}{(x)(P \mid Q) \vdash \Gamma}\right]\!\!\right] = \frac{\dfrac{[\![\pi_1\{y/x\}]\!] \qquad [\![\pi]\!]_{y,x} \vdash y : A^\perp, x : B}{(y)(P\{y/x\} \mid [\![\pi]\!]_{y,x}) \vdash \Gamma, x : B} \qquad [\![\pi_2]\!]}{(x)((y)(P\{y/x\} \mid [\![\pi]\!]_{y,x}) \mid Q) \vdash \Gamma} \tag{3}$$

where $\pi :: A \leqslant B^\perp$ and extended homomorphically to all the other typing rules in Table 4. Note that (3) turns every application of the [SUB] into two applications of the standard $\mu\mathsf{MALL}^\infty$ cut rule. The validity of the resulting typing derivation follows immediately from that of the original typing derivation and that for the coercion, as argued earlier.

Thanks to the correspondence between $\mu\mathsf{CP}^\infty$'s typing rules and $\mu\mathsf{MALL}^\infty$, well-typed $\mu\mathsf{CP}^\infty$ processes are well behaved. In particular, processes that are well typed in a singleton context are deadlock free.

Theorem 4.1 (deadlock freedom)**.** *If $P \vdash x : A$ then P is deadlock free.*

Moreover, the cut elimination property of $\mu\mathsf{MALL}^\infty$ [2, 6] can be used to prove that well-typed $\mu\mathsf{CP}^\infty$ processes terminate, similarly to related systems [16, 5].

Theorem 4.2 (termination)**.** *If $P \vdash \Gamma$ then P is terminating.*

Proof sketch. The typing derivation for $P \vdash \Gamma$ with the subtype coercion made explicit maps directly to a valid $\mu\mathsf{MALL}^\infty$ proof. Every reduction step of P maps directly to one or more principal reductions in the $\mu\mathsf{MALL}^\infty$ proof. The reason why we could have more than one principal reduction for each process reduction comes from our choice of not having an explicit process form triggering the unfolding of a fixed point (see $[\sigma]$). Now, suppose that P has an infinite run. Then there would be an infinite sequence of reduction steps starting from P, hence an infinite sequence of cut reductions in the corresponding $\mu\mathsf{MALL}^\infty$ proof, which contradicts [6, Proposition 3.5]. Thus every run of P must be finite. □

Note that Theorem 4.2 only assures that a well-typed process will not reduce forever, not necessarily that the final configuration of the process is free of restricted sessions. These may occur guarded by a prefix concerning some free channel in the process. We can formulate a property of "successful termination" by combining Theorems 4.1 and 4.2.

Corollary 4.1. *If $P \vdash x : \mathbf{1}$ then P eventually reduces to $x[]$.*

We conclude this section with an example showing that the additional clauses of Definition 3.1 are key to making sure that $\leqslant$ is a termination-preserving subtyping relation.

Example 4.2. Consider a degenerate client $\mathsf{Chatter}(x) \triangleq x[\mathsf{in}_1].\mathsf{Chatter}\langle x\rangle$ that engages into an infinite interaction with Server from Example 2.1 and let $C \stackrel{\text{def}}{=} \nu X.(\mathbf{1} \oplus X)$. The derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{Chatter}\langle x\rangle \vdash x : C}\,[\nu]}{x[\mathsf{in}_1].\mathsf{Chatter}\langle x\rangle \vdash x : \mathbf{1} \oplus C}\,[\oplus]}{\mathsf{Chatter}\langle x\rangle \vdash x : \mathbf{1} \oplus C}\,[\text{CALL}]}{\mathsf{Chatter}\langle x\rangle \vdash x : C}\,[\nu]$$

is valid since the only infinite branch contains a ν-thread $(x, x, \dots)$ along which we find infinitely many applications of $[\nu]$. If we allowed the relation $C \leqslant B^\perp$ (*cf.* the discussion leading to Definition 3.1) the composition $(x)(\mathsf{Chatter}\langle x\rangle \mid \mathsf{Server}\langle x, z\rangle)$ would be well typed and it would no longer be the case that well-typed processes terminate, as the interaction between Chatter and Server goes on forever. ⌟

5 Concluding remarks

We have defined a subtyping relation for session types as the precongruence that is insensitive to the (un)folding of recursive types and such that $\mathbf{0}$ and $\top$ act as least and greatest elements. Despite the minimalistic look of the relation and the apparent rigidity in the syntax of types, in which the arity of internal and external choices is fixed, $\leqslant$ captures the usual co/contra variance of labels thanks to the interpretation given to $\mathbf{0}$ and $\top$. Other refinement relations for session types with least and greatest elements have been studied in the past [19, 21], although without an explicit correspondance with logic.

Unlike subtyping relations for session types [9, 4, 17, 10] that only preserve *safety properties* of sessions (communication safety, protocol fidelity and deadlock freedom), $\leqslant$ also preserves termination, which is a *liveness property*. For this reason, $\leqslant$ is somewhat related to *fair subtyping* [20, 21], which preserves *fair termination* [11, 7]. It appears that $\leqslant$ is coarser than fair subtyping, although the exact relationship between the two relations is difficult to characterize because of the fundamentally different ways in which recursive behaviors are represented in the syntax of types. The subtyping relation defined in this paper inherits least and greatest fixed points from $\mu\mathsf{MALL}^\infty$ [2, 6], whereas fair subtyping has been studied on session type languages that either make use of general recursion [20] or that use regular trees directly [21]. A more conclusive comparison is left for future work.

A key difference between the treatment of fixed points in this work and a related logical approach to session subtyping [14] is that, while both guarantee deadlock freedom, the current approach also guarantees termination. Insight concerning the design of fixed points should be exportable to other session calculi independently from any logical interpretation. In particular, it would be interesting to study subtyping for *asynchronous session types* [17, 10] in light of Definition 3.1. This can be done by adopting a suitable coercion semantics to enable buffering of messages as in simple orchestrators [18].

Acknowledgments. We are grateful to the anonymous reviewers for their thoughtful comments.

References

[1] David Baelde, Amina Doumane, Denis Kuperberg & Alexis Saurin (2022): *Bouncing Threads for Circular and Non-Wellfounded Proofs: Towards Compositionality with Circular Proofs*. In Christel Baier & Dana Fisman, editors: *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, ACM, pp. 63:1–63:13, doi:10.1145/3531130.3533375.

[2] David Baelde, Amina Doumane & Alexis Saurin (2016): *Infinitary Proof Theory: the Multiplicative Additive Case*. In Jean-Marc Talbot & Laurent Regnier, editors: *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, *LIPIcs* 62, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 42:1–42:17, doi:10.4230/LIPIcs.CSL.2016.42.

[3] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear logic propositions as session types*. *Math. Struct. Comput. Sci.* 26(3), pp. 367–423, doi:10.1017/S0960129514000218.

[4] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino & Luca Padovani (2009): *Foundations of session types*. In António Porto & Francisco Javier López-Fraguas, editors: *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, ACM, pp. 219–230, doi:10.1145/1599410.1599437.

[5] Farzaneh Derakhshan & Frank Pfenning (2022): *Circular Proofs as Session-Typed Processes: A Local Validity Condition*. *Logical Methods in Computer Science* Volume 18, Issue 2, doi:10.46298/lmcs-18(2:8)2022.

[6] Amina Doumane (2017): *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. Ph.D. thesis, Paris Diderot University, France. Available at `https://tel.archives-ouvertes.fr/tel-01676953`.

[7] Nissim Francez (1986): *Fairness*. Monographs in Comp. Sci., Springer, doi:`10.1007/978-1-4612-4886-6`.

[8] Simon J. Gay (2016): *Subtyping Supports Safe Session Substitution*. In Sam Lindley, Conor McBride, Philip W. Trinder & Donald Sannella, editors: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 9600, Springer, pp. 95–108, doi:`10.1007/978-3-319-30936-1_5`.

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

[10] Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas & Nobuko Yoshida (2022): *Precise Subtyping for Asynchronous Multiparty Sessions*. *ACM Trans. Comput. Logic*, doi:`10.1145/3568422`. Just Accepted.

[11] Orna Grumberg, Nissim Francez & Shmuel Katz (1984): *Fair Termination of Communicating Processes*. In: *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, Association for Computing Machinery, New York, NY, USA, pp. 254–265, doi:`10.1145/800222.806752`.

[12] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 509–523, doi:`10.1007/3-540-57208-2_35`.

[13] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Lisbon, Portugal, March 28 - April 4*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:`10.1007/BFb0053567`.

[14] Ross Horne (2020): *Session Subtyping and Multiparty Compatibility Using Circular Sequents*. In Igor Konnov & Laura Kovács, editors: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, *LIPIcs* 171, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 12:1–12:22, doi:`10.4230/LIPIcs.CONCUR.2020.12`.

[15] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira & Gianluigi Zavattaro (2016): *Foundations of Session Types and Behavioural Contracts*. *ACM Comput. Surv.* 49(1), pp. 3:1–3:36, doi:`10.1145/2873052`.

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

[17] Dimitris Mostrous & Nobuko Yoshida (2015): *Session typing and asynchronous subtyping for the higher-order π-calculus*. *Inf. Comput.* 241, pp. 227–263, doi:`10.1016/j.ic.2015.02.002`.

[18] Luca Padovani (2010): *Contract-based discovery of Web services modulo simple orchestrators*. *Theor. Comput. Sci.* 411(37), pp. 3328–3347, doi:`10.1016/j.tcs.2010.05.002`.

[19] Luca Padovani (2010): *Session Types = Intersection Types + Union Types*. In Elaine Pimentel, Betti Venneri & Joe B. Wells, editors: *Proceedings Fifth Workshop on Intersection Types and Related Systems, ITRS 2010, Edinburgh, U.K., 9th July 2010*, *EPTCS* 45, pp. 71–89, doi:`10.4204/EPTCS.45.6`.

[20] Luca Padovani (2013): *Fair Subtyping for Open Session Types*. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska & David Peleg, editors: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, *Lecture Notes in Computer Science* 7966, Springer, pp. 373–384, doi:`10.1007/978-3-642-39212-2_34`.

[21] Luca Padovani (2016): *Fair subtyping for multi-party session types*. *Math. Struct. Comput. Sci.* 26(3), pp. 424–464, doi:`10.1017/S096012951400022X`.

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

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

Communicating Actor Automata – Modelling Erlang Processes as Communicating Machines

Dominic Orchard
University of Kent, UK
d.a.orchard@kent.ac.uk

Mihail Munteanu
Masabi Ltd.
mihailmunteanu944@gmail.com

Paulo Torrens
University of Kent, UK
paulotorrens@gnu.org

Brand and Zafiropulo's notion of Communicating Finite-State Machines (CFSMs) provides a succinct and powerful model of message-passing concurrency, based around channels. However, a major variant of message-passing concurrency is not readily captured by CFSMs: the actor model. In this work, we define a variant of CFSMs, called Communicating Actor Automata, to capture the actor model of concurrency as provided by Erlang: with mailboxes, from which messages are received according to repeated application of pattern matching. Furthermore, this variant of CFSMs supports dynamic process topologies, capturing common programming idioms in the context of actor-based message-passing concurrency. This gives a new basis for modelling, specifying, and verifying Erlang programs. We also consider a class of CAAs that give rise to freedom from race conditions.

1 Introduction

Modern software development often deviates from the traditional approach of sequential computation and thrives on *concurrency*, where code is written such that several processes may run simultaneously while potentially sharing resources. Out of the increasing complexity of software systems, platforms and programming languages were created to facilitate the development of concurrent, parallel, and distributed computations, such as the Erlang programming language [1, 2]. The need for formal specification of such systems has motivated the design of formal systems that allow programs to be reasoned about.

The Communicating Finite-State Machines (CFSMs) of Brand and Zafiropulo (also known as *communicating automata*) provide a model for describing concurrent, communicating processes in which a notion of *well-formed* communication protocols can be described [7]. The essential idea is to model a (finite) set of concurrent processes as finite automata (one automaton per process) whose labels correspond to sending and receiving messages on channels connecting each pair of automata, collectively called a *protocol*. Through such precise descriptions, properties of communicating processes can be studied, e.g., checking whether every message is received, or whether no process is left awaiting for a message which is never sent.

This model is useful for further studying the decidability, or undecidability, of various properties of concurrent systems (e.g., [16, 17, 11, 13, 14]). For example, Brand and Zafiropulo show that boundedness (i.e., that communication can proceed with bounded queues), deadlock freedom, and unspecified reception (sending messages that aren't received) are all decidable properties when restricted to two machines with a single type of message [7]. Furthermore, deciding these properties can be computed in polynomial time [18], and can be computed when only one machine is restricted to a single type of message. CFSMs have also been employed more recently as a core modelling tool that provides a useful interface between other models of concurrent programs, such as graphical choreographies [15].

