Proceedings of the
Seventh Workshop on

Programming Language Approaches to Concurrency and Communication-cEntric Software

12th April 2011
Grenoble, France
http://places14.di.fc.ul.pt/

Preface

This volume contains the preliminary proceedings of PLACES 2014, the seventh Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, to be held in Grenoble, France, on April 12th 2014, and co-located with ETAPS, the European Joint Conferences on Theory and Practice of Software. The PLACES workshop series aims to offer a forum where researchers from different fields exchange new ideas on one of the central challenges for programming in the near future: the development of programming languages, methodologies and infrastructures where concurrency and distribution are the norm rather than a marginal concern. Previous editions of PLACES were held in Rome (2013), Tallin (2012), Saarbrüken (2011), Paphos (2010) and York (2009), all co-located with ETAPS, and the first PLACES was held in Oslo and co-located with DisCoTec (2008).

The Program Committee, after a careful and thorough reviewing process, selected nine papers out of 12 submissions for presentation at the workshop and inclusion in this preliminary proceedings. Each submission was evaluated by three referees (with one paper receiving a fourth review), and the accepted papers were selected during a week-long electronic discussion. One of the nine accepted papers was conditionally accepted subject to a process of shepherding by a PC member, which was successful and led to the paper's full acceptance.

In addition to the contributed papers, the workshop will feature an invited talk by Akash Lal, Microsoft Research India, entitled "Finding Concurrency Bugs Under Imprecise Harnesses". The abstract for this talk appears in these proceedings.

PLACES 2014 was made possible by the contribution and dedication of many people. We thank all the authors who submitted papers for consideration. Thanks also to our invited speaker, Akash Lal. We are extremely grateful to the members of the Program Committee and additional experts for their careful reviews, and the balanced discussions during the selection process. The EasyChair system was instrumental in supporting the submission and review process and the production of these proceedings.

February 12th, 2014
Alastair F. Donaldson
Vasco T. Vasconcelos

Program Committee

Jade Alglave	University College London, UK
Michele Bugliesi	Università Ca' Foscari Venezia, Italy
Alastair Donaldson	Imperial College London, UK
Benedict Gaster	Qualcomm, USA
Simon Gay	University of Glasgow, UK
Thomas Hildebrandt	IT University of Copenhagen, Denmark
Jeroen Ketema	Imperial College London, UK
Hans-Wolfgang Loidl	Heriot Watt University, UK
Étienne Lozes	ENS Cachan, France, and University of Kassel, Germany
Francisco Martins	University of Lisbon, Portugal
Alan Mycroft	University of Cambridge, UK
Luca Padovani	Università di Torino, Italy
Shaz Qadeer	Microsoft Research, USA
Vasco T. Vasconcelos	University of Lisbon, Portugal

Additional Reviewers

Gabrielle Anderson
Stefano Calzavara
Tiago Cogumbreiro
Søren Debois
Dimitris, Mostrous

Table of Contents

Finding Concurrency Bugs Under Imprecise Harnesses (Invited talk) . . 1
AkashLal . . . . . .
Multiparty Sessions based on Proof Nets 2
Dimitris Mostrous . . . . . .
Sessions as propositions 9
Sam Lindley and J. Garrett Morris . . . . . .
Towards Reversible Sessions 16
Francesco Tiezzi and Nobuko Yoshida . . . . . .
Session Types for Broadcasting 23
Dimitrios Kouzapas, Ramunas Gutkovas and Simon Gay . . . . .
Multiparty Session Actors 30
Rumyana Neykova and Nobuko Yoshida . . . . . .
Lightening Global Types 36
Tzu-Chun Chen . . . . . .
Verifying Parallel Loops with Separation Logic 43
Stefan Blom, Saeed Darabi and Marieke Huisman . . . . . .
Towards Composable Concurrency Abstractions 50
Janwillem Swalens, Stefan Marr, Joeri De Koster and Tom Van Cutsem . . . . . .
Session Type Isomorphisms 57
Mariangiola Dezani-Ciancaglini, Luca Padovani and Jovanka Pantovic . . . . . .

Invited Talk
Finding concurrency bugs under imprecise harnesses

Akash Lal
Microsoft Research India

One way of scaling verification technology to real software is to reduce the code size under analysis. Instead of applying verification to the entire system, one can pick a single component and apply verification to it. While this can dramatically help reduce analysis time, it can lead to false alarms when, for instance, the test harness does not set up the initial state properly before calling into the component.

In this work, we explore the possibility of automatically filtering away (or prioritizing) warnings that result from imprecision in the harness. We limit our attention to the scenario when one is interested in finding bugs due to concurrency. We define a warning to be an interleaved bug when it manifests on an input for which no sequential interleaving produces a warning. We argue that limiting a static analysis to only consider interleaved bugs greatly reduces false positives during static concurrency analysis in the presence of an imprecise harness.

Biography

Akash Lal is a researcher at Microsoft Research, Bangalore. His interests are in the area of programming languages and program analysis, with a focus on building bug-finding tools for concurrent programs. He joined Microsoft in 2009 after completing his PhD from University of Wisconsin-Madison under the supervision of Prof. Thomas Reps. For his thesis, he received the Outstanding Graduate Researcher Award, given by the Computer Sciences Department of UW-Madison, and the ACM SIGPLAN Outstanding Doctoral Dissertation Award. He completed his Bachelor's degree from IIT-Delhi in 2003.

Multiparty Sessions based on Proof Nets

Dimitris Mostrous

LaSIGE, Department of Informatics, University of Lisbon, Portugal. dimitris@di.fc.ul.pt

1 Introduction

Since their inception, sessions [12, 19] and multiparty sessions [11] have been gaining momentum as a very useful foundation for the description and verification of *structured interactions*. Interestingly, recent works have established a close correspondence between typed, synchronous pi-calculus processes and sequent proofs of a variation of Intuitionistic Linear Logic [3]. This particular interpretation of Linear proofs is considered a sessions system because it has (for practical purposes) the same type contructors but with a clear logical motivation. In this paper we outline a system based on an interpretation of the proof objects of Classical Linear Logic, namely Proof Nets [8], improving our previous work [16]. The process language resembles Solos [14] and exhibits asynchrony in both input and output. Proof Nets have a number of advantages over sequent proofs, such as increased potential for parallelism and a very appealing graphical notation that could be seen as a new kind of *global type* [11]. Nevertheless, accurate logical interpretations are typically deterministic, which limits their applicability to concurrent programming. However, with a very modest adaptation that enables *synchronisation*, non-deterministic behaviours can be allowed without compromising the basic properties of interest, namely strong normalisation and deadlock-freedom.

Let us distinguish *multiparty behaviours* and the *multiparty session types* (global types) of [11]. A multiparty behaviour emerges when more than two processes can be part of the same session, and this is achieved at the operational level by a synchronisation mechanism such as the multicast request $\overline{a}[\mathsf{2..n}](\vec{s}).P$ [11]. We propose a similar mechanism in the form of replications with synchronisation, $!a_1(x_1)\cdots a_n(x_n).P$, which allow a service to be activated with multiple parties. A global type captures the interactions and sequencing constraints of the complete protocol of a program. In our proposal, the equivalent to a global type is the *proof net* of the program. Although our approach is technically very different, we believe that the logical foundations and simpler meta-theory are appealing. We show how pi-calculus channels with $\mathsf{i/o}$ type and a multi-party interaction from [11] can be encoded.

2 The Process Interpretation

Syntax The language is inspired by proof nets except that connectives have explicit locations (names). Types are ranged over by A, B, C, with type variables ranging over X, Y. We assume a countable set of *names*, ranged over by a, b, c, x, y, z, r, k. Then, $\widetilde{b}$ stands for a sequence $b_1, \ldots, b_n$ of length $|\widetilde{b}| = n$, and similarly for types. Processes, P, Q, R, are defined as follows:

$$\begin{array}{rcllllll} P & ::= & \overline{a}(\widetilde{A},\widetilde{x}) & \mid\ a(\widetilde{\mathsf{X}},\widetilde{y}) & \mid\ \overline{a} \triangleleft \boldsymbol{i}(b) & \mid\ a \triangleright \{\boldsymbol{i}(x_i{:}A_i).P_i\}_{i\in I} & \mid\ ?\,\overline{a}(b) & \mid\ !a_i(x_i{:}A_i)_{i\in I}.P \\ & \mid & ab & \mid\ \mathsf{X} \mapsto A & \mid\ (\boldsymbol{\nu} a{:}A)P & \mid\ (\boldsymbol{\nu}\mathsf{X})P & \mid\ (P \mid Q) & \mid\ \mathbf{0} \end{array}$$

There are two kinds of *solos*-like [14] communication devices: $a(\widetilde{\mathsf{X}},\widetilde{y})$ and $\overline{a}(\widetilde{A},\widetilde{x})$. In typed processes, we will be using the nullary signals a for $a()$ and $\overline{a}$ for $\overline{a}()$,[1] the binary input $a(b,c)$ (resp.

[1] They have no computational content, but without them reduction leaves garbage axioms of unit type.

output $\overline{a}(b,c)$) and the *asynchronous* polymorphic input $a(\mathsf{X},b)$ (resp. output $\overline{a}(B,b)$). The explicit substitution, ab, interprets Linear Logic axioms; this is standard in related works [16, 2]. The type alias $\mathsf{X} \mapsto A$ is simply a typing device, and the scope of X is restricted with $(\boldsymbol{\nu}\mathsf{X})P$. The branching connective $a \triangleright \{\boldsymbol{i}(x_i{:}A_i).P_i\}_{i\in I}$, with $I = \{1,\ldots,n\}$, written also in the form $a \triangleright \{\mathbf{1}(x_1{:}A_1).P_1 \,[]\, \cdots \,[]\, \boldsymbol{n}(x_n{:}A_n).P_n\}$, offers an indexed sequence of alternative behaviours. One of these can be selected using $\overline{a} \triangleleft \boldsymbol{k}(b)$ with $k \in I$. Our notion of replication enables synchronisation, similarly to a multiparty "accept" (cf. [11]). The notation is $!a_i(x_i{:}A_i)_{i\in I}.P$, written also as $!a_1(x_1{:}A_1)\cdots a_n(x_n{:}A_n).P$ with $n \geq 1$. Dually, $?\overline{a}(b)$ can be thought as a "request."