In the CFSM model, processes communicate via channels (FIFO queues) linking each process. Therefore a process knows from which other process a message is received. This differs to the actor

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop
on Programming Language Approaches to Concurrency and
Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. 38–48, doi:10.4204/EPTCS.378.4

© D. Orchard, M. Munteanu & P. Torrens

model where each process has a 'mailbox' into which other processes deposit messages, not necessarily with any information about the sender. This makes it difficult to capture actor-based approaches in traditional CFSMs. A further limitation of CFSMs is that they capture programs with fixed communication topologies: both sender and receiver of a message are fixed in the model, and a process cannot have its messages dynamically targeted to different processes. However, this is not the predominant programming idiom in concurrent programming settings. CFSMs also prescribe simple models of message reception, and do not capture more fine-grained reception methods, such as Erlang's mailbox semantics which allow processing messages other than the most recent one, leveraging pattern matching at the language level. Even so, CFSMs are tantalisingly close to Erlang's computational model, with every pair of processes representing sequential computation that is able to communicate bidirectionally.

We propose a variant of CFSMs to capture Erlang's asynchronous mailbox semantics, and furthermore allow dynamic topologies through the binding (and rebinding) of variables for process identifiers via a notion of memory within a process' automaton. Section 2 explicates the model including examples of models corresponding to simple Erlang programs. We consider properties of such models in Section 3. Section 4 concludes with a discussion, some related work, and next steps.

1.1 Communicating Finite-State Machines

To facilitate comparison, we briefly recap the formal definition of CFSMs [7]. A system of N-CFSMs is referred to as a *protocol* which comprises four components (usually represented as a 4-tuple):

- $(S_i)_{i=1}^{N}$ are N (disjoint) finite sets S_i giving the set of states of each process i;
- $(o_i)_{i=1}^{N}$ where $o_i \in S_i$ are the initial states of each process i;
- $(M_{i,j})_{i,j=1}^{N}$ are N^2 (disjoint) finite sets where $M_{i,j}$ represents the set of messages that can be sent from process i to process j, and where $M_{i,j} = \emptyset$ when $i = j$;
- $(\mathsf{succ} : (S_i \times \bigcup_{j=1}^{N}(M_{i,j} \uplus M_{j,i})) \to S_i)_{i=1}^{N}$ are N state transition functions (*partial* functions) where $\mathsf{succ}(s,l)$ computes the successor state s' for process i from the state s and given a message l that is either being sent from i to j (thus $l \in M_{i,j}$) or being received from j by i (thus $l \in M_{j,i}$).

The typical presentation views the above indexed sets as finite sequences. We use the notation l for messages as we later refer to these as being 'labels' of automata.

For a protocol, a *global state* (or configuration) is a pair (S,C) of a sequence of states for each process $S = \langle s_1 \in S_1, \dots s_N \in S_N \rangle$ and C is an $N \times N$ matrix whose elements $c_{i,j} \in C$ are finite sequences drawn from $M_{i,j}$ representing a FIFO queue (channel) of messages between process i and j (here and later we use $\cdot$ to concatenate such sequences and $[l_1, \dots, l_m]$ for an instance of a sequence with m messages).

A binary-relation *step* captures when one global state (S,C) can evolve into another global state (S',C') due to a single succ function. That is, $(S',C') \in \mathsf{step}(S,C)$ iff there exists i, k, $l_{i,k}$ and either:

	1.	(i sends to k):	$s'_i \in S' = \mathsf{succ}_i(s_i, l_{i,k})$ and $c'_{i,k} \in C' = c_{i,k} \cdot [l_{i,k}]$
or	2.	(reception from k by i):	$s'_k \in S' = \mathsf{succ}_k(s_k, l_{i,k})$ and $c_{i,k} \in C = [l_{i,k}] \cdot c'_{i,k}$

We adopt similar terminology and structure, but vary enough to capture the actor model of Erlang.

2 A variant of CFSMs for Erlang

Our main goal is to define a CFSM variant, which we call Communicating Actor Automata (CAA), by borrowing from Erlang's mailbox semantics. We first review some core Erlang concepts, and follow by

describing CAA and their composition into protocols. While the traditional definition of CFSMs immediately considers a global configuration (state) of some processes, we take each process (representing an Erlang actor) as a separate state machine, which gives a local "in-isolation" characterisation from which the global "protocol" characterisation is derived.

2.1 Erlang basic definitions

A key concept in Erlang is that of a *mailbox*: instead of processing messages strictly in FIFO order, each process (also referred to as an *actor*) possesses a queue of incoming messages from which they may *match*: given a sequence of patterns, a process picks the first message from the queue that unifies with one of those patterns, in an ordered fashion. If no pattern matches the first message in the queue, the next message is tried for all patterns, and so on [8]. In the following, we will use Erlang's term syntax in an abstract way (for details, see Carlsson et al. [9]), whose actual choice may vary. Concrete syntax is given in `monospace` font, e.g., `{1, 2}` is an Erlang tuple of two integer terms.

Definition 2.1 (Syntactic categories)**.** Let *Term* be the set of terms, ranged over by t. Let $\mathit{Var} \subset \mathit{Term}$ be the set of variables, ranged over by uppercase letters. Let $\mathit{Proc} \subset \mathit{Var}$ be the set of process identifiers which uniquely identify processes. Let *Pat* be the set of patterns, ranged over by pat. As Erlang has a call-by-value semantics, we define a subset $\mathit{Value} \subset \mathit{Term}$ of terms which are normal forms (called *values*), ranged over by v. Notice that $\mathit{Var} \subset \mathit{Value}$.

We note that, in regard to the semantics proposed in this paper, we mostly focus on four basic kinds of terms: process identifiers, variables, atoms and tuples. While identifiers and variables allow us to control the topology, atoms and tuples are useful for structuring message patterns (aiding in identifying which message is to be sent or received). In the example that will be given in Section 2.3, we shall also consider integers and arithmetic operators, but that's not necessary. It would also be possible to consider functions in the term syntax, but we won't entertain this possibility in this paper as we intend to focus on the expressivity of the process automaton itself and not on the term language.

In order to mimic Erlang's method of receiving messages, we need a notion of unification: incoming messages are matched against a set of patterns, and will proceed only if one of those is accepted.

Definition 2.2 (Unification)**.** We define the notion of *unification* [8, 9] between a term and a pattern written $t \rhd \mathit{pat}$ which either yields $\bot$ representing failure to pattern match or it yields an *environment* which is a finite map from a subset of variables $V \subseteq \mathit{Var}$ to terms, i.e., $\Gamma_V : V \to \mathit{Term}$ represents the binding context of a successful pattern match. Such maps are ranged over by γ. If $t \rhd \mathit{pat} = \gamma$, with $\gamma \in \Gamma_V$ for some V, then pat becomes equal to t if we replace every variable $v \in V$ in it by $\gamma_V(v)$. We write $\{X_1 \mapsto t_1,\ \dots,\ X_n \mapsto t_n\}$ to denote the environment that maps X_i to t_i (for all $1 \leq i \leq n$).

2.2 CAAs for individual processes

Just as in CFSMs, we describe actors by finite-state automata. Each automaton is described individually and represents a single Erlang process, having its own unique identifier. Our main intention is to capture possible states in an Erlang process, by saying which messages it's allowed to receive and to send at a given point during execution, and what it should do after it.

We formally define our notion of actor as follows (note the addition of final states, not always included for CFSMs).

Definition 2.3. A Communicating Actor Automata (CAA) is a 7-tuple $(S,\ o,\ F,\ \mathscr{L}_!,\ \mathscr{L}_?,\ \delta, <)$, which includes a finite set of states S, an initial state $o \in S$, a possibly empty set of final states $F \subseteq S$, a set of

send labels $\mathscr{L}_! \subseteq \mathit{Term}$, a finite set of receive labels $\mathscr{L}_? \subseteq \mathit{Pat}$, a function $\delta : S \times (\mathscr{L}_? \cup (\mathit{Var} \times \mathscr{L}_!)) \rightarrow S$, describing transitions, and an S-indexed family of order relations $<$ on $\mathscr{L}_?$. We impose a restriction on the domain of δ such that a state may only have either some number of receive labels or a single send label (but never both). We write $\delta(s, ?pat)$ for any transitions which receive a message matching *pat*, and $\delta(s, X!t)$ for transitions which send a message, where $X!t$ denotes the pair (X, t) of a message given by term t being sent to process X. For an state $s \in S$, we write $<_s$ as the order relation on such state.

Notice a CAA is essentially a deterministic finite automata (DFA) state machine with the alphabet defined as $\Sigma = \mathscr{L}_? \cup (\mathit{Var} \times \mathscr{L}_!)$. Non-determinism is exposed by interaction of many CAAs in a protocol, which will be described in Section 2.3. As such, a notion of a "static CAA" (as opposed to mobile) can be conceived of as having transitions $\delta : S \times (\mathscr{L}_? \cup (\mathit{Proc} \times \mathscr{L}_!)) \rightarrow S$ where we replace the variables associated with send labels by concrete process identifiers, i.e., the target of a send is always known ahead of time. We do not explore this notion further here.

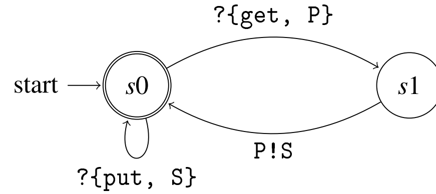
Example 2.1. Consider the following Erlang code which defines a function `mem` which emulates a memory cell via recursion:

```
mem(S) -> receive
            {get, P} -> P!S, mem(S);
            {put, X} -> mem(X)
          end.
```

The state of the memory cell is given by variable `S`. The process receives either a pair `{get, P}`, after which it sends `S` to the process identifier `P`, or a pair `{put, X}`, after which it recurses with `X` as the argument (the 'updated' memory cell state). Note that lowercase terms in Erlang are atoms.

Once spawned, this function can be modelled as a CAA with states $S = \{s_0,\ s_1\}$, initial and final states $o = s_0$ and $F = \{s_0\}$, send labels $\mathscr{L}_! = \mathit{Terms}$, receive labels $\mathscr{L}_? = \{$`{get, P}`, `{put, S}`$\}$, order stating `{get, P}` $<_{s_0}$ `{put, S}`, and the following transition (with corresponding automaton):

$$\begin{aligned} \delta(s_0,\ ?\texttt{\{get, P\}}) &= s_1 \\ \delta(s_0,\ ?\texttt{\{put, S\}}) &= s_0 \\ \delta(s_1,\ \texttt{P!S}) &= s_0 \end{aligned}$$



Notice we don't use the variable X in the above example for the 'put' message, as we want to rebind the received value (in the second component of the pair) to S in the recursive call. We do not consider the additional aspect of Erlang's semantics in which already bound variables may appear in pattern matches, incurring a unification, which is a further complication not considered in this paper.

The above definition is enough to capture the static semantics of an actor. However, during execution, further information is needed to represent its dynamic behaviour at runtime: namely the mailbox and the internal state of the actor. We proceed to formally define this.

Definition 2.4. A *local state* (or *machine configuration*) is a triple (s, m, γ), being comprised of a state s, a finite sequence of terms m, and an environment γ. The sequence of terms m models *message queues*, also called actor *mailboxes*, of unreceived messages, and γ is the actor's memory. We write ε for the empty sequence and $[t_1, \ldots, t_n]$ for the sequence comprising n elements with t_1 being the head of the queue. Two sequences m, m' can be appended, written $m \cdot m'$, e.g., $[1,2,3] \cdot [4,5] = [1,2,3,4,5]$.

2.3 Systems of CAAs: protocols, states, and traces

As with a CFSM model or an Erlang program, computation is described by the communication among concurrent actors. In order to formally define that, we give an operational semantics to the combination of several CAAs, called a protocol, through an evaluation step relation between states.

Definition 2.5 (Protocol)**.** A *protocol* is an indexed family of CAAs, of finite size (or arity) N, written $\langle(S_i,\ o_i,\ F_i,\ \mathscr{L}_{!i},\ \mathscr{L}_{?i},\ \delta_i)\rangle_{i=1}^{N}$. Each index i represents the unique process identifier of each process.

Definition 2.6 (Global state)**.** A *global state* (or *system configuration*) for a protocol comprises a finite sequence of N local states, written $\langle(s_i, m_i, \gamma_i)\rangle_{i=1}^{N}$ where every $s_i \in S_i$. We denote the set of global states for a protocol with arity N as G_N.

Given a protocol, we may derive what we call an initial global state: before starting, each process has an empty mailbox, and is in its initial state as defined in its own CAA. This initial state is deterministic: for any given protocol, there's a single possible initial state.

Definition 2.7 (Initial global state)**.** For a protocol, the *initial global state* is $\langle(o_i,\ \varepsilon,\ \emptyset)\rangle_{i=1}^{N} \in G_N$, i.e., every machine is in its initial state with an empty mailbox, and its mapping from variables to process identifiers is empty.

In order to use the mailbox semantics, we define a partial function that is defined only if a message may be accepted at the moment. As in Erlang, we look for each message that's in the mailbox in order, and only try the next message if no currently accepting pattern matches. While checking each message, patterns are checked in the defined order and only the first one that matches will be accepted.

Definition 2.8 (Pick function)**.** We implictly assume a CAA $(S,\ o,\ F,\ \mathscr{L}_!,\ \mathscr{L}_?,\ \delta,\ <)$ as our context. Then, the partial function $pick(s,m,v) = \langle pat, \gamma\rangle$ is defined if and only if:

- For all $pat' \in \mathscr{L}_?$, if $\delta(s, ?pat)$ is defined, then for all $v_k \in m$, we have that $v_k \rhd pat = \bot$;
- For all $pat' <_s pat$, we have $v \rhd pat' = \bot$; and
- There is a γ such that $v \rhd pat = \gamma$.

Finally, we now can define our semantics through a step relation, which defines what happens to a global state once one of the current possible transitions is performed.

Definition 2.9 (Step relation)**.** Given a protocol $\langle(S_i,\ o_i,\ F_i,\ \mathscr{L}_{!i},\ \mathscr{L}_{?i},\ \delta_i, <_i)\rangle_{i=1}^{N}$, the relation[1] *step* denotes the non-deterministic transitions of the overall concurrent system, of type $step : G_N \to \mathscr{P}(G_N)$. We write $step(c_1) \ni c_2$ if c_2 is amongst the possible outcomes of $step(c_1)$, as defined by the following two rules:

$$\frac{\delta_i(s_i, P!t) = s_i' \qquad \gamma_i(P) = j \qquad \gamma_i(t) \to^* v}{step(\langle(s_i, m_i, \gamma_i)\rangle_{i=1}^{N}) \quad \ni \quad \langle \ldots, (s_i', m_i, \gamma_i), \ldots, (s_j, m_j \cdot [v], \gamma_j), \ldots\rangle} \text{ (SEND)}$$

$$\frac{\delta_j(s_j, ?pat) = s_j' \qquad m_j = m \cdot [v] \cdot m' \qquad pick(s_j, m, v) = \langle pat, \gamma\rangle}{step(\langle(s_i, m_i, \gamma_i)\rangle_{i=1}^{N}) \quad \ni \quad \langle \ldots, (s_j', m \cdot m', \gamma_j \cup \gamma), \ldots\rangle} \text{ (RECV)}$$

[1] Notice *step* can be equivalently thought of as a non-deterministic function (producing many possible global states) or as a relation from a single input global state to many outcome global states. The definition here declaratively defines this relation.

The first rule, (SEND), actions a *send* transition with label $P!t$ for a process i in state s_i resulting in the state s_i'. We lookup the variable P from the process identifier environment γ_i to get the process identifier we are sending to, i.e., $\gamma_i(P) = j$, which means we are sending to process j. Notice that as γ_i is a finite map, $\gamma_i(P) = P$ if P is not in the domain of γ_i. By abuse of notation, $\gamma(t)$ replaces occurrences of variables of γ in t, allowing for the use of internal state, and then we evaluate the resulting term to a value v (which is denoted by the reduction relation $\rightarrow^*$). Subsequently in the resulting global configuration, process i is now in state s_i' and process j has the message v enqueued onto the end of its mailbox in its configuration, while the states for any other processes is kept the same.

The second rule, (RECV), actions a *receive* transition with label *pat* for a process j in state s_j resulting in the process j being in state s_j' under the condition that the mailbox m_j contains a message v at some point with prefix m and suffix m'. We use the *pick* function to check the semantic constraints: we want to pick the first message v for which some possible pattern matches, and the first one that does so. If this is the correct pattern, given state, prefix and value, then the value term v will unify with *pat* to produce an binding γ. Subsequently, process j is now in state s_j' with v removed from its mailbox and its process identifier environment updated to $\gamma_j \cup \gamma$: its internal state gets updated by any terms matched while receiving the message. Any states for processes other than j remains the same.

Following that, a way to describe a possible result for computation is through a trace, which represents a sequence of steps from the initial global state into one possible final state (as *step* is non-deterministic, several outcomes are possible). We proceed by formally defining a notion of trace.

Definition 2.10 (Trace)**.** A *trace* is a sequence of system configurations such that the first one is the initial global state, and the subsequent states are obtained from applying the *step* relation onto the previous configuration. We call a sequence of global states T a trace if it has the form $T = \langle t_0, t_1, ..., t_x \rangle$ and satisfies the following conditions:

- $t_0 = \langle (o_i, \varepsilon, \emptyset) \rangle_{i=1}^{N}$
- $step(t_i) \ni t_{i+1}$
- $step(t_x) = \emptyset$ where t_x is the last term in the trace sequence.

Example 2.2. Recall the `mem` function from Example 2.1, to which we assign process id 0. We consider a protocol comprises four machines, with three further machines with process identifiers 1, 2 and 3, given by the following definitions:

$$S_1 = \{c_0, c_1, c_2, c_3\} \quad o_1 = c_0 \quad F_1 = \{c_3\} \quad \mathscr{L}_{!1} = \{\texttt{\{get, 1\}}, \texttt{\{put, X+1\}}\} \quad \mathscr{L}_{?1} = \{\texttt{X}\}$$
$$\delta_1(c_0, 0!\texttt{\{get, 1\}}) = c_1 \quad \delta_1(c_1, ?\texttt{X}) = c_2 \quad \delta_1(c_2, 0!\texttt{\{put, X+1\}}) = c_3$$

$$S_2 = \{d_0, d_1, d_2, d_3\} \quad o_2 = d_0 \quad F_2 = \{d_3\} \quad \mathscr{L}_{!2} = \{\texttt{\{get, 2\}}, \texttt{\{put, X+2\}}\} \quad \mathscr{L}_{?2} = \{\texttt{X}\}$$
$$\delta_2(d_0, 0!\texttt{\{get, 2\}}) = d_1 \quad \delta_2(d_1, ?\texttt{X}) = d_2 \quad \delta_2(d_2, 0!\texttt{\{put, X+2\}}) = d_3$$

$$S_3 = \{e_0, e_1\} \quad o_3 = e_0 \quad F_3 = \{e_1\} \quad \mathscr{L}_{!3} = \{\texttt{\{put, 0\}}\} \quad \mathscr{L}_{?3} = \emptyset$$
$$\delta_3(e_0, 0!\texttt{\{put, 0\}}) = e_1$$

The first machine above (process id 1) makes a 'get' request to process 0 then receives the result `X` and sends back to 0 a 'put' message with `X+1`. The second machine above (process id 2) is similar to the first, requesting the value from process 0 but then sending back a 'put' message with `X+2`. The third machine (process id 3) sends to 0 a 'put' message with the initial value 0.

We can then get the following trace for the protocol $\langle (S_i, o_i, F_i, \mathscr{L}_{!i}, \mathscr{L}_{?i}, \delta_i) \rangle_{i=1}^{3}$, one of the many possibilities, demonstrating mobility and the ability to store internal state (we underline the parts of the configuration which have changed at each step of the trace for clarity):

$$
\begin{array}{llll}
\langle (s_0,\varepsilon,\emptyset), & (c_0,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & (e_0,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,[\{\texttt{put},0\}],\emptyset)}, & (c_0,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & \underline{(e_1,\varepsilon,\emptyset)}\rangle \\
\langle \underline{(s_0,\varepsilon,\{S\mapsto 0\})}, & (c_0,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,[\{\texttt{get},1\}],\{S\mapsto 0\})}, & \underline{(c_1,\varepsilon,\emptyset)}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_1,\varepsilon,\{S\mapsto 0,P\mapsto 1\})}, & (c_1,\varepsilon,\emptyset), & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,\varepsilon,\{S\mapsto 0,P\mapsto 1\})}, & \underline{(c_1,[0],\emptyset)}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle (s_0,\varepsilon,\{S\mapsto 0,P\mapsto 1\}), & \underline{(c_2,\varepsilon,\{X\mapsto 0\})}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,[\{\texttt{put},1\}],\{S\mapsto 0,P\mapsto 1\})}, & \underline{(c_3,\varepsilon,\{X\mapsto 0\})}, & (d_0,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,[\{\texttt{put},1\},\{\texttt{get},2\}],\{S\mapsto 0,P\mapsto 1\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_1,\varepsilon,\emptyset)}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,[\{\texttt{get},2\}],\{S\mapsto 1,P\mapsto 1\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & (d_1,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_1,\varepsilon,\{S\mapsto 1,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & (d_1,\varepsilon,\emptyset), & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,\varepsilon,\{S\mapsto 1,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_1,[1],\emptyset)}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle (s_0,\varepsilon,\{S\mapsto 1,P\mapsto 2\}), & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_2,\varepsilon,\{X\mapsto 1\})}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,[\{\texttt{put},3\}],\{S\mapsto 1,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & \underline{(d_3,\varepsilon,\{X\mapsto 1\})}, & (e_1,\varepsilon,\emptyset)\rangle \\
\langle \underline{(s_0,\varepsilon,\{S\mapsto 3,P\mapsto 2\})}, & (c_3,\varepsilon,\{X\mapsto 0\}), & (d_3,\varepsilon,\{X\mapsto 1\}), & (e_1,\varepsilon,\emptyset)\rangle
\end{array}
$$

3 Characterising CAA systems: race freedom and convergence

We consider how to characterise race conditions between CAAs and identify a subclass of CAA systems which is race free, or exhibiting *convergence*. To characterise race conditions, we first define the notion of what possible messages can be observed as 'incoming' to a process at a particular step in a trace. We define this notion via multisets of messages:

Definition 3.1 (Multiset of messages in a mailbox)**.** Given a mailbox m, we denote by A_m the multiset of elements in m, i.e., there is a way of 'arranging' the elements of A_m to obtain m.

Definition 3.2 (Incoming messages multiset)**.** Let $G_N = \langle c_1,...,(s_i,m_i,\gamma_i),...,c_N\rangle$ be a global configuration with process i at state s_i, where the next step of the system gives M possible configurations:

$$step(G_N) = \{\langle c_{11},...,(s_{i1},m_{i1},\gamma_{i1}),...,c_{N1}\rangle,...\langle c_{1M},...,(s_{iM},m_{iM},\gamma_{iM}),...,c_{NM}\rangle\}$$

The *incoming message multiset* I_i represents all the possible incoming messages for process i defined:

$$I_i = (\bigcup_{y=1}^{M} A_{m_{iy}}) - A_{m_i}$$

i.e., we take the union of all the possible mailbox multisets $A_{m_{iy}}$ for i obtained after a step is taken, from which is subtracted (multiset difference) the messages in the mailbox of i before that step. That is, the incoming messages I_i are all possible messages that can be added to the mailbox of i after taking a step.

The definition of I is implicitly parameterised by the starting configuration G_N. We will typically superscript I to denote it occurring at some position x in a trace, i.e., I_i^x.

Remark. We use overloaded notation $m_i = \langle I_i^0, I_i^1, ..., I_i^n\rangle$ to represent a mailbox where the first message is drawn from I_i^0, second from I_i^1 and so on. That is, $m_i = \langle l_{0i}, l_{1i}, ..., l_{ni}\rangle$ where $l_{xi} \in I_i^x$.

The mailbox of a process i can now be written as a sequence of multisets of incoming messages, more precisely a subsequence of $\langle I_i^0, I_i^1, ..., I_i^n\rangle$. Thus, $m_i = \langle I_i^{x_0}, I_i^{x_1}, ..., I_i^{x_m}\rangle$, where $x_0 < ... < x_m$. But why a subsequence? If message l_{xi} is consumed, then multiset I_i^x disappears from the mailbox.

Remark (Transition function δ notation overloading). Usually a transition function δ has as its argument a pair of a state and a label. We overload the second part of this pair to allow a multiset such that $\delta(s, ?I_i^x) = \{s' | , \forall l \in I_i^x, \delta(s, ?l) = s'\}$, i.e., the set of target states for any of the possible messages in I_i^x.

Definition 3.3 (Race condition)**.** For a protocol, a *race condition* represents the scenario in which there exists a state that can consume 2 or more messages from the same I_i^x and cannot consume any messages from the previous mailbox sets. That is:

$$\forall y \in [0, x-1].\ |\delta(s, ?I_i^y)| = 0 \wedge |\delta(s, ?I_i^x)| \geq 2$$

Intuitively, if in the current state for a process we can receive 2 or more messages from set I_i^x we are faced with a race condition, since these messages were sent at the same step and could arrive in any order. This is represented by the second part of the above conjunction. However, in order for us to reach I_i^x, we need to not consume any messages before that, hence the first part of the conjunction. If a previous multiset of messages has just one message we can consume, the race condition will not take place.

Trace convergence and race freedom for systems of two automata We consider a class of binary CAA systems (i.e., where $N = 2$) that is race free by showing that its traces always converge, that is, the system is deterministic. Furthermore, this makes the testing of a two automata system much easier, as we need only examine one trace to determine the final state of the system.