Free, passive, and active names The *free names* ($\mathsf{fn}(P)$) are defined in the standard way. We just note that the only bound names are a in $(\boldsymbol{\nu}a{:}A)\,P$ and the x_i in $a \triangleright \{\boldsymbol{i}(x_i{:}A_i).P_i\}_{i\in I}$ and $!a_i(x_i{:}A_i)_{i\in I}.P$. The *passive names* ($\mathsf{pn}(P)$) are defined similarly to $\mathsf{fn}(P)$ except for:

$$\mathsf{pn}(a(\widetilde{\mathsf{X}},\widetilde{x})) = \mathsf{pn}(\overline{a}(\widetilde{A},\widetilde{x})) = \{\widetilde{x}\} \qquad \mathsf{pn}(\overline{a} \triangleleft \boldsymbol{i}(b)) = \mathsf{pn}(?\overline{a}(b)) = \{b\} \qquad \mathsf{pn}(ab) = \emptyset$$
$$\mathsf{pn}(a \triangleright \{\boldsymbol{i}(x_i{:}A_i).P_i\}_{i\in I}) = \cup_{i\in I}(\mathsf{pn}(P_i) \setminus \{x_i\}) \qquad \mathsf{pn}(!a_i(x_i{:}A_i)_{i\in I}.P) = \mathsf{pn}(P) \setminus \cup_{i\in I}\{x_i\}$$

The *active names* ($\mathsf{an}(P)$) are defined by $\mathsf{an}(P) = \mathsf{fn}(P) \setminus \mathsf{pn}(P)$. For example, y in $(\boldsymbol{\nu}x)(\, a(x,y) \mid xy\,)$ is not active. (As usual, we assume the name convention.)

Structure Equivalence With $\equiv$ we denote the least congruence on processes that is an equivalence relation, equates processes up to α-conversion, satisfies the abelian monoid laws for parallel composition, the usual laws for scope extrusion, and satisfies the following axioms:[2]

$$(\boldsymbol{\nu}\mathsf{X})\mathbf{0} \equiv \mathbf{0} \qquad (\boldsymbol{\nu}\mathsf{X})P \mid Q \equiv (\boldsymbol{\nu}\mathsf{X})(\, P \mid Q\,) \quad (\mathsf{X} \notin \mathsf{ftv}(Q))$$
$$(\boldsymbol{\nu}\mathsf{X})(\boldsymbol{\nu}\mathsf{Y})P \equiv (\boldsymbol{\nu}\mathsf{Y})(\boldsymbol{\nu}\mathsf{X})P \qquad (\boldsymbol{\nu}\mathsf{X})(\boldsymbol{\nu}a{:}A)P \equiv (\boldsymbol{\nu}a{:}A)(\boldsymbol{\nu}\mathsf{X})P \quad (\mathsf{X} \notin \mathsf{ftv}(A))$$
$$ab \equiv ba \qquad ab \mid !a_1(x_1{:}A_1)\cdots a(x{:}A)\cdots a_n(x_n{:}A_n).P \equiv ab \mid !a_1(x_1{:}A_1)\cdots b(x{:}A)\cdots a_n(x_n{:}A_n).P$$

The most notable axiom is the last one, which effects a forwarding, *e.g.*, $?\overline{a}(x) \mid ab \mid !b(y).P \equiv^2 ab \mid ?\overline{a}(x) \mid !a(y).P$. As can be seen next, the term on the right can now reduce.

Reduction "$\longrightarrow$" is the smallest binary relation on terms such that:

$$\overline{a}(\widetilde{A},\widetilde{y}) \mid a(\widetilde{\mathsf{X}},\widetilde{x}) \;\longrightarrow\; \widetilde{\mathsf{X} \mapsto A} \mid \widetilde{xy} \qquad |\widetilde{A}| = |\widetilde{\mathsf{X}}|,\ |\widetilde{x}| = |\widetilde{y}| \qquad \text{(R-Com)}$$
$$\overline{a} \triangleleft \boldsymbol{k}(b) \mid a \triangleright \{\boldsymbol{i}(x_i{:}A_i).P_i\}_{i\in I} \;\longrightarrow\; P_k\{{}^{b}/_{x_k}\} \qquad k \in I \qquad \text{(R-Sel)}$$
$$\textstyle\prod_{i\in I} ?\overline{a}_i(b_i) \mid !a_i(x_i{:}A_i)_{i\in I}.P \;\longrightarrow\; P\{{}^{b_i}/_{x_i}\}_{i\in I} \mid !a_i(x_i{:}A_i)_{i\in I}.P \qquad \text{(R-Sync)}$$
$$(\boldsymbol{\nu}a{:}A)(\, ab \mid P\,) \;\longrightarrow\; P\{{}^{b}/_{a}\} \qquad a \neq b,\ a \in \mathsf{an}(P) \qquad \text{(R-Ax)}$$
$$P \equiv P' \longrightarrow Q' \equiv Q \;\Rightarrow\; P \longrightarrow Q \quad \text{(R-Str)} \qquad P \longrightarrow Q \;\Rightarrow\; \mathsf{C}[\,P\,] \longrightarrow \mathsf{C}[\,Q\,] \quad \text{(R-Ctx)}$$
$$\textit{Contexts in } \text{(R-Ctx)}: \qquad \mathsf{C}[\,\cdot\,] ::= \cdot \mid (\,\mathsf{C}[\,\cdot\,] \mid P\,) \mid (\boldsymbol{\nu}a{:}\,A)\,\mathsf{C}[\,\cdot\,] \mid (\boldsymbol{\nu}\mathsf{X})\,\mathsf{C}[\,\cdot\,]$$

(R-Com) resembles solos reduction [14] but with explicit fusions [7, 16]. Specifically, given two vectors $\widetilde{x}$ and $\widetilde{y}$ of length n, the notation $\widetilde{xy}$ stands for $x_1y_1 \mid \cdots \mid x_ny_n$ or $\mathbf{0}$ if the vectors are empty. For polymorphism we create type aliases: $\widetilde{\mathsf{X} \mapsto A}$ stands for $\mathsf{X}_1 \mapsto A_1 \mid \cdots \mid \mathsf{X}_n \mapsto A_n$. Combined type and name communication appears also in a synchronous setting [18]. (R-Sel) is standard. In (R-Sync) we synchronise on all a_i, obtaining a form of multi-party session against $?\overline{a}_1(b_1) \mid \cdots \mid ?\overline{a}_n(b_n)$. (R-Ax) effects a capture-avoiding name substitution, defined in the standard way. The side-condition $a \neq b$ guarantees that no bound name becomes free; $a \in \mathsf{an}(P)$ ensures that the cut is applied correctly, that is, against two (or more) conclusions.

[2] The free type variables ($\mathsf{ftv}(P)$) are defined in a standard way, noting that $\mathsf{ftv}((\boldsymbol{\nu}\mathsf{X})P) = \mathsf{ftv}(P) \setminus \mathsf{X}$. The free type variables of a type A ($\mathsf{ftv}(A)$) are also standard and arise from $\forall/\exists$.

The Caires-Pfenning axiom reduction (R-Ax) is based on $(\boldsymbol{\nu} a)(\,[a \leftrightarrow b] \mid P\,) \longrightarrow P\{{}^{b}/_{a}\}(a \neq b)$ from [17], which is similar to the "Cleanup" rule of [1]. However, in an asynchronous language, this rule breaks subject reduction, which motivates our side-condition $a \in \mathsf{an}(P)$. For example, $Q \doteq (\boldsymbol{\nu} x, y)(\,\overline{a}\langle x, y\rangle \mid [x \leftrightarrow b] \mid [y \leftrightarrow c]\,)$ is typable in the system of [5], with conclusion $b\colon A, c\colon B \vdash Q :: a\colon A \otimes B$, but it reduces to $(\boldsymbol{\nu} y)(\,\overline{a}\langle b, y\rangle \mid [y \leftrightarrow c]\,)$ which is not typable.[3]

Types and duality The types, ranged over by $A, B, C, D\ldots$, are linear logic formulae [8]:

$$A ::= \mathbf{1} \mid \bot \mid A \otimes B \mid A \parr B \mid A \,\&\, B \mid A \oplus B \mid !_{\mathfrak{m}} A \mid ?_{\mathfrak{m}} A \mid \forall \mathsf{X}.A \mid \exists \mathsf{X}.A \mid \mathsf{X} \mid {\sim}\mathsf{X}$$

The *mode* $\mathfrak{m}$ can be $\boldsymbol{\varepsilon}$ (empty) or $\star$ (synchronising): $\boldsymbol{\varepsilon}$ is a formality and is never shown; $\star$ is used to enforce some restrictions, but does not generally alter the meaning of types. Negation ${\sim}(\,\cdot\,)$, which corresponds to duality, is an involution on types (${\sim}({\sim}A) = A$) defined in the usual way (we use the notation from [10]):

$$\begin{gathered}
{\sim}\mathbf{1} \doteq \bot \qquad {\sim}\bot \doteq \mathbf{1} \qquad {\sim}(A \otimes B) \doteq {\sim}A \parr {\sim}B \qquad {\sim}(A \parr B) \doteq {\sim}A \otimes {\sim}B \\
{\sim}(A \,\&\, B) \doteq {\sim}A \oplus {\sim}B \qquad {\sim}(A \oplus B) \doteq {\sim}A \,\&\, {\sim}B \qquad {\sim}(!_{\mathfrak{m}} A) \doteq ?_{\mathfrak{m}}{\sim}A \qquad {\sim}(?_{\mathfrak{m}} A) \doteq !_{\mathfrak{m}}{\sim}A \\
{\sim}(\forall \mathsf{X}.A) \doteq \exists \mathsf{X}.{\sim}A \qquad {\sim}(\exists \mathsf{X}.A) \doteq \forall \mathsf{X}.{\sim}A \qquad {\sim}({\sim}\mathsf{X}) \doteq \mathsf{X}
\end{gathered}$$

The multiplicative conjunction $A \otimes B$ (with unit $\mathbf{1}$) is the type of a channel that communicates a name of type A and a name of type B, offered by disconnected terms; it can be thought as an "output." The multiplicative disjunction $A \parr B$ (with unit $\bot$) is only different in that the communicated names can be offered by one term; this possibility of dependency makes it an "input." In a standard way, the additive conjunction $A \,\&\, B$ is an external choice (branching), and dually additive disjunction $A \oplus B$ is an internal choice (selection). Ignoring modes, the exponential types $!A$ and $?A$ can be understood as a decomposition of the "shared" type in sessions: $!A$ is assigned to a persistent term that offers A; dually, $?A$ can be assigned to any name with type A so that it can communicate with $!{\sim}A$. The second-order types $\forall \mathsf{X}.A$ and $\exists \mathsf{X}.A$ are standard, as is type substitution: $A[{}^{B}/_{\mathsf{X}}]$ stands for A with B for X, and ${\sim}B$ for ${\sim}\mathsf{X}$.

Judgements and interfaces A judgement $P \triangleright \Gamma$ denotes that term P can be assigned the *interface* Γ. Interfaces, ranging over Γ, Δ, are sequences with possible repetition, defined by:

$$\Gamma \quad ::= \quad \emptyset \quad \mid \quad \Gamma, a\colon A \quad \mid \quad \Gamma, [a\colon A] \quad \mid \quad \Gamma, \mathsf{X}$$

$a\colon A$ is standard. A *discharged occurrence* $[a\colon A]$ indicates that a has been used as A: it serves to protect linearity, since a can no longer be used. Γ, X records that X appears free in the term, ensuring freshness of type variables. $\widetilde{a}\colon \widetilde{A}$ stands for $a_1\colon A_1, \ldots, a_n\colon A_n$. $?\Gamma$ stands for $\widetilde{a}\colon \widetilde{?A}$, i.e., $a_1\colon ?A_1, \ldots, a_n\colon ?A_n$. Similarly, $[\Gamma]$ means $[\widetilde{a}\colon \widetilde{A}]$. Let $\mathsf{fn}(a\colon A) = \mathsf{fn}([a\colon A]) = a$ and $\mathsf{fn}(\mathsf{X}) = \emptyset$, plus the obvious definition for free type variables ($\mathsf{ftv}(\Gamma)$). We consider *well-formed interfaces*, in which only $a\colon ?A$ can appear multiple times, but $a\colon ?_{\star}A$ cannot. Moreover, (Γ, X) is well-formed when Γ is well-formed and $\exists \sigma.\sigma(\Gamma) = [\Delta], \Sigma$ such that $\mathsf{X} \notin \mathsf{ftv}(\Sigma)$. For example in the ($\forall$) rule the conclusion is $\Gamma, [x\colon A], \mathsf{X}, a\colon \forall \mathsf{X}.A$ with X possibly free in A.

Subtyping The usual structural rules of Linear Logic are incorporated into the relation $\Gamma \preccurlyeq \Delta$:

$$\Gamma, [a\colon A] \preccurlyeq \Gamma, [a\colon {\sim}A] \qquad \Gamma \preccurlyeq \sigma(\Gamma) \qquad \Gamma, a\colon ?_{\mathfrak{m}} A \preccurlyeq \Gamma$$

$$\Gamma, a\colon ?A \preccurlyeq \Gamma, a\colon ?A, a\colon ?A \qquad \Gamma, a\colon ?A \preccurlyeq \Gamma, a\colon ?_{\star}A \qquad \Gamma, a\colon !_{\star}A \preccurlyeq \Gamma, a\colon !A$$

The first rule identifies the type of a discharged occurrence and its dual, matching a type annotation which may be $(\boldsymbol{\nu} a\colon A)$ or $(\boldsymbol{\nu} a\colon {\sim}A)$. Then we have *exchange*, *weakening*, *contraction*. The last two axioms alter the mode: we can forget $\star$ in $?_{\star}A$, and dually we can record it on $!A$.

[3] The reduction rule is not mentioned in [5], but the type rule is given and one of the authors relayed to me that reduction is assumed to be the same as in [17].

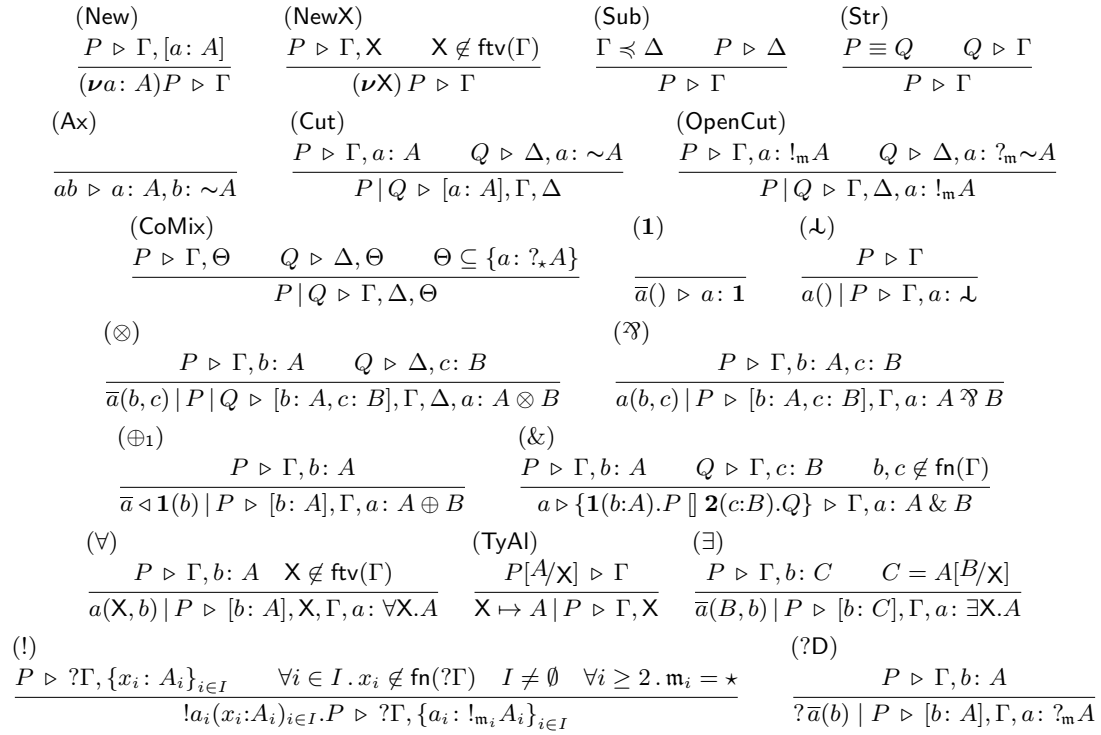


Figure 1: Linear Logic Typing with Multiparty Promotion

Typing rules can be found in Fig. 1. We type modulo structure equivalence, a possibility suggested by [15] and used in [3]. This is because associativity of " | " does not preserve typability, i.e., a cut between P and $(Q \,|\, R)$ may be untypable as $(P \,|\, Q) \,|\, R$; $(\boldsymbol{\nu} a)$ causes similar problems.

In (Cut) the name a is discharged and can then be closed with (New). (OpenCut) was added for two reasons. First, it is intuitive, since we are not required to close the name, i.e., to fix the number of clients of $!A$, departing from the de facto interpretation of "cut as composition under name restriction." Second, it is needed for soundness. Take $R \doteq (\boldsymbol{\nu} a\colon !A)(\, ab \mid ?\overline{a}(x) \mid P \mid !a(y).Q\,)$ typed with $R \triangleright \Gamma, b\colon !A$. Using (R-Ax) we obtain $R \longrightarrow ?\overline{b}(x) \mid P\{{}^{b}/_{a}\} \mid !b(y).Q$, which is *only* typable with the same interface by using (OpenCut); with (Cut) we obtain $\Gamma, [b\colon !A]$.[4]

Asynchronous messages can encode standard sessions (see [4, 5]): $\overline{a}(b,c)$ with type $A \otimes B$ maps to the session type $!_{\mathsf{s}}{\sim}A.B$ *or* $!_{\mathsf{s}}{\sim}B.A$. Dually, $a(x,y)$ with ${\sim}A \parr {\sim}B$ maps to $?_{\mathsf{s}}{\sim}A.{\sim}B$ *or* $?_{\mathsf{s}}{\sim}B.{\sim}A$. To write processes in standard sessions style, with reuse of names (*e.g.*, $\overline{a}b; a(x); \mathbf{0}$), we introduce abbreviations that use the second component for a's continuation:

$$\begin{array}{rcl@{\qquad}rcl}
\overline{a}b; P & \doteq & (\boldsymbol{\nu} x,y)(\,\overline{a}(x,y) \mid xb \mid P\{{}^{y}/_{a}\}\,) & a(x); P & \doteq & (\boldsymbol{\nu} x,y)(\,a(x,y) \mid P\{{}^{y}/_{a}\}\,) \\
a \lhd l_k; P & \doteq & (\boldsymbol{\nu} x)(\,\overline{a} \lhd \boldsymbol{k}(x) \mid P\{{}^{x}/_{a}\}\,) & a \rhd \{l_i.P_i\}_{i\in I} & \doteq & a \rhd \{\boldsymbol{i}(x_i).P_i\{{}^{x_i}/_{a}\}\}_{i\in I} \\
\overline{a}B; P & \doteq & (\boldsymbol{\nu} x)(\,\overline{a}(B,x) \mid P\{{}^{x}/_{a}\}\,) & a(\mathsf{X}); P & \doteq & (\boldsymbol{\nu} \mathsf{X},x)(\,a(\mathsf{X},x) \mid P\{{}^{x}/_{a}\}\,)
\end{array}$$

It is easy to check that linear redices commute with all other redices, and therefore a "real"

[4] Several works [17, 5, 20, 2] would not enjoy subject reduction if this example could be transferred: their cut rule requires $(\boldsymbol{\nu} b)(\cdots)$, which is here missing. These works don't have "Mix" (here: (CoMix)), which we used in the example; but this should be checked, since "Mix" can be encoded with a new conclusion $c\colon \bot \otimes \bot$ [8, p. 100].

prefix would not have any effect on computation except to make it more sequential.

The rule (!) implements an extension of the logic:

$$\frac{?\Gamma, A}{?\Gamma, !A}\ \textit{(promotion)} \qquad \textbf{becomes} \qquad \frac{?\Gamma, A_1, \ldots, A_n}{?\Gamma, !A_1, \ldots, !A_n}\ \textit{(multi-promotion)}$$

Actually we need to employ some restrictions on this rule, which is why all conclusions except the first must have a $\star$-mode. Since there is no contraction for $?_\star$,[5] all the $a_i : ?_\star{\sim}A_i$ $(i \geq 2)$ will come from terms with just one call to the session.[6] The first conclusion, $a_1 : !_{\mathfrak{m}_1} A_1$, can have standard mode ($\mathfrak{m}_1 = \varepsilon$), which allows a client's call with $a_1 : ?{\sim}A_1$ to be connected to (i.e., to depend on) other calls on a_1. In this way we provide a *hook* for one client to participate in another instance of the same session, and this facilitates a form of *dynamic join*. We return to this concept in the first example.