We recall that our definition of CAAs allows only for a restricted set of transitions: if a state has a transition, it can only be either some number of receive transition, properly ordered, or a single send transition. In fact, Erlang's semantics is such that transitions cannot be mixed in other ways and should have at most one send transition from any state, a condition we refer to as *affine sends*. We use this condition in order to reason about the possibility of non-determinism in a trace.

In the following, we will assume that *self-messaging* is disallowed: i.e., given any local state (s_i, m_i, γ_i) for $s_i \in S_i$, $\delta(s_i, P!t)$ is undefined for any P such that $\gamma_i(P) = i$[2]. For a class of binary models where no self messaging is allowed we observe (and formally prove below) that there will be no race conditions. A pre-requisite of a race condition (Definition 3.3) is that an actor can receive two incoming messages at the same trace step. By the stated conditions, there can only be one sender of messages at a time and therefore there is at most one transition for every state of an actor, across all global configurations. Thus, if a final state is reached then there will only be one possible trace to it.

To prove that we can have at most one transition at a specific time for an actor, we need to look at the mailbox. Send actions are affine (i.e., deterministic) and so we only need to show that receive transitions are deterministic. The proof considers two possible scenarios where we have two different global configurations for the system:

- Automata in both configurations are in states about to send, or both about to receive. Since both automata are going to perform the same type of action, they will only affect one mailbox; the mailbox configuration cannot diverge in this case.
- One automata is in a state about to send and the other to receive. If the actor in the receive state does not have a valid message to consume, the only valid action is the send (i.e., the other configuration progresses), otherwise both actions would impact the same mailbox making these two configurations diverge. However the send state is a "constructive" action, which will append a message at the end of the mailbox, while the receive action is "destructive" consuming a message from the mailbox, their disjoint nature will result in the convergence of the mailbox.

[2]This restriction is given to avoid both static messages to self, such as in $1!t$, and dynamic ones, such as $P!t$ where P is bound to 1. A more strict approach would be to require that each process only sends static messages to each other.

Proposition 3.1 (Convergence). *Let $X_1 = (S_1, o_1, F_1, \mathscr{L}_{!_1}, \mathscr{L}_{?_1}, \delta_1)$ and $X_2 = (S_2, o_2, F_2, \mathscr{L}_{!_2}, \mathscr{L}_{?_2}, \delta_2)$ form a protocol of two actors, where there are no mixed transitions, only affine sends, and no self messaging. Such a protocol, if it converges to a final state, will do so deterministically, i.e., any global configuration with converge to the same final global configuration, and as a consequence, be race free.*

Proof. We prove our desired goal by showing that any two traces T_1 and T_2 of the same size are the same. We note that a possible sequence of states must follow a pattern $\langle s_0, ..., s_n\rangle$, for some n, and show that for any $k \leq n$, the prefix of size k of T_1 and T_2 is the same. Taking k to be n, they are the same. This proof follows by well-founded induction on k (which has an upper bound):

1. Base case: we take $k = 1$. Both prefixes should then be $\langle s\rangle$, where s is the initial global state. This follows by definition of a trace, as it deterministically specifies which is the first state for any given configuration.
2. Inductive step: our inductive assumption says that our prefixes $\langle s_0...s_k\rangle$ match. If $k = n$, then we are done. If, however, $k \leq n$, then s_k is not a final state and we have that $T_1 = \langle s_0...s_k t_1\rangle$ and $T_2 = \langle s_0...s_k t_2\rangle$. We must now show that $t_1 = t_2$. Since we can only have one other process sending messages, we have that $\forall j \in [1, k+1], |I_i^{j+1}| \leq 1$. Now, since the size of the sets is at most one, consuming a message would nullify the set, therefore we can represents the mailbox as a subsequence of the set $\{I_i^0, I_i^1, .., I_i^k\}$, with $m_i = \langle I_i^{x_1}, I_i^{x_2}, ...I_i^{x_y}\rangle$ where $x_1 < x_2 < ... < x_y < k+1$. We find the greatest z such that $\forall j \in [1, z], |\delta(s_k, ?I_i^j)| = 0$. Let P_i be the set of all possible states after a receive transition. If $j = y$, then $|P_i| = 0$, so no messages can be consumed and we stay put at the same state, otherwise $P_i = \{s' | \forall m \in I_i^{z+1}, \delta(s_k, ?m) = s'\}$. Since we can receive at most one message, $|I_i^{z+1}| \leq 1$ therefore $|P_i| \leq 1$, having at most one possible transition, so just one possible new state. This can't be zero as we have t_1 and t_2: thus the list of possible next states has only one member, and $t_1 = t_2$ as expected.

□

Compatibility in Erlang Early work on CFSMs identified the notion of *compatibility* between machines as a key step towards guaranteeing progress of systems [14]. Compatibility is the automata analogue of what is commonly known as duality: that every receive has a corresponding send and vice versa. A pair of compatible, deterministic machines is then free from deadlock and unspecified receptions [14].

Due to Erlang's design principles, we might not always care about ending up with an empty mailbox and admit systems as being compatible even if some messages are not received, leaving remaining messages in the mailbox. For future work we thus propose 'tiers' of final conditions on a system which characterise, roughly, different levels of compatibility.

In the following, let the set of all possible traces for a system of CAAs be Y.

Definition 3.4. (Tier 1) A system is *strongly compatible* iff, $\forall T \in Y, t_{|T|} = \langle (f_i, \varepsilon, \gamma_i)\rangle_{i=1}^N$ where $f_i \in F_i$, i.e., no process has any message left in their mailboxes and they have reached final states.

Definition 3.5. (Tier 2) A system is *weakly compatible* iff, $\forall T \in Y, t_{|T|} = \langle (f_i, m_i, \gamma_i)\rangle_{i=1}^N$ where $f_i \in F_i$, i.e., some processes may have some messages left in their mailbox, but all have reached final states.

Definition 3.6. (Tier 3) A system is *communication-lacking* iff, $\forall T \in Y, t_{|T|} = \langle (s_i, \varepsilon, \gamma_i)\rangle_{i=1}^N$ where $s_i \in S_i/F_i$, i.e., processes didn't reach their final states but can't continue because of lack of input.

Definition 3.7. (Tier 4) If none of the previous conditions are met, the system is said to be *incompatible*.

We remark that the system in Example 2.2 is strongly compatible, as all possible traces will end up with empty mailboxes and in final states.

4 Discussion and Related work

Fowler describes a framework for generating runtime monitors for Erlang/OTP's `gen_server` behaviours from multiparty session types as conceived of in the Scribble language [12]. This leverages the idea, due to Deniélou and Yoshida [10] of projecting Scribble's global types (multiparty session types) into local types, and then implementing local types as CFSMs. It is not clear however to what extent this models Erlang's general mailbox semantics. This warrants a further investigation.

A classic mantra of Erlang is to "let it crash" (or "let it fail"). Our model here does not deal with process failure, although recent models have incorporated such aspects [4]. In the model of Bocchi et al. [4], actors communicate via unidirectional links to their mailboxes, similar to the structure of CFSMs, but with increased flexibility in the way that steps occur. The approach doesn't integrate pattern matching or dynamic topologies. Mailbox MSCs (Message Sequence Charts) [5, 6], build a message sequent chart model of processes with a single incoming channel and matching semantics similar in philosophy to our model but not directly based in the CFSM tradition. A deeper comparison with our approach is further work. One considerable difference in our approach is the integration of dynamic topologies by the 'memory' environment γ for each process, which enables messages to be sent to a variable which is a process identifier bound by a preceding receive.

In preliminary work, we have created a tool for extracting a CAA model from the Erlang code, and also the other way around, generating an Erlang skeleton of concurrent communicating code from a description of a CAA protocol. Further work includes developing this into a tool for analysis and specification of Erlang programs. We also note that due to the possibility to describe mobile processes, we conjecture that Milner's CPS translations from the λ-calculus into the π-calculus could be adapted to use CAAs as a target language. If that's the case, then it follows that CAAs describe a Turing-complete model of computation, and we intend to investigate this possibility.

Acknowledgments With thanks to Kartik Jalal for insights coming from model extraction of CAAs from Erlang, Simon Thompson for his Erlang expertise in the early days of this idea, and University of Kent undergraduates taking *CO545: Functional and Concurrent Programming* between 2017-20 for being a test audience for this model and its use in diagnosing race conditions and deadlocks. This work was partly supported by EPSRC project EP/T013516/1 (Granule). Orchard is also supported in part by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program.

References

[1] Joe Armstrong (1997): *The development of Erlang*. In: *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pp. 196–203, doi:10.1145/258948.258967.

[2] Joe Armstrong (2007): *A history of Erlang*. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pp. 6–1, doi:10.1145/1238844.1238850.

[3] Samik Basu, Tevfik Bultan & Meriem Ouederni (2012): *Deciding choreography realizability*. *ACM SIGPLAN Notices* 47(1), pp. 191–202, doi:10.1145/2103621.2103680.

[4] Laura Bocchi, Julien Lange, Simon Thompson & A. Laura Voinea (2022): *A Model of Actors and Grey Failures*. In Maurice H. ter Beek & Marjan Sirjani, editors: *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, *Lecture Notes in Computer Science* 13271, Springer, pp. 140–158, doi:10.1007/978-3-031-08143-9_9.

[5] Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Etienne Lozes & Amrita Suresh (2021): *A unifying framework for deciding synchronizability*. In: *CONCUR 2021-32nd International Conference on Concurrency Theory*, pp. 1–33, doi:10.4230/LIPIcs.CONCUR.2021.14.

[6] Ahmed Bouajjani, Constantin Enea, Kailiang Ji & Shaz Qadeer (2018): *On the completeness of verifying message passing programs under bounded asynchrony*. In: *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, Springer, pp. 372–391, doi:10.1007/978-3-319-96142-2_23.

[7] Daniel Brand & Pitro Zafiropulo (1983): *On communicating finite-state machines*. *Journal of the ACM (JACM)* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[8] Richard Carlsson (2001): *An introduction to Core Erlang*. In: *Proceedings of the PLI*, 1, Citeseer.

[9] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson & Robert Virding (2000): *Core Erlang 1.0 language specification*. *Information Technology Department, Uppsala University, Tech. Rep.*

[10] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty session types meet communicating automata*. In: *European Symposium on Programming*, Springer, pp. 194–213, doi:10.1007/978-3-642-28869-2_10.

[11] Alain Finkel & Etienne Lozes (2017): *Synchronizability of communicating finite state machines is not decidable*. *arXiv preprint arXiv:1702.07213*, doi:10.48550/arXiv.1702.07213.

[12] Simon Fowler (2016): *An Erlang implementation of multiparty session actors*. *arXiv preprint arXiv:1608.03321*, doi:10.4204/EPTCS.223.3.

[13] Mohamed G Gouda (1984): *Closed covers: to verify progress for communicating finite state machines*. *IEEE transactions on software engineering* (6), pp. 846–855, doi:10.1109/TSE.1984.5010313.

[14] Mohamed G Gouda, Eric G Manning & Yao-Tin Yu (1984): *On the progress of communication between two finite state machines*. *Information and control* 63(3), pp. 200–216, doi:10.1016/S0019-9958(84)80014-5.

[15] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In Sriram K. Rajamani & David Walker, editors: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, ACM, pp. 221–232, doi:10.1145/2676726.2676964.

[16] Jan Pachl (2003): *Reachability problems for communicating finite state machines*. *arXiv preprint cs/0306121*, doi:10.48550/arXiv.cs/0306121.

[17] Wuxu Peng & S Purushothaman (1992): *Analysis of a class of communicating finite state machines*. *Acta Informatica* 29(6-7), pp. 499–522, doi:10.1007/BF01185558.

[18] Louis E Rosier & Mohamed G Gouda (1984): *Deciding progress for a class of communicating finite state machines*.

Choreographic Programming of Isolated Transactions

Ton Smeele
Open University of the Netherlands
Heerlen, the Netherlands

Sung-Shik Jongmans
Open University of the Netherlands
Heerlen, the Netherlands
Centrum Wiskunde & Informatica (CWI)
Amsterdam, the Netherlands
ssj@ou.nl

Implementing distributed systems is hard; choreographic programming aims to make it easier. In this paper, we present the design of a new choreographic programming language that supports isolated transactions among overlapping sets of processes. The first idea is to track for every variable which processes are permitted to use it. The second idea is to use model checking to prove isolation.

1 Introduction

1.1 Background: Choreographic Programming

Implementing distributed systems is hard; *choreographic programming* aims to make it easier [8, 10, 37]. Figure 1 shows the idea.