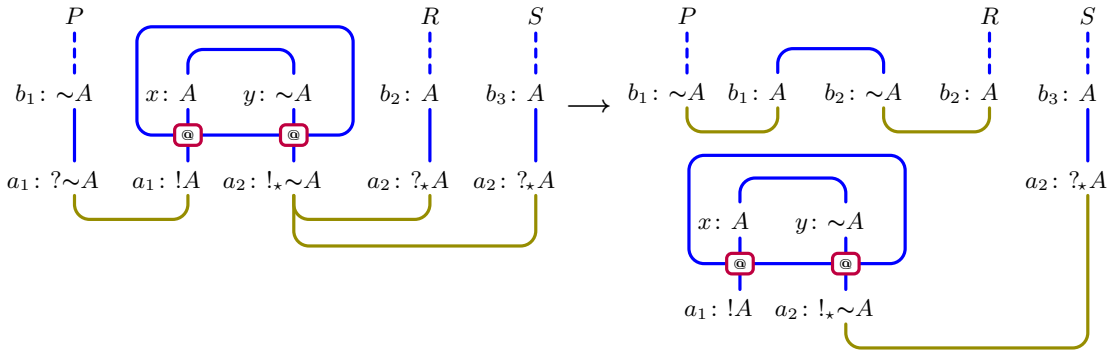
Finally, the sidecondition in $(\&, !)$ forbids premises from having multiple copies of a name (*e.g.*, $x : ?A, x : ?A$) which should be removed in the conclusion; essentially it forces contractions (by $\preccurlyeq$). Other rules are immune by the well-formedness of $\Gamma, [a : A]$[7].

Expresiveness & Properties The system is an extension of proof nets, in process form, so it can encode System F, inductive sessions (using second-order features), etc. Due to space limitations we only show two examples: (a) how shared channels can be simulated with synchronisation; (b) the (two Buyer, one Seller) protocol from [11].

a) channels Non-determinism can be expressed by sharing a channel between multiple competing processes trying to send and receive messages. This is impossible with existing logical sessions systems, and more generally if we follow the logic "by the book." A channel a with i/o type $(A, {\sim}A)$, i.e., that exports two complementary capabilities A and ${\sim}A$, can be encoded by the two names a_1 and a_2 in $!a_1(x{:}A)\, a_2(y{:}{\sim}A).xy \triangleright a_1 : !A, a_2 : !_\star{\sim}A$. The channel is used by terms with $a_1 : ?{\sim}A$ or $a_2 : ?_\star A$, and there can be multiple instances of each, giving rise to critical (non-deterministic) pairs. Moreover, A can be linear, i.e., we can communicate linear values through shared channels, which is a novel feature. For example:

$$\begin{aligned} &?\overline{a}_1(b_1) \mid ?\overline{a}_2(b_2) \mid ?\overline{a}_2(b_3) \mid !a_1(x{:}A)\, a_2(y{:}{\sim}A).xy \mid P \mid R \mid S \\ &\longrightarrow^{(a)} b_1 b_2 \mid ?\overline{a}_2(b_3) \mid !a_1(x{:}A)\, a_2(y{:}{\sim}A).xy \mid P \mid R \mid S \\ &\longrightarrow^{(b)} b_1 b_3 \mid ?\overline{a}_2(b_2) \mid !a_1(x{:}A)\, a_2(y{:}{\sim}A).xy \mid P \mid R \mid S \end{aligned}$$

First, note that confluence is lost: assume P, R, S cannot reduce and it becomes obvious. The graphical notation with a reduction of the first possibility is depicted below.



[5] More accurately: contraction of $?_\star{\sim}A_i$ is *multiplicative*.

[6] In the sense that two calls can never depend on each other.

[7] It is subtle but due to the variable convention, $(\&)$ is actually immune too; the condition serves for clarity.

It is possible that P has another call to a_1, but by the restriction on $?_\star$-types there cannot be a "trip" from b_2 to a_2, as this would lead to a cycle. Concretely, if R has another call to a_2, then it is from a part disconnected to b_2, and similarly for S; see (CoMix).[8]

b) multiparty interactions The (two Buyer, one Seller) protocol from [11] is shown below, with insignificant adaptations, using the previously explained abbreviations (we omit some signals for $\mathbf{1}/\bot$):

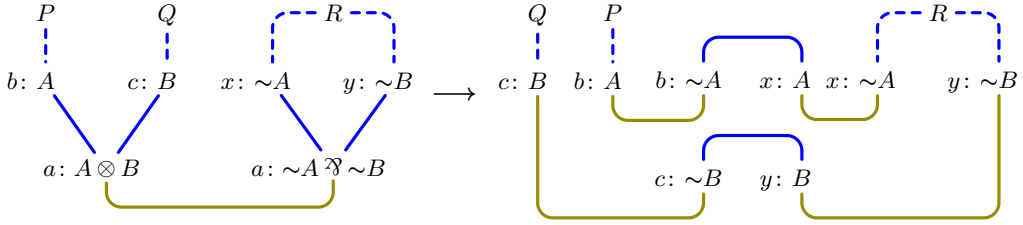
$$\begin{array}{lcl}
\mathsf{Buyer1} & \doteq & (\boldsymbol{\nu} b_1)\left(?\,\overline{a}_1(b_1) \mid \overline{b_1}\text{“The Art of War”}; b_1(\mathit{quote}); (\boldsymbol{\nu} z)(\,\overline{b_1} z; \mathbf{0} \mid \overline{z}\,\mathit{quote}/2; P_1\,)\right) \\
\mathsf{Buyer2} & \doteq & (\boldsymbol{\nu} b_2)\left(?\,\overline{a}_2(b_2) \mid b_2(\mathit{quote}); b_2(z); z(\mathit{contrib}); b_2 \triangleleft \mathsf{ok}; \overline{b_2}\text{“SW12 3AZ”}; b_2(\mathit{date}); P_2\right) \\
\mathsf{Seller} & \doteq & !a_1(x_1)\,a_2(x_2).\left(\begin{array}{l} x_1(\mathit{title}); \overline{x_1}\,20\,€; \overline{x_2}\,20\,€; x_1(z); \overline{x_2} z; \\ \qquad x_2 \triangleright \{\mathsf{ok}.x_2(\mathit{address}); \overline{x_2}\text{“7/Feb”}; Q \,[]\, \mathsf{quit}.\mathbf{0}\}\end{array}\right)
\end{array}$$

We note that the simplicity of the example has not been sacrificed, compared to the code in [11]. One difference is that we passed z from Buyer1 to Buyer2 *through* Seller using $x_1(z); \overline{x_2} z$, when in [11] all names are known to all participants. We do not employ the *global types* of [11], but there is a *proof net* for Buyer1 | Buyer2 | Seller, not shown due to space contraints, and we postulate that:

The proof net can serve as an alternative notion of global type.

Outline of results The expected soundness result for reduction, $P \triangleright \Gamma$ and $P \longrightarrow P'$ implies $P' \triangleright \Gamma$, is obtained in a standard way, but fails without the $\star$-mode. Strong Normalisation (**sN**), i.e., $P \triangleright \Gamma$ implies that *all* reduction sequences from P are finite, is shown by an adaptation of the reducibility candidates technique from [8]. The loss of confluence complicates the proof, which is in fact obtained for an extended (confluent) reduction relation using a technique of [6], from which we derive as a corollary the result. For **sN** we prove the (initially) stronger property of *reducibility* [8], which can also serve as a very strong progress guarantee.

A Curry-Howard correspondence can be obtained easily for a fragment of the language. For the multiplicative, additive, and second-order cut-elimination we only need to perform extra axiom cuts (i.e., substitutions). For exponentials, we restrict replications to a single input and simulate the actual copying (with contraction links) that takes place in proof nets with sharing and sequentialised cut-elimination steps. Indeed, there is still a loss of parallelism compared to standard proof nets, but the term language is more realistic. We show just one case of cut-elimination, the cut $(\otimes \text{—} \parr)$, implemented by $\overline{a}(b,c) \mid a(x,y) \longrightarrow bx \mid cy$, adding appropriate contexts (P, Q, R):



3 Conclusion

We claim that our language is simpler and proof-theoretically more appealing than related works such as [3]: structured interactions take place as expected (*fidelity*), but parallelism is not inhibited by the use of prefix, which cannot anyway alter the result in a deterministic setting. It is really a question of proof nets vs. sequent proofs, and in logic the first are almost always preferable. Even with synchronisation and the induced non-determinism, the system we propose retains good properties, for example it seems to be the first approach to multiparty behaviours that enjoys strong normalisation. Finally, our notion of *proof net as global type* seems to be a reasonable solution for logically founded multiparty sessions.

[8] In general, derelictions can be connected; try with $a\colon ?(A \oplus {\sim}A)$.

In relation to Abramsky's interpretation [1], it is close to proof nets *with boxes*, i.e., to a completely synchronous calculus. Moreover, it is not so friendly syntactically, it does not have a notion of bound name, copying of exponentials is explicit (no *sharing*), and of course it is completely deterministic.

An interesting future direction would be to obtain a *light* variation of our system, *e.g.*, following [9]. Then we could speak of implicit complexity for multiparty sessions, similarly to what has been done in [13] for binary sessions. Due to space restrictions, more examples and all proofs have been omitted. These will appear in a longer version, see `http://www.di.fc.ul.pt/~dimitris/`.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. ESOP'13, pages 330–349, Berlin, Heidelberg, 2013. Springer-Verlag.

[3] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR'10*. Springer-Verlag, 2010.

[4] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 280–296, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] H. DeYoung, L. Caires, F. Pfenning, and B. Toninho. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *CSL'12*, 2012.

[6] Thomas Ehrhard and Olivier Laurent. Interpreting a finitary pi-calculus in differential interaction nets. *Information and Computation*, 2010.

[7] Philippa Gardner, Cosimo Laneve, and Lucian Wischik. Linear forwarders. *Information and Computation*, 205(10):1526–1550, 2007.

[8] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[9] Jean-Yves Girard. Light linear logic, 1995.

[10] Jean-Yves Girard. *The Blind Spot*. European Mathematical Society, 2011.

[11] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *POPL*, 43(1):273–284, 2008.

[12] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, 1998.

[13] Ugo Dal Lago and Paolo Di Giamberardino. Soft session types. In Bas Luttik and Frank Valencia, editors, *EXPRESS*, volume 64 of *EPTCS*, pages 59–73, 2011.

[14] Cosimo Laneve and Björn Victor. Solos in concert. *Mathematical Structures in Computer Science*, 13(5):657–683, October 2003.

[15] Robin Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.

[16] Dimitris Mostrous. Proof nets in process algebraic form. Available at `http://www.di.fc.ul.pt/~dimitris/`, April 2012.

[17] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In *ESOP '12*, 2012.

[18] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, May 2000.

[19] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE '94*, pages 398–413. Springer-Verlag, 1994.

[20] Philip Wadler. Propositions as sessions. In *Proceedings of the International Conference on Functional Programming (ICFP '12)*. ACM, 2012.

Sessions as propositions

Sam Lindley and J. Garrett Morris

The University of Edinburgh
{Sam.Lindley,Garrett.Morris}@ed.ac.uk

Abstract

Recently, Wadler presented a continuation-passing translation from a session-typed functional language, GV, to a process calculus based on classical linear logic, CP. However, this translation is one-way: CP is more expressive than GV. We propose an extension of GV, called HGV, and give translations showing that it is as expressive as CP. The new translations shed light both on the original translation from GV to CP, and on the limitations in expressiveness of GV.

1 Introduction

Linear logic has long been regarded as a potential typing discipline for concurrency. Girard [7] observes that the connectives of linear logic can be interpreted as parallel computation. Abramsky [1] and Bellin and Scott [2] interpret linear logic proofs as processes in Milner's π-calculus. While they provide π-calculus interpretations of all linear logic proofs, they do not provide a proof-theoretic interpretation for arbitrary π-calculus terms. Caires and Pfenning [3] observe that the multiplicative connectives can be interpreted as session types. Their process calculus, based on intuitionistic linear logic, is closer to π-calculus than are traditional session-typed languages. Wadler [8] shows that a core session-typed linear functional language, GV, patterned after a similar language due to Gay and Vasconcelos [6], may be translated into a process calculus, CP, whose terms are proofs in classical linear logic. However, GV is less expressive than CP: there are proofs which do not correspond to any GV program.

Our primary contribution is HGV (Harmonious GV), a version of GV extended with constructs for session forwarding, replication, and polymorphism. We identify HGVπ, the session-typed fragment of HGV, and give a type-preserving translations from HGV to HGVπ ($(-)^\star$); this translation depends crucially on the new constructs of HGV. We show that HGV is sufficient to express all linear logic proofs by giving type-preserving translations from HGVπ to CP ($[\![-]\!]$), and from CP to HGVπ ($(\!|-|\!)$). Factoring the translation of HGV into CP through $(-)^\star$ simplifies the presentation, and illuminates regularities that are not apparent in Wadler's original translation of GV into CP. Finally, we show that HGV, HGVπ, and CP are all equally expressive.

2 The HGV Language

This section describes our session-typed language HGV, contrasting it with Gay and Vasconcelos's functional language for asynchronous session types [6], which we call LAST, and Wadler's GV [8]. In designing HGV, we have opted for programming convenience over uniformity, while insisting on a tight correspondence with linear logic. The session types of HGV are given by the following grammar:

$$S ::= !T.S \mid ?T.S \mid \oplus\{l_i : S_i\}_i \mid \&\{l_i : S_i\}_i \mid \mathsf{end}_! \mid \mathsf{end}_? \mid X \mid \overline{X} \mid ![X].S \mid ?[X].S \mid \flat S \mid \sharp S$$

Types for input ($?T.S$), output ($!T.S$), selection ($\oplus\{l_i : S_i\}_i$) and choice ($\&\{l_i : S_i\}_i$) are standard. Like GV, but unlike LAST, we distinguish output ($\mathsf{end}_!$) and input ($\mathsf{end}_?$) session ends; this matches the situation in linear logic, where there is no conveniently self-dual proposition to represent the end of a session. Variables and their duals ($X, \overline{X}$) and type input ($?[X].S$) and output ($![X].S$), permit definition of polymorphic sessions. We include a notion of replicated sessions, corresponding to exponentials in linear logic: a channel of type $\sharp S$ is a "service", providing any number of channels of type S; a channel of type $\flat S$ is the "server" providing such a service. Each session type S has a dual $\overline{S}$ (with the obvious dual for variables X):

$$\begin{array}{llllll} \overline{!T.S} = ?T.\overline{S} & \overline{\oplus\{l_i : S_i\}_i} = \&\{l_i : \overline{S_i}\}_i & \overline{\mathsf{end}_!} = \mathsf{end}_? & \overline{![X].S} = ?[X].\overline{S} & \overline{\sharp S} = \flat\overline{S} \\ \overline{?T.S} = !T.\overline{S} & \overline{\&\{l_i : S_i\}_i} = \oplus\{l_i : \overline{S_i}\}_i & \overline{\mathsf{end}_?} = \mathsf{end}_! & \overline{?[X].S} = ![X].\overline{S} & \overline{\flat S} = \sharp\overline{S} \end{array}$$

Note that dualisation leaves input and output types unchanged. In addition to sessions, HGV's types include linear pairs, and linear and unlimited functions:

$$T, U, V ::= S \mid T \otimes U \mid T \multimap U \mid T \to U$$

Every type T is either linear ($lin(T)$) or unlimited ($un(T)$); the only unlimited types are services ($un(\sharp S)$), unlimited functions ($un(T \to U)$), and end input session types ($un(\mathsf{end}_?)$). In GV, $\mathsf{end}_?$ is linear. We choose to make it unlimited in HGV because then we can dispense with GV's explicit $\mathsf{terminate}$ construct while maintaining a strong correspondence with CP—$\mathsf{end}_?$ corresponds to $\bot$ in CP, for which weakening and contraction are derivable.

Figure 1 gives the terms and typing rules for HGV; the first block contains the structural rules, the second contains the (standard) rules for lambda terms, and the third contains the session-typed fragment. The fork construct provides session initiation, filling the role of GV's $\mathsf{with} \ldots \mathsf{connect}$ structure, but without the asymmetry of the latter. The two are interdefinable, as follows:

$$\mathsf{fork}\ x.M \equiv \mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ x \qquad \mathsf{with}\ x\ \mathsf{connect}\ M\ \mathsf{to}\ N \equiv \mathsf{let}\ x = \mathsf{fork}\ x.M\ \mathsf{in}\ N$$

We add a construct $\mathsf{link}\ M\ N$ to implement channel forwarding; this form is provided in neither GV nor LAST, but is necessary to match the expressive power of CP. (Note that while we could define session forwarding in GV or LAST for any particular session type, it is not possible to do so in a generic fashion.) We add terms $\mathsf{sendType}\ S\ M$ and $\mathsf{receiveType}\ X.M$ to provide session polymorphism, and $\mathsf{serve}\ x.M$ and $\mathsf{request}\ M$ for replicated sessions. Note that, as the body M of $\mathsf{serve}\ x.M$ may be arbitrarily replicated, it can only refer to the unlimited portion of the environment. Channels of type $\sharp S$ offer arbitrarily many sessions of type S; correspondingly, channels of type $\flat S$ must consume arbitrarily many S sessions. The rule for $\mathsf{serve}\ x.M$ parallels that for fork: it defines the server (which replicates M) and returns the channel by which it may be used (of type $\overline{\flat S} = \sharp\overline{S}$). As a consequence, there is no rule involving type $\flat S$. We experimented with having such a rule, but found that it was always used immediately inside a fork, while providing no extra expressive power. Hence we opted for the rule presented here.

3 From HGV to HGVπ

The language HGVπ is the restriction of HGV to session types, that is, HGV without $\multimap$, $\to$, or $\otimes$. In order to avoid $\otimes$, we disallow plain $\mathsf{receive}\ M$, but do permit it to be fused with a pair elimination $\mathsf{let}\ (x, y) = \mathsf{receive}\ M\ \mathsf{in}\ N$. We can simulate all non-session types as session types

Structural rules

$$\frac{}{x : T \vdash x : T} \qquad \frac{\Phi \vdash N : U \qquad un(T)}{\Phi, x : T \vdash N : U} \qquad \frac{\Phi, x : T, x' : T \vdash N : U \qquad un(T)}{\Phi, x : T \vdash N[x/x'] : U}$$

Lambda rules

$$\frac{\Phi, x : T \vdash N : U}{\Phi \vdash \lambda x.N : T \multimap U} \qquad \frac{\Phi \vdash L : T \multimap U \qquad \Psi \vdash M : T}{\Phi, \Psi \vdash L\ M : U} \qquad \frac{\Phi \vdash L : T \multimap U \qquad un(\Phi)}{\Phi \vdash L : T \to U}$$

$$\frac{\Phi \vdash L : T \to U}{\Phi \vdash L : T \multimap U} \qquad \frac{\Phi \vdash M : T \qquad \Psi \vdash N : U}{\Phi, \Psi \vdash (M, N) : T \otimes U} \qquad \frac{\Phi \vdash M : T \otimes U \qquad \Psi, x : T, y : U \vdash N : V}{\Phi, \Psi \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N : V}$$

Session rules

$$\frac{\Phi \vdash M : T \qquad \Psi \vdash N : !T.S}{\Phi \vdash \mathsf{send}\ M\ N : S} \qquad \frac{\Phi \vdash M : ?T.S}{\Phi \vdash \mathsf{receive}\ M : T \otimes S}$$

$$\frac{\Phi \vdash M : \oplus\{l_i : S_i\}_i}{\Phi \vdash \mathsf{select}\ l_j\ M : S_j} \qquad \frac{\Phi \vdash M : \&\{l_i : S_i\}_i \qquad \{\Psi, x : S_i \vdash N_i : T\}_i}{\Phi, \Psi \vdash \mathsf{case}\ M\ \mathsf{of}\ \{l_i(x).N_i\}_i : T}$$

$$\frac{\Phi, x : S \vdash M : \mathsf{end}_!}{\Phi \vdash \mathsf{fork}\ x.M : \overline{S}} \qquad \frac{\Phi \vdash M : S \qquad \Phi \vdash N : \overline{S}}{\Phi \vdash \mathsf{link}\ M\ N : \mathsf{end}_!} \qquad \frac{\Phi \vdash M : ![X].S'}{\Phi \vdash \mathsf{sendType}\ S\ M : S'[S/X]}$$

$$\frac{\Phi \vdash M : ?[X].S \qquad X \notin FV(\Phi)}{\Phi \vdash \mathsf{receiveType}\ X.M : S} \qquad \frac{\Phi, x : S \vdash M : \mathsf{end}_! \qquad un(\Phi)}{\Phi \vdash \mathsf{serve}\ x.M : \sharp\overline{S}} \qquad \frac{\Phi \vdash M : \sharp S}{\Phi \vdash \mathsf{request}\ M : S}$$

Figure 1: Typing rules for HGV

via a translation from HGV to HGVπ. The translation on types is given by the homomorphic extension of the following equations:

$$(T \multimap U)^\star = !(T)^\star.\overline{(U)^\star} \qquad (T \to U)^\star = \sharp(!(T)^\star.\overline{(U)^\star}) \qquad (T \otimes U)^\star = ?(T)^\star.\overline{(U)^\star}$$