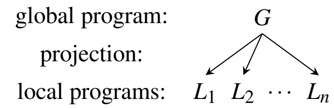


Figure 1: Method

Initially, a distributed system is written as a *global program* G ("the choreography"). It implements the behaviour of all processes collectively, in a sequential programming style (easy to write, but hard to run as a distributed system). For instance, the following global program implements a distributed system in which, first, a data object is communicated from Alice to Bob, and second, its hash.

$$G_{\mathtt{ab}} = (\mathtt{a}.\texttt{"foo"} \twoheadrightarrow \mathtt{b}.\mathtt{x}) \,;\, (\mathtt{a}.\mathtt{hash} {:=} \mathtt{md5}(\texttt{"foo"})) \,;\, (\mathtt{a}.\mathtt{hash} \twoheadrightarrow \mathtt{b}.\mathtt{y})$$

Here, $p.e \twoheadrightarrow q.y$ and $q.y{:=}e$ express inter-process *communication* and intra-process *computation*. Communication $p.e \twoheadrightarrow q.y$ implements the output of the value of expression e at process p and the corresponding input into variable y at process q; the transport is asynchronous, reliable, and FIFO. Computation $q.y{:=}e$ implements the storage of the value of expression e in variable y at process q.

Subsequently, the distributed system is run as a family of *local programs* $L_1, \ldots, L_n$, automatically extracted from the global program through *projection*. The local programs implement the behaviour of each process individually, in a parallel programming style (easy to run as a distributed system, but hard to write). For instance, the following local programs implement Alice and Bob:

$$L_{\mathtt{a}} = (\mathtt{ab}!\texttt{"foo"}) \,;\, (\mathtt{a}.\mathtt{hash} {:=} \mathtt{md5}(\texttt{"foo"})) \,;\, (\mathtt{ab}!\mathtt{hash}) \qquad L_{\mathtt{b}} = (\mathtt{ab}?\mathtt{x}) \,;\, (\mathtt{ab}?\mathtt{y})$$

Here, *send* $pq!e$ and *receive* $pq?y$ implement an output and an input through the channel from p to q.

The keystone assurance of choreographic programming is *operational equivalence*: methodically, a global program and its family of local programs are assured to have the same behaviour. To prove properties of families of local programs, operational equivalence allows us to prove them of global programs instead. This is typically simpler. A premier example of such a property is *absence of deadlocks*.

Choreographic programming originated with Carbone et al. [7, 8] (using binary session types [31]) and with Carbone and Montesi [10, 37] (using multiparty session types [32]); substantial progress has been made since. Montesi and Yoshida developed a theory of compositional choreographic programming

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. 49–60, doi:10.4204/EPTCS.378.5

© T. Smeele & S. Jongmans
This work is licensed under the Creative Commons Attribution License.

that supports open distributed systems [38]; Carbone et al. studied connections between choreographic programming and linear logic [6, 11]; Dalla Preda et al. combined choreographic programming with dynamic adaptation [39–41]; Cruz-Filipe and Montesi developed a minimal Turing-complete language of global programs [21]; Cruz-Filipe et al. and Kjær et al. presented techniques to extract global programs from families of local programs [17, 35]; Giallorenzo et al. studied a correspondence between choreographic programming and multitier languages [27]; Jongmans and Van den Bos combined choreographic programming with deductive verification [34]; Hirsch and Garg and Cruz-Filipe et al. developed functional choreographic programming languages [16, 30]. Other work includes results on case studies [18], procedural abstractions [20], asynchronous communication [19], polyadic communication [22, 29], implementability [26], and formalisation/mechanisation in Coq [23, 24, 30]. These theoretical developments are supported in practice by several tools [4, 10, 27, 40, 41].

1.2 Open Problem: Isolated Transactions

Suppose we need to implement a distributed system that fulfils the following requirements:

1. A data object and its hash are communicated from both Alice and Carol, in parallel, to Bob.
2. Either Alice's data object and its hash are eventually stored at Bob, or Carol's (but no mixture).

Requirement 1 can readily be fulfilled in a choreographic programming language with parallel composition (free interleaving), as demonstrated in the following global program:

$$G^{\mathrm{v1}}_{\mathsf{acb}} = G^{\S 1.1}_{\mathsf{ab}} \parallel G_{\mathsf{cb}} \qquad G_{\mathsf{cb}} = (\texttt{c."bar"} \twoheadrightarrow \texttt{b.x})\,;(\texttt{c.hash:=md5("bar")})\,;(\texttt{c.hash} \twoheadrightarrow \texttt{b.y})$$

In contrast, requirement 2 cannot be fulfilled in any choreographic programming language that we know of (i.e., none of the choreographic programming languages cited in §1.1 seem to be capable of it). What is needed, is a mechanism to run G_{ab} and G_{cb} as isolated *transactions*.

One possibility is to enrich the language with the standard non-deterministic choice operator $+$. In that case, the system can be implemented as $(G_{\mathsf{ab}}\,;G_{\mathsf{cb}}) + (G_{\mathsf{cb}}\,;G_{\mathsf{ab}})$. However, such an approach, in which parallel compositions are explicitly expanded into choices, generally leads to exponentially sized global programs (in the number of transactions), while obscuring the intention of the system. For instance, if Dave were added as a third client of Bob, we need to write the following global program:

$$\begin{aligned} G^{\mathrm{v1}}_{\mathsf{abcd}} = {} & (G_{\mathsf{ab}}\,;((G_{\mathsf{cb}}\,;G_{\mathsf{db}}) + (G_{\mathsf{db}}\,;G_{\mathsf{cb}}))) + \\ & (G_{\mathsf{cb}}\,;((G_{\mathsf{ab}}\,;G_{\mathsf{db}}) + (G_{\mathsf{db}}\,;G_{\mathsf{ab}}))) + \\ & (G_{\mathsf{db}}\,;((G_{\mathsf{ab}}\,;G_{\mathsf{cb}}) + (G_{\mathsf{cb}}\,;G_{\mathsf{ab}}))) \end{aligned} \qquad \begin{aligned} G_{\mathsf{db}} = {} & (\texttt{d."baz"} \twoheadrightarrow \texttt{b.x})\,; \\ & (\texttt{d.hash:=md5("baz")})\,; \\ & (\texttt{d.hash} \twoheadrightarrow \texttt{b.y}) \end{aligned}$$

Moreover, if we want to allow *independent segments* of transactions, which use disjoint sets of variables, to overlap to improve performance (i.e., their interleaved execution would not break isolation), then programmability is further complicated with the non-deterministic choice approach.

To avoid these issues, we propose a more fine-grained approach in this paper that supports eventual consistency while allowing for interleaved execution of isolated transactions. Instead of manually implementing isolated transactions by enumerating admissible sequences of communications, in our approach, isolation emerges out of explicit programming language support.

1.3 Contributions of This Paper

We present the design of a new choreographic programming language that supports isolated transactions.

The first idea is to track for each variable which processes are permitted to use it. Initially, each process is permitted to use each variable. Subsequently, process p can *acquire* exclusive permission

to use variable y of process q. When granted, each usage of y by not-p is blocked until p *releases* its exclusive permission. Management of usage permissions is transparant to the programmer; it is a feature of the programming language. The following global programs demonstrate the syntax and fulfil requirement 2 in §1.2:

$$G^{\text{v2}}_{\mathsf{acb}} = ((\mathsf{a}\ \mathbf{acq}\ \mathsf{b.x})\ ;\ G^{\S 1.1}_{\mathsf{ab}}\ ;\ (\mathsf{a}\ \mathbf{rel}\ \mathsf{b.x}))\ \|\ ((\mathsf{c}\ \mathbf{acq}\ \mathsf{b.x})\ ;\ G^{\S 1.2}_{\mathsf{cb}}\ ;\ (\mathsf{c}\ \mathbf{rel}\ \mathsf{b.x}))$$
$$G^{\text{v2}}_{\mathsf{acdb}} = G^{\text{v2}}_{\mathsf{acb}}\ \|\ ((\mathsf{d}\ \mathbf{acq}\ \mathsf{b.x})\ ;\ G^{\S 1.2}_{\mathsf{db}}\ ;\ (\mathsf{d}\ \mathbf{rel}\ \mathsf{b.x}))$$

We note that $G^{\text{v2}}_{\mathsf{acdb}}$ is *compositionally constructed* out of $G^{\text{v2}}_{\mathsf{acb}}$, without the need to refer to sub-programs G_{ab} and G_{cb}; this is not possible when parallel compositions are explicitly expanded into choices.

Thus, the idea of tracking usage permissions—and blocking those usages that are forbidden—enables the programmer to write more compact global programs, intended to better preserve the intention of the system. However:

- This feature does not guarantee isolation by itself; it is just a means to achieve it. In other words, a separate mechanism is still needed to check isolation and guarantee it is preserved by projection.
- "Blocking those usages that are forbidden" also has an adverse side-effect: processes that compete to acquire permission to use the same variables can deadlock. For instance, the following global program implements a system in which Alice tries to acquire permission to use variables `x` and `y` of Bob, while Carol tries to acquire permission to use the same variables, but in reverse:

  $$(\mathsf{a}\ \mathbf{acq}\ \mathsf{b.x}\ ;\ \mathsf{a}\ \mathbf{acq}\ \mathsf{b.y}\ ;\ \cdots)\ \|\ (\mathsf{c}\ \mathbf{acq}\ \mathsf{b.y}\ ;\ \mathsf{c}\ \mathbf{acq}\ \mathsf{b.x}\ ;\ \cdots)$$

  A deadlock arises when Alice acquires permission to use `x`, while Carol acquires permission to use `y`, so neither one of them can acquire permission to use a second variable.[1]

To address these points, the second idea of this paper is to specify properties, such as isolation and absence of deadlocks, in temporal logic and use model checking to prove that they are satisfied. We believe this combination with choreographic programming is new.

2 The Design

We define a language in which both global programs and families of local programs can be expressed. Figure 2 shows the design. It has four layers: every *system* is defined in terms of *programs* (either a single global one, or multiple local ones), *stores* (one for every process), and *channels* (one between every pair of processes); every program, store, or channel is defined in terms of *actions*, process/channel *names*, and *data*; every action is itself defined in terms of names and data, too.

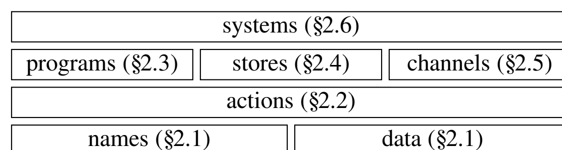


Figure 2: Design

2.1 Names and Data

First, we define: the syntax of names (Definition 1); the syntax of data (Definition 2). As the topic of interest is "processes that communicate", instead of "data that are communicated", we omit most details.

[1] We note that this is a different source of deadlock than the *communication deadlocks* that choreographic programming traditionally avoids (i.e., waiting for a message that is never sent).

Definition 1. Let $\mathbb{R} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \dots\}$ denote the set of *process names*, ranged over by p, q, r. Let $\mathbb{R} \times \mathbb{R} \setminus \{(r,r) \mid r \in R\}$ denote the set of *channel names*. □

Definition 2. Let $\mathbb{X} = \{_, \mathtt{x}, \mathtt{y}, \mathtt{z}, \dots\}$ denote the set of *variables*, ranged over by x, y, z. Let $\mathbb{V} = \{\mathtt{unit}, \mathtt{true}, \mathtt{false}, 0, 1, 2, \dots, \mathtt{acq}, \mathtt{rel}\}$ denote the set of *values*, ranged over by u, v, w. Let $\mathbb{E}$ denote the set of *expressions*, ranged over by E; it is defined as follows:

$$E \ ::= \ x \ \big| \ u \ \big| \ E_1 \,\texttt{==}\, E_2 \ \big| \ \texttt{\~{}}E \ \big| \ E_1 \,\texttt{\&\&}\, E_2 \ \big| \ E_1 \texttt{+} E_2 \ \big| \ \cdots$$ □

Symbol $_$ is a special variable that loses all data written to it, similar to `/dev/null` in Unix. Symbols `acq` and `rel` are special values to control usage permissions of variables (§2.4).

2.2 Actions

Next, we define: the syntax of actions that processes can execute (Definition 3); functions to retrieve the "subject" and the "object" of an action (Definition 4). The subject is the process that executes an action; the object is the channel through which an action is executed, if any.

Definition 3. Let $\mathbb{A}$ denote the set of *actions*, ranged over by α; it is defined as follows:

$$\alpha \ ::= \ p.E \ \big| \ q.y {:=} E \ \big| \ pq!E \ \big| \ pq?E \ \big| \ \tau$$ □

Action $p.E$ implements a *test* of expression E at process p. Action $q.y{:=}E$ implements an *assignment* of the value of expression E to variable y at process q. Actions $pq!E$ and $pq?E$ implement an *asynchronous send* and *receive* of the value of expression E from process p to process q. Action τ implements *idling*.

Definition 4. Let $\mathsf{subj}(\alpha)$ and $\mathsf{obj}(\alpha)$ denote the *subject* and the *object* of α; they are defined as follows:

$$\begin{array}{lll} \mathsf{subj}(p.E) \ \ = p & \mathsf{subj}(q.y{:=}E) = q & \mathsf{obj}(pq!E) = pq \\ \mathsf{subj}(pq!E) = p & \mathsf{subj}(pq?y) \ \ \ = q & \mathsf{obj}(pq?E) = pq \end{array}$$ □

2.3 Programs

Next, we define: the syntax of programs (Definition 5); a function to extract local programs from a global program (Definition 6); the operational semantics of programs (Definition 7).