Each target type is the *interface* to the simulated source type. A linear function is simulated by input on a channel; its interface is output on the other end of the channel. An unlimited function is simulated by a server; its interface is the service on the other end of that channel. A tensor is simulated by output on a channel; its interface is input on the other end of that channel. This duality between implementation and interface explains the flipping of types in Wadler's original CPS translation from GV to CP. The translation on terms is given by the homomorphic extension of the following equations:

$$\begin{aligned}
(\lambda x.M)^\star &= \mathsf{fork}\ z.\mathsf{let}\ (x, z) = \mathsf{receive}\ z\ \mathsf{in}\ \mathsf{link}\ (M)^\star\ z \\
(L\ M)^\star &= \mathsf{send}\ (M)^\star\ (L)^\star \\
(M \otimes N)^\star &= \mathsf{fork}\ z.\mathsf{link}\ (\mathsf{send}\ (M)^\star\ z)\ (N)^\star \\
(\mathsf{let}\ (x, y) = M\ \mathsf{in}\ N)^\star &= \mathsf{let}\ (x, y) = \mathsf{receive}\ (M)^\star\ \mathsf{in}\ (N)^\star \\
(L : T \to U)^\star &= \mathsf{fork}\ z.\mathsf{link}\ z\ (\mathsf{serve}\ y.\mathsf{link}\ (L)^\star\ y) \\
(L : T \multimap U)^\star &= \mathsf{request}\ (L)^\star \\
(\mathsf{receive}\ M)^\star &= (M)^\star
\end{aligned}$$

$$\frac{}{w \leftrightarrow x \vdash w : A^\perp, x : A} \qquad \frac{P \vdash \Gamma, x : A \qquad Q \vdash \Delta, x : A^\perp}{\nu x.(P \mid Q) \vdash \Gamma, \Delta} \qquad \frac{P \vdash \Gamma, y : A \qquad Q \vdash \Delta, x : B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x : A \otimes B}$$

$$\frac{R \vdash \Theta, y : A, x : B}{x(y).R \vdash \Theta, x : A \parr B} \qquad \frac{P \vdash \Gamma, x : A_i}{x[l_i].P \vdash \Gamma, x : \oplus\{l_i : A_i\}_i} \qquad \frac{\{Q_i \vdash \Delta, x_i : A_i\}_i}{x.\mathsf{case}\ \{l_i.Q_i\}_i \vdash \Delta, x : \&\{l_i : A_i\}_i}$$

$$\frac{P \vdash ?\Gamma, y : A}{!x(y).P \vdash ?\Gamma, x : !A} \qquad \frac{Q \vdash \Delta, y : A}{?x[y].Q \vdash \Delta, x : ?A} \qquad \frac{Q \vdash \Delta}{Q \vdash \Delta, x : ?A} \qquad \frac{Q \vdash \Delta, x : ?A, x' : ?A}{Q[x/x'] \vdash \Delta, x : ?A}$$

$$\frac{P \vdash \Gamma, x : B[A/X]}{x[A].P \vdash \Gamma, x : \exists X.B} \qquad \frac{Q \vdash \Delta, x : B \qquad X \notin \Delta}{x(X).Q \vdash \Delta, x : \forall X.B} \qquad \frac{}{x[].0 \vdash x : 1} \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x : \bot}$$

Figure 2: Typing rules for CP

Formally, this is a translation on derivations. We write type annotations to indicate $\to$ introduction and elimination. For all other cases, it is unambiguous to give the translation on plain term syntax. Each introduction form translates to an interface $\mathsf{fork}\ z.M$ of type $\overline{S}$, where $M : \mathsf{end}_!$ provides the implementation, with $z : S$ bound in M. We can extend the translation on types to a translation on contexts:

$$(x_1 : T_1, \ldots, x_n : T_n)^\star = x_1 : (T_1)^\star, \ldots, x_n : (T_n)^\star$$

Finally, it is straightforward to verify that our translation preserves typing.

Theorem 1. *If $\Phi \vdash M : T$ then $(\Phi)^\star \vdash (M)^\star : (T)^\star$.*

4 From HGVπ to CP

We present the typing rules of CP in Figure 2. Due to lack of space we omit the definition of the cut relation $\longrightarrow$. A detailed description of CP, including the cut rules, can be found in Wadler's work [8]. Note that the propositions of CP are exactly those of classical linear logic, as are the cut rules (if we ignore the terms). Thus, CP enjoys all of the standard meta theoretic properties of classical linear logic, including confluence and weak normalisation. A minor syntactic difference between our presentation and Wadler's is that our sum ($\oplus$) and choice ($\&$) types are n-ary, matching the corresponding session types in HGV, whereas he presents binary and nullary versions of sum and choice. Duality on CP types ($(-)^\perp$) is standard:

$$\begin{array}{llllll} (A \otimes B)^\perp = A^\perp \parr B^\perp & (\oplus\{l_i : A_i\}_i)^\perp = \&\{l_i : {A_i}^\perp\}_i & 1^\perp = \bot & (\exists X.B)^\perp = \forall X.B^\perp & (!A)^\perp = ?A^\perp \\ (A \parr B)^\perp = A^\perp \otimes B^\perp & (\&\{l_i : A_i\}_i)^\perp = \oplus\{l_i : {A_i}^\perp\}_i & \bot^\perp = 1 & (\forall X.B)^\perp = \exists X.B^\perp & (?A)^\perp = !A^\perp \end{array}$$

We now give a translation from HGVπ to CP. Post composing this with the embedding of HGV in HGVπ yields a semantics for HGV. The translation on session types is as follows:

$$\begin{array}{llll} [\![!T.S]\!] = [\![T]\!]^\perp \otimes [\![S]\!] & [\![\oplus\{l_i : S_i\}_i]\!] = \oplus\{l_i : [\![S_i]\!]\}_i & [\![\flat S]\!] = ![\![S]\!] & [\![![X].S]\!] = \exists X.[\![S]\!] \\ [\![?T.S]\!] = [\![T]\!] \parr [\![S]\!] & [\![\&\{l_i : S_i\}_i]\!] = \&\{l_i : [\![S_i]\!]\}_i & [\![\sharp S]\!] = ?[\![S]\!] & [\![?[X].S]\!] = \forall X.[\![S]\!] \\ [\![\mathsf{end}_!]\!] = 1 & [\![\mathsf{end}_?]\!] = \bot & [\![X]\!] = X & [\![\overline{X}]\!] = X^\perp \end{array}$$

The translation is homomorphic except for output, where the output type is dualised. This accounts for the discrepancy between $\overline{!T.S} = ?T.\overline{S}$ and $(A \otimes B)^\perp = A^\perp \parr B^\perp$.

The translation on terms is formally specified as a CPS translation on derivations as in Wadler's presentation. We provide the full translations of weakening and contraction for $\mathsf{end}_?$, as these steps are implicit in the syntax of HGV terms. The other constructs depend only on the immediate syntactic structure, so we abbreviate their translations as mappings on plain terms:

$$\left[\!\!\left[\frac{\Phi \vdash N : S}{\Phi, x : \mathsf{end}_? \vdash N : S} \right]\!\!\right] z = \frac{[\![N]\!] z \vdash [\![\Phi]\!], z : [\![S]\!]^\perp}{x().[\![N]\!] z \vdash [\![\Phi]\!], x : \bot, z : [\![S]\!]^\perp}$$

$$\left[\!\!\left[\frac{\Phi, x : \mathsf{end}_?, x' : \mathsf{end}_? \vdash N : S}{\Phi, x : \mathsf{end}_? \vdash N[x/x'] : S} \right]\!\!\right] z = \frac{[\![N]\!] z \vdash [\![\Phi]\!], x : \bot, x' : \bot, z : [\![S]\!]^\perp}{\nu x'.([\![N]\!] z \mid x'[].0) \vdash [\![\Phi]\!], x : \bot, z : [\![S]\!]^\perp}$$

$$\begin{aligned}
[\![x]\!] z &= x \leftrightarrow z \\
[\![\mathsf{send}\ M\ N]\!] z &= \nu x.(x[y].([\![M]\!] y \mid x \leftrightarrow z) \mid [\![N]\!] x) \\
[\![\mathsf{let}\ (x, y) = \mathsf{receive}\ M\ \mathsf{in}\ N]\!] z &= \nu y.([\![M]\!] y \mid y(x).[\![N]\!] z) \\
[\![\mathsf{select}\ l\ M]\!] z &= \nu x.([\![M]\!] x \mid x[l].x \leftrightarrow z) \\
[\![\mathsf{case}\ M\ \mathsf{of}\ \{l_i(x).N_i\}_i]\!] z &= \nu x.([\![M]\!] x \mid x.\mathsf{case}\ \{l_i.[\![N_i]\!] z\}_i) \\
[\![\mathsf{fork}\ x.M]\!] z &= \nu x.(\nu y.([\![M]\!] y \mid y[].0) \mid x \leftrightarrow z) \\
[\![\mathsf{link}\ M\ N]\!] z &= z().\nu x.([\![M]\!] x \mid [\![N]\!] x) \\
[\![\mathsf{sendType}\ S\ M]\!] z &= \nu x.([\![M]\!] x \mid x[[\![S]\!].x \leftrightarrow z) \\
[\![\mathsf{receiveType}\ X.M]\!] z &= \nu x.([\![M]\!] x \mid x(X).x \leftrightarrow z) \\
[\![\mathsf{serve}\ y.M]\!] z &= !z(y).\nu x.([\![M]\!] x \mid x[].0) \\
[\![\mathsf{request}\ M]\!] z &= \nu x.([\![M]\!] x \mid ?x[y].y \leftrightarrow z)
\end{aligned}$$

Channel z provides a continuation, consuming the output of the process representing the original HGVπ term. The translation on contexts is pointwise.

$$[\![x_1 : T_1, \ldots, x_n : T_n]\!] = x_1 : [\![T_1]\!], \ldots, x_n : [\![T_n]\!]$$

As with the translation from HGV to HGVπ, we can show that this translation preserves typing.

Theorem 2. *If* $\Phi \vdash M : S$ *then* $[\![M]\!] z \vdash [\![\Phi]\!], z : [\![S]\!]^\perp$.