Definition 5. Let $\mathbb{P}$ denote the set of *programs*, ranged over by P, G, L; it is defined as follows:

$$P \ ::= \ \mathbf{1} \ \big| \ \alpha \ \big| \ P_1 + P_2 \ \big| \ P_1 \parallel P_2 \ \big| \ P_1 \,;\, P_2$$ □

Program $\mathbf{1}$ implements an *empty execution*. Program $P_1 + P_2$ implements a *choice* between P_1 and P_2. Program $P_1 \parallel P_2$ implements an *interleaving* of P_1 and P_2. Program $P_1 \,;\, P_2$ implements a *sequence* of P_1 and P_2. Furthermore, we use the following shorthand notation:

$$\begin{array}{rll} p.E \twoheadrightarrow q.y & \text{instead of} & pq!E \,;\, pq?y \\ p \,\mathbf{acq}\, q.y & \text{instead of} & (p.\mathtt{acq} \twoheadrightarrow q.y) \,;\, (q.\mathtt{unit} \twoheadrightarrow p._) \\ p \,\mathbf{acq}\, q.[y_1, \dots, y_n] & \text{instead of} & p \,\mathbf{acq}\, q.y_1 \,;\, \cdots \,;\, p \,\mathbf{acq}\, q.y_n \\ p \,\mathbf{rel}\, q.y & \text{instead of} & p.\mathtt{rel} \twoheadrightarrow q.y \\ p \,\mathbf{rel}\, q.[y_1, \dots, y_n] & \text{instead of} & p \,\mathbf{rel}\, q.y_1 \,;\, \cdots \,;\, p \,\mathbf{rel}\, q.y_n \\ \mathbf{if}\ p.e\ P_1\ P_2 & \text{instead of} & (p.E \,;\, P_1) + (p.\texttt{\~{}}E \,;\, P_2) \end{array}$$

A program is *global* if at least two subjects occur in it; it is *local* if it at most one subject occurs in it. A local program for process r can be extracted from global program G through projection. The idea is to replace every action in G of which r is not the subject with τ.

Definition 6. Let $P \upharpoonright r$ denote the *projection* of P onto r; it is induced by the following equations:

$$\frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}} \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'} \qquad \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 + P_2 \xrightarrow{\alpha} P_2'} \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1' \parallel P_2} \qquad \frac{P_2 \xrightarrow{\alpha} P_2'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1 \parallel P_2'}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \,;\, P_2 \xrightarrow{\alpha} P_1' \,;\, P_2} \qquad \frac{\mathsf{subj}(\alpha) \notin \{\mathsf{subj}(\hat{\alpha}) \mid P_1 \to \cdots \to \xrightarrow{\hat{\alpha}}\} \qquad P_2 \xrightarrow{\alpha} P_2'}{P_1 \,;\, P_2 \xrightarrow{\alpha} P_2'}$$

(a) Programs. Let $\to \cdots \to$ denote a sequence of 0-or-more reductions.

$$\frac{S[\![E]\!]_p = \mathtt{true}}{S \xrightarrow[p.\mathtt{true}]{p.E} S} \qquad \frac{S[\![E]\!]_q = v}{S \xrightarrow[q.y:=v]{q.y:=E} S[y \mapsto v]_q} \qquad \frac{S[\![E]\!]_p = u}{S \xrightarrow[pq!u]{pq!E} S} \qquad \frac{}{S \xrightarrow[pq?v]{pq?y} S[y \mapsto v]_p} \qquad \frac{}{S \xrightarrow[\tau]{\tau} S}$$

(b) Stores

$$\frac{}{C \xrightarrow[p.v]{} C} \qquad \frac{}{C \xrightarrow[q.y:=v]{} C} \qquad \frac{|\vec{v}| < n}{(\vec{v}, n) \xrightarrow[pq!u]{} (u \cdot \vec{v}, n)} \qquad \frac{}{(\vec{u} \cdot v, n) \xrightarrow[pq?v]{} (\vec{u}, n)} \qquad \frac{}{C \xrightarrow[\tau]{} C}$$

(c) Channels

$$\frac{P \xrightarrow{\alpha} P'}{\{P\} \cup \mathcal{P} \xrightarrow{\alpha} \{P'\} \cup \mathcal{P}} \qquad \frac{S \xrightarrow[\underline{\alpha}]{\alpha} S' \qquad \mathsf{subj}(\alpha) = r}{\{\mathsf{subj}(\alpha) \mapsto S\} \cup \mathcal{S} \xrightarrow[\underline{\alpha}]{\alpha} \{\mathsf{subj}(\alpha) \mapsto S'\} \cup \mathcal{S}}$$

$$\frac{C \xrightarrow[\underline{\alpha}]{} C' \qquad \mathsf{obj}(\underline{\alpha}) = pq}{\{pq \mapsto C\} \cup \mathcal{C} \xrightarrow[\underline{\alpha}]{} \{pq \mapsto C'\} \cup \mathcal{C}} \qquad \frac{\mathcal{P} \xrightarrow{\alpha} \mathcal{P}' \qquad \mathcal{S} \xrightarrow[\underline{\alpha}]{\alpha} \mathcal{S}' \qquad \mathcal{C} \xrightarrow[\underline{\alpha}]{} \mathcal{C}'}{(\mathcal{P}, \mathcal{S}, \mathcal{C}) \xrightarrow[\underline{\alpha}]{\alpha} (\mathcal{P}', \mathcal{S}', \mathcal{C}')}$$

(d) Systems

Figure 3: Operational semantics

$$\begin{array}{llll} \alpha \restriction \mathsf{subj}(\alpha) = \alpha & & \mathbf{1} \restriction r = \mathbf{1} & \\ \alpha \restriction r \quad = \tau & \textbf{if: } r \neq \mathsf{subj}(\alpha) & P_1 \circ P_2 \restriction r = (P_1 \restriction r) \circ (P_2 \restriction r) & \textbf{if: } \circ \in \{+, \parallel, ;\} \end{array}$$ □

We define the operational semantics of programs through a labelled reduction relation.

Definition 7. Let $P \xrightarrow{\alpha} P'$ denote *reduction* from P to P' with α; it is defined in Figure 3a. □

Most rules are standard. The only special rule is the second rule for sequencing: it allows sequences of actions to be executed *out-of-order*, so long as they are executed at different processes (i.e., they are independent; insisting on a sequential order would be unreasonable in a parallel environment). That is, the left premise of the rule entails that the subject of α does not occur in P_1 (cf. the operational semantics of global multiparty session types). For instance, in a.x:=5 ; b.y:=6, the assignments at Alice and Bob may be executed out-of-order. In contrast, in a.x:=5 ; a.x+1 ⇾ b.y, the assignment and the communication must be executed in-order.

2.4 Stores

Next, we define: the syntax of stores (Definition 8); functions to read expressions from a store and write values to it (Definition 9); the operational semantics of stores (Definition 10).

Definition 8. Let $\mathbb{S} = (\mathbb{X} \setminus \{_\}) \rightharpoonup (\mathbb{V} \times 2^{\mathbb{R}})$ denote the set of *stores*, ranged over by S. □

Storage $S(x) = (u, R)$ means that variable x has value u, and that the processes in R are permitted to use it. Typically, $R \in \{\mathbb{R}\} \cup \{\{r\} \mid r \in \mathbb{R}\}$: either every process is permitted to use x (if $R = \mathbb{R}$), or only one process (if $R = \{r\}$ for some $r \in \mathbb{R}$). Every process has its own store, but through communications, other processes can use it, too.

Definition 9. Let $S[\![E]\!]_r$ and $S[y \mapsto v]_r$ denote the *read* of E in S by r and the *write* of v to y in S by r; they are defined as follows:

$$
\begin{array}{lll}
S[\![x]\!]_r = u \quad \textbf{if: } S(x) = (u, R) \textbf{ and } r \in R & & S[\![E_1 \texttt{==} E_2]\!]_r = \dots \\
S[\![u]\!]_r = u & & S[\![\texttt{\~{}}E]\!]_r = \dots \\
S[_ \mapsto v]_r = S & & S[\![E_1 \,\&\&\, E_2]\!]_r = \dots \\
S[y \mapsto v]_r = \{x \mapsto S(x) \mid x \neq y\} \cup \begin{cases} \{y \mapsto (v, R)\} & \textbf{if: } \texttt{acq} \neq v \neq \texttt{rel} \\ \{y \mapsto (u, \{r\})\} & \textbf{if: } \texttt{acq} = v \neq \texttt{rel} \\ \{y \mapsto (u, \mathbb{R})\} & \textbf{if: } \texttt{acq} \neq v = \texttt{rel} \end{cases} & & S[\![E_1 + E_2]\!]_r = \dots \\
\qquad \textbf{if: } y \neq _ \textbf{ and } S(y) = (u, R) \textbf{ and } r \in R & & \vdots \quad \vdots \; \vdots
\end{array}
$$
□

Writes $S[y \mapsto \texttt{acq}]_r$ and $S[y \mapsto \texttt{rel}]_r$ mean that process r tries to acquire or release exclusive permission to use y, without changing the value; it succeeds only if r already has permission (possibly non-exclusive).

The crux of the definition is that $S[\![E]\!]_r$ and $S[y \mapsto v]_r$ are undefined when r is not permitted to use a variable that occurs in E or y. Such undefinedness is leveraged in the operational semantics of stores (next definition). We note that $S[\![E]\!]_r$ is also undefined when operations are performed on sub-expressions of incompatible types. For instance, $S[\![\texttt{5} + \texttt{true}]\!]_r$ is undefined. A type system can be used to catch such errors statically; this is orthogonal to the aim of this paper.

We define the operational semantics of stores through a labelled reduction relation. Every reduction has two labels: an action (written above the arrow) and the "ground" version of the action (written below). In the ground version, every expression is replaced by its value, if any.

Definition 10. Let $S \xrightarrow[\underline{\alpha}]{\alpha} S'$ denote *reduction* from S to S' with α and $\underline{\alpha}$; it is defined in Figure 3b. □

The first rule states that a test $p.E$ is executed on a store by reading E, if the value of E is $\texttt{true}$, and if p has enough permissions. The second rule states that an assignment $q.y{:=}E$ is executed by reading E, and by writing the value of E to y, if q has enough permissions. The third rule states that a send $pq!E$ is executed by reading E, if p has enough permissions. The fourth rule states that a receive $pq?y$ and its ground version $pq?v$ are executed by writing v to y, if p has permission to use y (not q; essentially, we treat receives as remote assignments). If a process does not have enough permissions for a rule to be applicable, the store cannot reduce, so the action is blocked.

2.5 Channels

Next, we define: the syntax of channels (Definition 11); the operational semantics (Definition 12). Henceforth, we write $\vec{u}$ for a list of values, and we write $v{\cdot}\vec{u}$ and $\vec{u}{\cdot}v$ for prefixing and suffixing.

Definition 11. Let $\mathbb{C} = \mathbb{V}^* \times \{0, 1, 2, \dots, \infty\}$ denote a set of *channels*, ranged over by C. □

Channel $(\vec{u}, n)$ means that its n-capacity buffer contains the values in $\vec{u}$; the buffer is reliable and FIFO.

We define the operational semantics of channels through a labelled reduction relation. As channels contain values, every reduction has one label: a ground action (written below the arrow).

Definition 12. Let $C \underset{\underline{\alpha}}{\rightarrow} C'$ denote *reduction* from C to C' with $\underline{\alpha}$; it is defined in Figure 3c. □

The first and second rule state that a test and an assignment are executed on a channel without really using it. The third rule states that a send is executed by enqueueing a value to the buffer, if it is not full. The fourth rule states that a receive is executed by dequeueing a value from the buffer, if it is not empty. Henceforth, we omit reduction labels when they do not matter.

2.6 Systems

Last, we define: the syntax of systems (Definition 13); the operational semantics (Definition 14); operational equivalence (Definition 15)

Definition 13. Let $\mathbf{P} = 2^{\mathbb{P}} \setminus \{\emptyset\}$ denote the set of (non-empty) *sets of programs*, ranged over by $\mathcal{P}$. Let $\mathbf{S} = \mathbb{R} \rightharpoonup \mathbb{S}$ denote the set of *families of stores*, ranged over by $\mathcal{S}$. Let $\mathbf{C} = \mathbb{R} \times \mathbb{R} \rightharpoonup \mathbb{C}$ denote the set of *families of channels*, ranged over by $\mathcal{C}$. Let $\mathbf{P} \times \mathbf{S} \times \mathbf{C}$ denote the set of *systems*, ranged over by $\mathcal{D}$. □

System $(\mathcal{P}, \mathcal{S}, \mathcal{C})$ means that the program(s) in $\mathcal{P}$, the stores in $\mathcal{S}$, and the channels in $\mathcal{C}$ are executed together. It is well-formed if there exists a set of processes $R = \{r_1, \ldots, r_n\}$ such that the domain of $\mathcal{S}$ is R (every process has a store), and the domain of $\mathcal{C}$ is $R \times R$ (every pair of processes has a channel), and:

$$\mathcal{P} \in \{\{P\} \mid P \text{ is global and every subject that occurs in } P \text{ occurs in } R\} \cup$$
$$\{\{P_1, \ldots, P_n\} \mid \text{for each } 1 \le i \le n,\, P_{r_i} \text{ is local and every subject that occurs in } P_{r_i} \text{ is } r_i\}$$

We define the operational semantics of systems through a labelled reduction relation.