5 From CP to HGVπ

We now present the translation $(\!|-|\!)$ from CP to HGVπ. The translation on types is as follows:

$$\begin{array}{llll}
(\!|A \otimes B|\!) = !\overline{(\!|A|\!)}.(\!|B|\!) & (\!|\oplus\{l_i : A_i\}_i|\!) = \oplus\{l_i : (\!|A_i|\!)\}_i & (\!|\exists X.A|\!) = ![X].(\!|A|\!) & (\!|?A|\!) = \sharp(\!|A|\!) \\
(\!|A \parr B|\!) = ?(\!|A|\!).(\!|B|\!) & (\!|\&\{l_i : A_i\}_i|\!) = \&\{l_i : (\!|A_i|\!)\}_i & (\!|\forall X.A|\!) = ?[X].(\!|A|\!) & (\!|!A|\!) = \flat(\!|A|\!) \\
(\!|1|\!) = \mathsf{end}_! & (\!|\bot|\!) = \mathsf{end}_? & (\!|X|\!) = X & (\!|X^\perp|\!) = \overline{X}
\end{array}$$

The translation on terms makes use of let expressions to simplify the presentation; these are expanded to HGVπ as follows:

$$\mathsf{let}\ x = M\ \mathsf{in}\ N \equiv ((\lambda x.N)M)^\star \equiv \mathsf{send}\ M\ (\mathsf{fork}\ z.\mathsf{let}\ (x, z) = \mathsf{receive}\ z\ \mathsf{in}\ \mathsf{link}\ N\ z).$$

$$
\begin{aligned}
(\!|x[y].(P \mid Q)|\!) &= \mathsf{let}\ x = \mathsf{send}\ (\mathsf{fork}\ y.(\!|P|\!))\ x\ \mathsf{in}\ (\!|Q|\!) \\
(\!|x(y).P|\!) &= \mathsf{let}\ (y, x) = \mathsf{receive}\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|x[l].P|\!) &= \mathsf{let}\ x = \mathsf{select}\ l\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|x.\mathsf{case}\ \{l_i.P_i\}_i|\!) &= \mathsf{case}\ x\ \mathsf{of}\ \{l_i(x).(\!|P_i|\!)\}_i \\
(\!|x[].0|\!) &= x \\
(\!|x().P|\!) &= (\!|P|\!) \\
(\!|\nu x.(P \mid Q)|\!) &= \mathsf{let}\ x = \mathsf{fork}\ x.(\!|P|\!)\ \mathsf{in}\ (\!|Q|\!) \\
(\!|x \leftrightarrow y|\!) &= \mathsf{link}\ x\ y \\
(\!|x[A].P|\!) &= \mathsf{let}\ x = \mathsf{sendType}\ (\!|A|\!)\ x\ \mathsf{in}\ (\!|P|\!) \\
(\!|x(X).P|\!) &= \mathsf{let}\ x = \mathsf{receiveType}\ X.x\ \mathsf{in}\ (\!|P|\!) \\
(\!|!s(x).P|\!) &= \mathsf{link}\ s\ (\mathsf{serve}\ x.(\!|P|\!)) \\
(\!|?s[x].P|\!) &= \mathsf{let}\ x = \mathsf{fork}\ x.\mathsf{link}\ (\mathsf{request}\ s)\ x\ \mathsf{in}\ (\!|P|\!)
\end{aligned}
$$

Again, we can extend the translation on types to a translation on contexts, and show that the translation preserves typing.

Theorem 3. *If* $P \vdash \Gamma$ *then* $(\!|\Gamma|\!) \vdash (\!|P|\!) : \mathsf{end}_!$.

6 Correctness

If we extend $[\![-]\!]$ to non-session types, as in Wadler's original presentation (Figure 3), then it is straightforward to show that this monolithic translation factors through $(-)^\star$.

Theorem 4. $[\![(M)^\star]\!]z \longrightarrow^* [\![M]\!]z$ *(where* $\longrightarrow^*$ *is the transitive reflexive closure of* $\longrightarrow$*).*

The key soundness property of our translations is that if we translate a term from CP to HGVπ and back, then we obtain a term equivalent to the one we started with.

Theorem 5. *If* $P \vdash \Gamma$ *then* $\nu z.(z[].0 \mid [\![(\!|P|\!)]\!]z) \longrightarrow^* P$.

Together, Theorem 4 and 5 tell us that HGV, HGVπ, and CP are equally expressive, in the sense that every X program can always be translated to an equivalent Y program, where $X, Y \in \{\text{HGV, HGV}\pi\text{, CP}\}$.

Here our notion of expressivity is agnostic to the nature of the translations. It is instructive also to consider Felleisen's more refined notion of expressivity [5]. Both $(-)^\star$ and $(\!|-|\!)$ are local translations, thus both HGV and CP are *macro-expressible* [5] in HGVπ. However, the need for a global CPS translation from HGVπ to CP illustrates that HGVπ is not macro-expressible in CP; hence HGVπ is more expressive, in the Felleisen sense, than CP.

7 Conclusions and Future Work

We have proposed a session-typed functional language, HGV, building on similar languages of Wadler [8] and of Gay and Vasconcelos [6]. We have shown that HGV is sufficient to encode arbitrary linear logic proofs, completing the correspondence between linear logic and session types. We have also given an embedding of all of HGV into its session-typed fragment, simplifying translation from HGV to CP.

Dardha et al [4] offers an alternative foundation for session types through a CPS translation of π-calculus with session types into a linear π-calculus. There appear to be strong similarities between their CPS translation and ours. We would like to make the correspondence precise by studying translations between their systems and ours.

Types

$$[\![T \multimap U]\!] = ([\![T]\!]^\perp \parr [\![U]\!])^\perp$$

$$[\![T \to U]\!] = (!([\![T]\!]^\perp \parr [\![U]\!]))^\perp$$

$$[\![T \otimes U]\!] = ([\![T]\!] \otimes [\![U]\!])^\perp$$

Terms

$$[\![\lambda x.N]\!]z = z(x).[\![N]\!]z$$
$$[\![L\ M]\!]z = \nu y.([\![L]\!]y \mid y[x].([\![M]\!]x \mid y \leftrightarrow z))$$
$$[\![L : T \to U]\!]z = !z(y).[\![L]\!]y$$
$$[\![L : T \multimap U]\!]z = \nu y.([\![L]\!]y \mid ?y[x].x \leftrightarrow z)$$
$$[\![(M, N)]\!]z = z[y].([\![M]\!]y \mid [\![N]\!]z)$$
$$[\![\mathsf{let}\ (x, y) = M\ \mathsf{in}\ N]\!]z = \nu y.([\![M]\!]y \mid y(x).[\![N]\!]z)$$

The outer duals appear in the type translation because, as in Section 3, we must expose *interfaces* rather than implementations of simulated types. As in the definition of $(-)^\star$ in Section 3, we write type annotations to indicate $\to$ introduction and elimination.

Figure 3: Extension of $[\![-]\!]$ to non-session types

In addition we highlight several other areas of future work. First, the semantics of HGV is given only by cut elimination in CP. We would like to give HGV a semantics directly, in terms of reductions of configurations of processes, and then prove a formal correspondence with cut elimination in CP. Second, replication has limited expressive power compared to recursion; in particular, it cannot express services whose behaviour changes over time or in response to client requests. We believe that the study of fixed points in linear logic provides a mechanism to support more expressive recursive behaviour without sacrificing the logical interpretation of HGV. Finally, as classical linear logic proofs, and hence CP processes, enjoy confluence, HGV programs are deterministic. We hope to identify natural extensions of HGV that give rise to non-determinism, and thus allow programs to exhibit more interesting concurrent behaviour, while preserving the underlying connection to linear logic.

Acknowledgements We would like to thank Philip Wadler for his suggestions on the direction of this work, and for his helpful feedback on the results. This work was funded by EPSRC grant number EP/K034413/1.

References

[1] Samson Abramsky. Proofs as processes. MFPS '92, pages 5–9. Elsevier, 1994.

[2] Gianluigi Bellin and Philip J. Scott. On the π-Calculus and linear logic. *Theoretical Computer Science*, 135(1):11–65, 1994.

[3] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *International Conference on Concurrency Theory*, CONCUR '10, pages 222–236, 2010.

[4] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP*, pages 139–150, 2012.

[5] Matthias Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1–3):35–75, 1991.

[6] Simon J. Gay and Vasco T. Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19–50, 2010.

[7] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987.

[8] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286. ACM, 2012.

Towards Reversible Sessions*

Francesco Tiezzi[1] and Nobuko Yoshida[2]

[1] IMT Institute for Advanced Studies, Lucca, Italy
francesco.tiezzi@imtlucca.it

[2] Imperial College, London, U.K.
n.yoshida@imperial.ac.uk

Abstract

In this work, we incorporate reversibility into structured communication-based programming, to allow parties of a session to automatically undo, in a rollback fashion, the effect of previously executed interactions. This permits taking different computation paths along the same session, as well as reverting the whole session and starting a new one. Our aim is to define a theoretical basis for examining the interplay in concurrent systems between reversible computation and session-based interaction. We thus enrich a session-based variant of π-calculus with memory devices, dedicated to keep track of the computation history of sessions in order to reverse it. We discuss our initial investigation concerning the definition of a session type discipline for the proposed reversible calculus, and its practical advantages for static verification of safe composition in communication-centric distributed software performing reversible computations.

1 Introduction

Reversible computing aims at providing a computational model that, besides the standard (forward) executions, also permits backward execution steps in order to undo the effect of previously performed forward computations. Reversibility is a key ingredient in different application domains since many years and, recently, also in the design of reliable concurrent systems, as it permits understanding existing patterns for programming reliable systems (e.g., compensations, checkpointing, transactions) and, possibly, improving them or developing new ones.

A promising line of research on this topic advocates reversible variants of well-established process calculi, such as CCS and π-calculus, as formalisms for studying reversibility mechanisms in concurrent systems. Our work incorporates reversibility into a variant of π-calculus equipped with *session* primitives supporting structured communication-based programming. A (binary) session consists in a series of reciprocal interactions between two parties, possibly with branching and recursion. Interactions on a session are performed via a dedicated private channel, which is generated when initiating the session. Session primitives come together with a session type discipline offering a simple static checking framework to guarantee the correctness of communication patterns.

Practically, combining reversibility and sessions paves the way for the development of session-based communication-centric distributed software intrinsically capable of performing reversible computations. In this way, without further coding effort by the application programmer, the interaction among session parties is relaxed so that, e.g., the computation can automatically go back, thus allowing to take different paths when the current one is not satisfactory. As an application example, used in this paper for illustrating our approach, we consider a simple scenario involving a client and multiple providers offering the same service (e.g., on-demand video streaming). The client connects to a provider to request a given service (specifying, e.g., title of a movie, video quality, etc.). The provider replies with a quote

*This work has been partially supported by the COST Action BETTY (IC1201), by the EU project ASCENS (257414), and by the Italian MIUR PRIN project CINA (2010LHT4KM).

determined according to the requested quality of service and to the servers status (current load, available bandwidth, etc.). Then, the client can either accept, negotiate or reject the quote. If a problem occurs during the interaction between the client and the provider, the computation can be reverted, in order to allow the client to automatically start a new session with (possibly) another provider.

The proposed reversible session-based calculus relies on memories to store information about interactions and their effects on the system, which otherwise would be lost during forward computations. This data is used to enable backward computations that revert the effects of the corresponding forward ones. Each memory is devoted to record data concerning a single event, which can correspond to the taking place of a communication action, a choice or a thread forking. Memories are connected each other, in order to keep track of the computation history, by using unique thread identifiers as links. Like all other formalisms for reversible computing in concurrent settings, forward computations are undone in a *causal-consistent* fashion, i.e. backtracking does not have to necessarily follow the exact order of forward computations in reverse, because independent actions can be undone in a different order.