Definition 14. Let $(\mathcal{P}, \mathcal{S}, \mathcal{C}) \overset{\alpha}{\underset{\underline{\alpha}}{\rightarrow}} (\mathcal{P}, \mathcal{S}, \mathcal{C})'$ denote *reduction* from $(\mathcal{P}, \mathcal{S}, \mathcal{C})$ to $(\mathcal{P}, \mathcal{S}, \mathcal{C})$ with α and $\underline{\alpha}$; it is defined in Figure 3d. □

The first, second, and third rule lift reduction from individual programs, stores, and channels to sets of programs, families of stores, and families of channels. The fourth rule connects them together.

Two systems are operationally equivalent if they have the same behaviour. We formalise "having the same behaviour" in terms of *branching bisimilarity* [28] (in contrast to trace equivalence as usual), because: it is insensitive to idling; it preserves the validity of formulas in many temporal logics (including LTL, CTL, CTL*, and μ-calculus, subject to conditions), which we require to specify properties of global programs. Two systems (resp. processes, stores, channels, sets of processes, families of stores, families of channels) are branching bisimilar iff they can repeatedly mimic each other's reductions, modulo idling.

Definition 15. Let $\{\approx_1, \approx_2, \ldots\}$ denote the set of *branching bisimulations*, ranged over by $\approx$; it is defined as follows, coinductively:

$$\frac{\begin{array}{l}\bullet \text{ for each } \mathcal{D}_1 \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_1^\dagger \overset{\alpha}{\underset{\underline{\alpha}}{\rightarrow}} \mathcal{D}_1^\ddagger \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_1',\ \text{for some } \mathcal{D}_2 \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_2^\dagger \overset{\alpha}{\underset{\underline{\alpha}}{\rightarrow}} \mathcal{D}_2^\ddagger \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_2',\ \mathcal{D}_1^\dagger \approx \mathcal{D}_2^\dagger,\ \mathcal{D}_1^\ddagger \approx \mathcal{D}_2^\ddagger,\ \mathcal{D}_1' \approx \mathcal{D}_2' \\ \bullet \text{ for each } \mathcal{D}_2 \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_2^\dagger \overset{\alpha}{\underset{\underline{\alpha}}{\rightarrow}} \mathcal{D}_2^\ddagger \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_2',\ \text{for some } \mathcal{D}_1 \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_1^\dagger \overset{\alpha}{\underset{\underline{\alpha}}{\rightarrow}} \mathcal{D}_1^\ddagger \overset{\tau}{\underset{\tau}{\rightarrow}}{}^* \mathcal{D}_1',\ \mathcal{D}_1^\dagger \approx \mathcal{D}_2^\dagger,\ \mathcal{D}_1^\ddagger \approx \mathcal{D}_2^\ddagger,\ \mathcal{D}_1' \approx \mathcal{D}_2'\end{array}}{\mathcal{D}_1 \approx \mathcal{D}_2}$$

Let $\equiv\; = \;\approx_1 \cup \approx_2 \cup \cdots$ denote *operational equivalence* (i.e., the largest branching bisimulation). □

The following proposition states that operational equivalence of sets of programs implies that of the systems they constitute. Specifically, if P is a global program, and if $\{P\} \equiv \{P \upharpoonright r \mid r \text{ is a subject of } P\}$, then the local programs extracted from P have the same behaviour as P in *any* initial stores and channels. In the absence of loops, as in this paper, checking $\mathcal{P}_1 \equiv \mathcal{P}_2$ is clearly decidable; in the presence of loops, it is not. We leave decidable approximations of $\equiv$ (e.g., well-formedness conditions on the syntax of choices, as usual) for future work, when we extend our work with loops.

Proposition 1. For all $\mathcal{S},\mathcal{C}$, if $\mathcal{P}_1 \equiv \mathcal{P}_2$, then $(\mathcal{P}_1,\mathcal{S},\mathcal{C}) \equiv (\mathcal{P}_2,\mathcal{S},\mathcal{C})$. □

2.7 Properties

To prove properties, we adopt a state-based temporal logic in the style of CTL [25]. We are primarily interested in two classes of properties (although other classes may be specified, too): isolation of transactions and absence of deadlock; our logic has special predicates to formulate such properties. The need to explicitly prove absence of deadlock arises from the fact that systems in this paper are not deadlock-free by construction. For instance, any system that consists of the following program can deadlock (elaboration of the last example in §1.3):

$$G^{\text{v3}}_{\mathsf{acb}} = ((\mathsf{a}\ \mathbf{acq}\ \mathsf{b}.[\mathtt{x},\mathtt{y}])\ ;\ G^{\S1.1}_{\mathsf{ab}}\ ;\ (\mathsf{a}\ \mathbf{rel}\ \mathsf{b}.[\mathtt{x},\mathtt{y}])) \parallel ((\mathsf{c}\ \mathbf{acq}\ \mathsf{b}.[\mathtt{y},\mathtt{x}])\ ;\ G^{\S1.2}_{\mathsf{cb}}\ ;\ (\mathsf{c}\ \mathbf{rel}\ \mathsf{b}.[\mathtt{x},\mathtt{y}]))$$

The problem is that Alice acquires x and y (in that order), while Carol acquires y and x (in that order).

Definition 16. Let $\mathbb{F}$ denote the set of *formulas*, ranged over by φ; it is defined as follows:

$$\varphi \ ::= \ \top \ \big|\ \neg\varphi \ \big|\ \varphi_1 \wedge \varphi_2 \ \big|\ \mathsf{EG}(\varphi) \ \big|\ \mathsf{EU}(\varphi_1,\varphi_2) \ \big|\ p.E \ \big|\ \mathsf{AX}_{q.y}(\varphi) \ \big|\ \mathsf{dead} \qquad \square$$

Formula $\top$ specifies *truth*. Formulas $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ specify *negation* and *conjunction*. Formula $\mathsf{EG}(\varphi)$ specifies that, in some branch, φ is *always* true. Formula $\mathsf{EU}(\varphi_1,\varphi_2)$ specifies that, in some branch, φ_1 is true *until* φ_2 is true. Formula $p.E$ specifies *proposition* E at p. Formula $\mathsf{AX}_{q.y}(\varphi)$ specifies that φ is true *next* if variable y at process q was changed. Formula dead specifies the presence of deadlock. Furthermore, we use the following shorthand notation (standard):

$$\begin{array}{rlrl} \bot & \text{instead of } \ \neg\top & \mathsf{AG}(\varphi) & \text{instead of } \ \mathsf{EU}(\top,\neg\varphi) \\ \phi_1 \vee \phi_2 & \text{instead of } \ \neg(\neg\phi_1 \wedge \neg\phi_2) & \mathsf{AU}(\varphi_1,\varphi_2) & \text{instead of } \ \neg(\mathsf{EU}(\neg\varphi_2,\neg(\varphi_1 \vee \varphi_2)) \vee \mathsf{EG}(\neg\varphi_2)) \end{array}$$

Definition 17. Let $\mathcal{D} \models \varphi$ denote *entailment* of φ by $\mathcal{D}$; it is defined as follows:

$$\frac{}{\mathcal{D} \models \top} \qquad \frac{\mathcal{D} \not\models \varphi}{\mathcal{D} \models \neg\varphi} \qquad \frac{\mathcal{D} \models \varphi_1 \qquad \mathcal{D} \models \varphi_2}{\mathcal{D} \models \varphi_1 \wedge \varphi_2} \qquad \frac{\mathcal{D} \to \mathcal{D}' \qquad \mathcal{D} \models \varphi \qquad \mathcal{D}' \models \mathsf{EG}(\varphi)}{\mathcal{D} \models \mathsf{EG}(\varphi)}$$

$$\frac{\mathcal{D} \models \varphi_2}{\mathcal{D} \models \mathsf{EU}(\varphi_1,\varphi_2)} \qquad \frac{\mathcal{D} \to \mathcal{D}' \qquad \mathcal{D} \models \varphi_1 \qquad \mathcal{D}' \models \mathsf{EU}(\varphi_1,\varphi_2)}{\mathcal{D} \models \mathsf{EU}(\varphi_1,\varphi_2)}$$

$$\frac{\mathcal{S} \xrightarrow[p.\mathtt{true}]{p.E} \mathcal{S}}{(\mathcal{P},\mathcal{S},\mathcal{C}) \models p.E} \qquad \frac{\begin{array}{c}\text{for each } (\mathcal{P},\mathcal{S},\mathcal{C}) \to (\mathcal{P}',\mathcal{S}',\mathcal{C}') \\ \text{if } \mathcal{S}(q)(y) \neq \mathcal{S}'(q)(y), \text{ then } (\mathcal{P}',\mathcal{S}',\mathcal{C}') \models \varphi\end{array}}{(\mathcal{P},\mathcal{S},\mathcal{C}) \models \mathsf{AX}_{q.y}(\varphi)} \qquad \frac{\begin{array}{c}\mathcal{P} \to \\ (\mathcal{P},\mathcal{S},\mathcal{C}) \not\to\end{array}}{(\mathcal{P},\mathcal{S},\mathcal{C}) \models \mathsf{dead}} \qquad \square$$

The rules on the first two lines are the standard ones for CTL. The first rule on the third line states that a proposition is true if the corresponding test succeeds. The second rule on the third line states that every reduction that changes variable y at process q must make φ true. The third rule on the third line states that the presence of deadlock is true if the set of programs can reduce, but the system cannot (i.e., program reduction is blocked by stores and/or channels).

In the absence of loops, as in this paper, the model checking problem is decidable: it is straightforward to adapt classical model checking algorithms for CTL (e.g., Clarke et al. [13]) to also support our formulas $p.E$, $\mathsf{AX}_{q.y}(\varphi)$, and dead. If a global program G satisfies operational equivalence, then it suffices to model check the system that consists of G instead of model checking the system that consists of G's projections; the former is generally much more efficient as the state space of G's projections can be exponentially larger than that of G (due to τ-reductions of the projections).

2.8 Examples

We end this section with some examples. Let:

$$\mathcal{S} = \{\mathsf{a} \mapsto \{\mathtt{hash} \mapsto \mathtt{0}\}, \mathsf{b} \mapsto \{\mathtt{x} \mapsto \mathtt{""}, \mathtt{y} \mapsto \mathtt{0}\}, \mathsf{c} \mapsto \{\mathtt{hash} \mapsto \mathtt{0}\}\}$$
$$\mathcal{C} = \{pq \mapsto (\epsilon, \infty) \mid p,q \in \{\mathsf{a},\mathsf{b},\mathsf{c}\} \text{ and } p \neq q\}$$

In words, $\mathcal{S}$ is an initial family of stores (for Alice, Bob, and Carol) in which all variables have default values, while $\mathcal{C}$ is an initial family of empty channels (between Alice, Bob, and Carol). Furthermore, in addition to $G^{\mathrm{v1}}_{\mathsf{acb}}$ in §1.2, $G^{\mathrm{v2}}_{\mathsf{acb}}$ in §1.3, and $G^{\mathrm{v3}}_{\mathsf{acb}}$ in §2.7, let:

$$G^{\mathrm{v4}}_{\mathsf{acb}} = ((\mathsf{a}\ \mathbf{acq}\ \mathsf{b}.\mathtt{x})\ ;\ G^{\S 1.1}_{\mathsf{ab}}\ ;\ (\mathsf{a}\ \mathbf{rel}\ \mathsf{b}.\mathtt{x}))\ \|\ G^{\S 1.2}_{\mathsf{cb}} \qquad G^{\mathrm{v5}}_{\mathsf{acb}} = (G^{\S 1.1}_{\mathsf{ab}}\ ;\ G^{\S 1.2}_{\mathsf{cb}}) + (G^{\S 1.2}_{\mathsf{cb}}\ ;\ G^{\S 1.1}_{\mathsf{ab}})$$

- Regarding isolation of transactions, the property to be proved can be specified as follows:
  $$\varphi = \mathsf{AG}(\mathsf{AX}_{\mathsf{b.x}}(\mathsf{AU}(\mathsf{AX}_{\mathsf{b.x}}(\bot) \wedge \mathsf{AX}_{\mathsf{b.y}}(\bot), \mathsf{AX}_{\mathsf{b.y}}(\mathsf{b}.(\mathtt{md5(x)} == \mathtt{y})))))$$
  That is: in all branches, always (AG), if x is changed at Bob ($\mathsf{AX}_{\mathsf{b.x}}$), it is not changed again ($\mathsf{AX}_{\mathsf{b.x}}(\bot)$) until y is changed at Bob ($\mathsf{AX}_{\mathsf{b.y}}$) such that x and y are consistent ($\mathsf{b}.(\mathtt{md5(x)} == \mathtt{y})$). System $(\{G^{\mathrm{v1}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$ violates φ, as informally explained in §1.2. System $(\{G^{\mathrm{v2}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$ satisfies φ, as Alice and Carol acquire exclusive permission to use x and y at Bob. System $(\{G^{\mathrm{v3}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$ also satisfies φ: when the system does deadlock, it does so before x at Bob is changed; when it does not deadlock, Alice and Carol acquire exclusive permission. System $(\{G^{\mathrm{v4}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$ violates φ: while $G^{\S 1.1}_{\mathsf{ab}}$ runs as an isolated transaction (as Alice does acquire exclusive permission), $G^{\S 1.2}_{\mathsf{cb}}$ is not (as Carol does not). System $(\{G^{\mathrm{v5}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$ satisfies φ, too, but it violates operational equivalence.
- Regarding absence of deadlocks, the property to be proved can be specified as $\varphi = \mathsf{AG}(\neg \mathsf{dead})$. Systems $(\{G^{\mathrm{v1}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$, $(\{G^{\mathrm{v2}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$, $(\{G^{\mathrm{v4}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$, and $(\{G^{\mathrm{v5}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$ satisfy φ. In contrast, system $(\{G^{\mathrm{v3}}_{\mathsf{acb}}\}, \mathcal{S}, \mathcal{C})$ violates φ.

3 Conclusion

3.1 Related Work

Advances in choreographic programming were cited in §1.1. Outside choreographic programming, closest to our work are mechanisms in the literature on session types to assure mutual exclusion. In the literature on *binary* session types, mutual exclusion and related patterns are supported in the work of Balzer et al. [1] (without deadlock freedom) and by Balzer et al. and Kokke et al. [2, 36] (with deadlock freedom) in the form of typing disciplines for linear and shared channels. In the literature *multiparty* session types, mutual exclusion is supported in the work of Voinea et al. [42] in the form of a typing discipline for linear and shared channels in the special case when *multiple processes* together implement *a single role*. More generally, parallel composition has been studied in the context of multiparty session typing in several ways: through static interleaving of types (e.g., [32, 33]); through dynamic interleaving of programs (e.g., [3, 14]); through a combination of those two (e.g., in the form of nesting [9, 12]).

3.2 This Work

We presented the design of a new choreographic programming language that supports isolated transactions among overlapping sets of processes. The first idea was to track for every variable which processes are permitted to use it. The second idea was to use model checking to prove isolation. This paper is our first one in which we pursue these ideas. We believe there is plenty of room to explore alternative designs and/or refine our work as presented. Examples include new primitives in the choreographic programming language to implement programs and new modalities in the temporal logic to specify properties.

3.3 Future Work

On the theoretical side, we see three main avenues. First, we aim to extend the choreographic programming language with primitives that guarantee isolation and absence of deadlocks by construction. One possible design is a primitive of the form "**isolate** P" that implements P as an isolated transaction. The challenge is to define the operational semantics such that exclusive permission of variables is automatically acquired as late as possible, and released as soon as possible, while avoiding deadlocks (e.g., by imposing a total order on variables). Second, we aim to study an extension of our choreographic programming language with loops. Third, we aim to investigate symbolic methods to prove properties.

On the practical side, we are now developing a proof-of-concept implementation of the design in the form of a compiler from our choreographic programming language to mCRL2 [5, 15]. On input of a global program, the compiler extracts a family of local programs through projection and translates both the global program and its family to mCRL2 specifications. Using the mCRL2 toolset, we can then check properties of the global program (μ-calculus versions of our CTL formulas) and operational equivalence.

References

[1] Stephanie Balzer & Frank Pfenning (2017): *Manifest sharing with session types*. Proc. ACM Program. Lang. 1(ICFP), pp. 37:1–37:29, doi:10.1145/3110281.

[2] Stephanie Balzer, Bernardo Toninho & Frank Pfenning (2019): *Manifest Deadlock-Freedom for Shared Session Types*. In: ESOP, Lecture Notes in Computer Science 11423, Springer, pp. 611–639, doi:10.1007/978-3-030-17184-1_22.

[3] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): *Global Progress in Dynamically Interleaved Multiparty Sessions*. In: CONCUR, Lecture Notes in Computer Science 5201, Springer, pp. 418–433, doi:10.1007/978-3-540-85361-9_33.

[4] Petra van den Bos & Sung-Shik Jongmans (2023): *VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs*. In: FM, Lecture Notes in Computer Science 14000, Springer, pp. 321–339, doi:10.1007/978-3-031-27481-7_19.

[5] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability*. In: TACAS (2), Lecture Notes in Computer Science 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.

[6] Marco Carbone, Luís Cruz-Filipe, Fabrizio Montesi & Agata Murawska (2018): *Multiparty Classical Choreographies*. In: LOPSTR, Lecture Notes in Computer Science 11408, Springer, pp. 59–76, doi:10.1007/978-3-030-13838-7_4.

[7] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: ESOP, Lecture Notes in Computer Science 4421, Springer, pp. 2–17, doi:10.1007/978-3-540-71316-6_2.

[8] Marco Carbone, Kohei Honda & Nobuko Yoshida (2012): *Structured Communication-Centered Programming for Web Services*. ACM Trans. Program. Lang. Syst. 34(2), pp. 8:1–8:78, doi:10.1145/2220365.2220367.

[9] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann & Philip Wadler (2016): *Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types*. In: CONCUR, LIPIcs 59, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 33:1–33:15, doi:10.4230/LIPIcs.CONCUR.2016.33.

[10] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-freedom-by-design: multiparty asynchronous global programming*. In: POPL, ACM, pp. 263–274, doi:10.1145/2429069.2429101.

[11] Marco Carbone, Fabrizio Montesi & Carsten Schürmann (2018): *Choreographies, logically*. Distributed Comput. 31(1), pp. 51–67, doi:10.1007/s00446-017-0295-1.

[12] Marco Carbone, Fabrizio Montesi, Carsten Schürmann & Nobuko Yoshida (2017): *Multiparty session types as coherence proofs*. Acta Informatica 54(3), pp. 243–269, doi:10.1007/s00236-016-0285-y.

[13] Edmund M. Clarke, E. Allen Emerson & A. Prasad Sistla (1986): *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Trans. Program. Lang. Syst. 8(2), pp. 244–263, doi:10.1145/5397.5399.

[14] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida & Luca Padovani (2016): *Global progress for dynamically interleaved multiparty sessions*. Mathematical Structures in Computer Science 26(2), pp. 238–302, doi:10.1017/S0960129514000188.

[15] 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.

[16] Luís Cruz-Filipe, Eva Graversen, Lovro Lugovic, Fabrizio Montesi & Marco Peressotti (2022): *Functional Choreographic Programming*. In: ICTAC, Lecture Notes in Computer Science 13572, Springer, pp. 212–237, doi:10.1007/978-3-031-17715-6_15.

[17] Luís Cruz-Filipe, Kim S. Larsen & Fabrizio Montesi (2017): *The Paths to Choreography Extraction*. In: FoSSaCS, Lecture Notes in Computer Science 10203, pp. 424–440, doi:10.1007/978-3-662-54458-7_25.

[18] Luís Cruz-Filipe & Fabrizio Montesi (2016): *Choreographies in Practice*. In: FORTE, Lecture Notes in Computer Science 9688, Springer, pp. 114–123, doi:10.1007/978-3-319-39570-8_8.

[19] Luís Cruz-Filipe & Fabrizio Montesi (2017): *Encoding asynchrony in choreographies*. In: SAC, ACM, pp. 1175–1177, doi:10.1145/3019612.3019901.

[20] Luís Cruz-Filipe & Fabrizio Montesi (2017): *Procedural Choreographic Programming*. In: FORTE, Lecture Notes in Computer Science 10321, Springer, pp. 92–107, doi:10.1007/978-3-319-60225-7_7.

[21] Luís Cruz-Filipe & Fabrizio Montesi (2020): *A core model for choreographic programming*. Theor. Comput. Sci. 802, pp. 38–66, doi:10.1016/j.tcs.2019.07.005.

[22] Luís Cruz-Filipe, Fabrizio Montesi & Marco Peressotti (2018): *Communications in choreographies, revisited*. In: SAC, ACM, pp. 1248–1255, doi:10.1145/3167132.3167267.

[23] Luís Cruz-Filipe, Fabrizio Montesi & Marco Peressotti (2021): *Certifying Choreography Compilation*. In: ICTAC, Lecture Notes in Computer Science 12819, Springer, pp. 115–133, doi:10.1007/978-3-030-85315-0_8.

[24] Luís Cruz-Filipe, Fabrizio Montesi & Marco Peressotti (2021): *Formalising a Turing-Complete Choreographic Language in Coq*. In: ITP, LIPIcs 193, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 15:1–15:18, doi:10.4230/LIPIcs.ITP.2021.15.

[25] E. Allen Emerson & Edmund M. Clarke (1982): *Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*. Sci. Comput. Program. 2(3), pp. 241–266, doi:10.1016/0167-6423(83)90017-5.

[26] Saverio Giallorenzo, Fabrizio Montesi & Maurizio Gabbrielli (2018): *Applied Choreographies*. In: FORTE, Lecture Notes in Computer Science 10854, Springer, pp. 21–40, doi:10.1007/978-3-319-92612-4_2.

[27] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi & Pascal Weisenburger (2021): *Multiparty Languages: The Choreographic and Multitier Cases (Pearl)*. In: *ECOOP*, *LIPIcs* 194, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 22:1–22:27, doi:10.4230/LIPIcs.ECOOP.2021.22.

[28] Rob J. van Glabbeek & W. P. Weijland (1996): *Branching Time and Abstraction in Bisimulation Semantics*. *J. ACM* 43(3), pp. 555–600, doi:10.1145/233551.233556.

[29] Thomas T. Hildebrandt, Tijs Slaats, Hugo A. López, Søren Debois & Marco Carbone (2019): *Declarative Choreographies and Liveness*. In: *FORTE*, *Lecture Notes in Computer Science* 11535, Springer, pp. 129–147, doi:10.1007/978-3-030-21759-4_8.

[30] Andrew K. Hirsch & Deepak Garg (2022): *Pirouette: higher-order typed functional choreographies*. *Proc. ACM Program. Lang.* 6(POPL), pp. 1–27, doi:10.1145/3498684.

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

[32] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, ACM, pp. 273–284, doi:10.1145/1328438.1328472.

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

[34] Sung-Shik Jongmans & Petra van den Bos (2022): *A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming*. In: *ESOP*, *Lecture Notes in Computer Science* 13240, Springer, pp. 520–547, doi:10.1007/978-3-030-99336-8_19.

[35] Bjørn Angel Kjær, Luís Cruz-Filipe & Fabrizio Montesi (2022): *From Infinity to Choreographies - Extraction for Unbounded Systems*. In: *LOPSTR*, *Lecture Notes in Computer Science* 13474, Springer, pp. 103–120, doi:10.1007/978-3-031-16767-6_6.

[36] Wen Kokke, J. Garrett Morris & Philip Wadler (2020): *Towards Races in Linear Logic*. *Log. Methods Comput. Sci.* 16(4), doi:10.23638/LMCS-16(4:15)2020.

[37] Fabrizio Montesi (2013): *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

[38] Fabrizio Montesi & Nobuko Yoshida (2013): *Compositional Choreographies*. In: *CONCUR*, *Lecture Notes in Computer Science* 8052, Springer, pp. 425–439, doi:10.1007/978-3-642-40184-8_30.

[39] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese & Jacopo Mauro (2015): *Dynamic Choreographies - Safe Runtime Updates of Distributed Applications*. In: *COORDINATION*, *Lecture Notes in Computer Science* 9037, Springer, pp. 67–82, doi:10.1007/978-3-319-19282-6_5.

[40] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese & Jacopo Mauro (2017): *Dynamic Choreographies: Theory And Implementation*. *Log. Methods Comput. Sci.* 13(2), doi:10.1007/BF01221097.

[41] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro & Maurizio Gabbrielli (2014): *AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications*. In: *SLE*, *Lecture Notes in Computer Science* 8706, Springer, pp. 161–170, doi:10.1007/978-3-319-11245-9_9.

[42] A. Laura Voinea, Ornela Dardha & Simon J. Gay (2019): *Resource Sharing via Capability-Based Multiparty Session Types*. In: *IFM*, *Lecture Notes in Computer Science* 11918, Springer, pp. 437–455, doi:10.1007/978-3-030-34968-4_24.