The resulting formalism offers a theoretical basis for examining the interplay between reversible computations and session-based structured interactions. We notice that reversibility enables session parties not only to partially undo the interactions performed along the current session, but also to automatically undo the whole session and restart it, possibly involving different parties. The advantage of the reversible approach is that this behaviour is realised without explicitly implementing loops. On the other hand, the session type discipline affects reversibility as it forces concurrent interactions to follow structured communication patterns. In fact, linearizing behaviours on sessions reduces the effect of causal consistency, because concurrent interactions along the same session are forbidden and, hence, the rollback along a session follows a single path. However, interactions along different sessions are still concurrent and, therefore, they can be reverted as usual in a causal-consistent fashion. Notably, interesting issues concerning reversibility and session types are still open questions, especially for what concerns the validity in the reversible setting of standard properties (e.g., progress enforcement) and possibly new properties (e.g., reversibility of ongoing session history, irreversible closure of sessions).

2 Related work

We review here the most closely related works, which concern the definition of reversible process calculi; we refer to [9] for a more detailed review of reversible calculi.

Reversible CCS (RCCS) [3] is the first proposal of reversible calculus, from which all subsequent works drew inspiration. To each currently running thread is associated an individual memory stack keeping track of past actions, as well as forks and synchronisations. Information pushed on the memory stacks, upon doing a forward transition, can be then used for a roll-back. The memories also serve as a naming scheme and yield unique identifiers for threads. When a process divides in two sub-threads, each sub-thread inherits the father memory together with a fork number indicating which of the two sons the thread is. A drawback of this approach is that the parallel operator does not satisfy usual structural congruence rules as commutativity, associativity and nil process as neutral element.

CCS-R [4] is another reversible variant of CCS, which mainly aims at formalising biological systems. Like RCCS, it relies on memory stacks for storing data needed for backtracking, which now also includes events corresponding to unfolding of process definitions. Differently from RCCS, specific identifiers are used to label threads, and a different approach is used for dealing with forking.

CCS with communication Keys (CCSK) [8] is a reversible process calculus obtained by applying a general procedure to produce reversible calculi. A relevant aspect of this approach is that it does not rely on memories for supporting backtracking. The idea is to maintain the structure of processes fixed throughout computations, thus avoiding to consume guards and alternative choices. To properly revert synchronisations, the two threads have to agree on a key, uniquely identifying that communication.

$\rho\pi$ [7] is a reversible variant of the higher-order π-calculus. It borrows from RCCS the use of memories for keeping track of past actions, although in $\rho\pi$ they are not stacks syntactically associated to threads, but parallel terms each one dedicated to a single communication. The connection between memories and threads is kept by resorting to identifiers, which resemble CCSK keys. Fork handling is based on structured tags, used to connect the identifier of a thread with the identifiers of its sub-threads. This approach to reversibility has been applied in [5] to a distributed tuple-based language.

Another reversible variant of π-calculus is Rπ [2]. Similarly to RCCS, this calculus relies on memory stacks, now recording communication events and forking. Differently from $\rho\pi$, it considers standard π-calculus (without choice and replication) as a host calculus and its semantics is defined in terms of a labelled transition relation.

Finally, reversible structures [1] is a simple computational calculus for modelling chemical systems. Reversible structures does not exploit memories, but maintains the structure of terms and uses a special symbol to indicate the next forward and backward operations that a term can perform.

In our work, we mainly take inspiration from the $\rho\pi$ approach. In fact, all other approaches are based on CCS and cannot be directly applied to a calculus with name-passing. Moreover, the $\rho\pi$ approach is preferable to the Rπ one because the former proposes a reduction semantics, which we are interested in, while the latter proposes a labelled semantics, which would complicate our theoretical framework in order to properly deal with scope extrusion.

3 Reversible Session-based π-calculus

In this section, we introduce a reversible extension of a π-calculus enriched with primitives for managing *binary* sessions, i.e. structured interactions between two parties. We call *ReS*π (*Reversible Session-based* π-calculus) this formalism. Due to lack of space, some technical details about semantics and results have been omitted; we refer the interested reader to [9] for a more complete account.

From π-calculus to *ReS*π. Our approach to keep track of computation history in *ReS*π is as follows: we tag processes with unique identifiers (tagged processes are called *threads*) and use memories to store the information needed to reverse each single forward reduction. Thus, the history of a reduction sequence is stored in a number of small memories connected each other by using tags as links. In this way, *ReS*π terms can perform, besides *forward computations* (denoted by $\twoheadrightarrow$), also *backward computations* (denoted by $\rightsquigarrow$) that undo the effect of the former ones in a causal-consistent fashion.

π-calculus *processes* and *expressions* are given by the grammars in Figure 1. The synchronisation on a shared channel a of processes $\bar{a}(x).P$ and $a(y).Q$ initiates a session along a fresh session channel s. This channel consists in a pair of (dual) endpoints, denoted by s and $\bar{s}$ (such that $\bar{\bar{s}} = s$), each one dedicated to one party to exchange values with the other. These endpoints replace variables x and y, by means of a substitution application, in order to be used by P and Q, respectively, for later communications. Primitives $k!\langle e\rangle.P$ and $k'?(x).Q$ denote output and input via session endpoints identified by k and k', respectively. These communication primitives realise the standard synchronous message passing, where messages result from expressions evaluation and may contain endpoints (*delegation*). Constructs $k \triangleleft l.P$ and $k' \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ denote label selection and branching (with $l_1, \ldots, l_n$ pairwise distinct) via k and k', respectively. The above interaction primitives are combined by conditional choice, parallel composition, restriction, recursion and inaction.

ReSπ processes are built upon π-calculus processes by labelling them with *tags* to uniquely identify threads $t : P$. Uniqueness of tags is ensured by using the restriction operator and by only considering *reachable* terms (Def. 1). Moreover, *ReS*π extends π-calculus with three kinds of *memories* m. An *action memory* $\langle t_1 \mathrel{-}\mathrm{A}\mathrel{\rightarrow} t_2, t_1', t_2'\rangle$ stores an action event A together with the tag t_1 of the active party of

Shared channels $a, b, \ldots$ | Session channels $s, s', \ldots$ | Session endpoints $s, \bar{s}, s', \bar{s}', \ldots$ | Variables $x, y, \ldots$

Labels $l, l', \ldots$ | Process variables $X, Y, \ldots$ | Tags $t, t', \ldots$ | Shared ids $u ::= a \mid x$

Channels $c ::= a \mid s$ | Names $h ::= c \mid t$ | Session ids $k ::= s \mid \bar{s} \mid x$

$$\begin{array}{rrl}
\textbf{Processes} & P & ::= \bar{u}(x).P \mid u(x).P \mid k!\langle e\rangle.P \mid k?(x).P \mid k \triangleleft l.P \mid k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \\
& & \quad \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \mid P \mid Q \mid (\nu c)P \mid X \mid \mu X.P \mid \mathbf{0} \\
\textbf{Expressions} & e & ::= v \mid x \mid \mathrm{op}(e_1, \ldots, e_n) \\
\textbf{Values} & v & ::= \texttt{true} \mid \texttt{false} \mid 0, 1, \ldots \mid a \mid s \mid \bar{s} \\
\textbf{\textit{ReS}}\pi\textbf{ processes} & M & ::= t : P \mid (\nu h)M \mid M \mid N \mid m \mid \mathrm{nil} \\
\textbf{Memories} & m & ::= \langle t_1 -\mathrm{A}\rightarrow t_2, t_1', t_2'\rangle \mid \langle t, e?P : Q, t'\rangle \mid \langle t \rightrightarrows (t_1, t_2)\rangle \\
& \mathrm{A} & ::= a(x)(y)(\nu s)PQ \mid k\langle e\rangle(x)PQ \mid k \triangleleft l_i P\{l_1 : P_1, \ldots, l_n : P_n\}
\end{array}$$

Figure 1: *ReS*π syntax

the action, the tag t_2 of the passive party, and the tags t_1' and t_2' of the new threads activated by the corresponding reduction. An *action event* records information necessary to revert each kind of interactions, which can be either a session initiation $a(x)(y)(\nu s)PQ$, a communication along an established session $k\langle e\rangle(x)PQ$, or a branch selection $k \triangleleft l_i P\{l_1 : P_1, \ldots, l_n : P_n\}$. A *choice memory* $\langle t, e?P : Q, t'\rangle$ stores a choice event together with the tag t of the conditional choice and t' of the new activated thread. The event $e?P : Q$ records the evaluated expression e, and processes P and Q of the then- and else-branch, respectively. A *fork memory* $\langle t \rightrightarrows (t_1, t_2)\rangle$ stores the tag t of a splitting thread, of the form $t : (P \mid Q)$, and the tags t_1 and t_2 of the new activated threads $t_1 : P$ and $t_2 : Q$; these memories are analogous to *connectors* in [5]. Threads and memories are composed by parallel composition and restriction operators.

Not all processes allowed by the syntax are semantically meaningful. In a general term, the history stored in the memories may not be consistent, due to the use of non-unique tags or broken connections between continuation tags within memories and corresponding threads. For example, given the choice memory $\langle t, e?P : Q, t'\rangle$, we have a broken connection when no thread tagged by t' exists in the *ReS*π process and no memory of the form $\langle t' -\mathrm{A}\rightarrow t_2, t_1', t_2'\rangle$, $\langle t_1 -\mathrm{A}\rightarrow t', t_1', t_2'\rangle$, $\langle t', e?P_1 : P_2, t_1\rangle$, and $\langle t' \rightrightarrows (t_1, t_2)\rangle$ exist. Thus, as in [2], to ensure history consistency we only consider *reachable* processes, i.e. processes obtained by means of forward and backward reductions from processes with unique tags and no memory.

Def. 1 (Reachable processes)**.** *The set of* reachable *ReS*π *processes is the closure under* $\rightarrowtail$ *(see below) of the set of terms, whose threads have distinct tags, generated by* $M ::= t : P \mid (\nu c)M \mid M \mid N \mid \mathrm{nil}$.

***ReS*π semantics.** The *ReS*π operational semantics is given in terms of a *reduction relation* $\rightarrowtail$, given as the union of the forward and backward reduction relations. We report here, by way of examples, the forward and backward rules for session initiation (we require $s, \bar{s}$ fresh in P_1 and P_2 in the forward rule):

$$t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2 \twoheadrightarrow (\nu s, t_1', t_2')(t_1' : P_1[\bar{s}/x] \mid t_2' : P_2[s/y] \mid \langle t_1 -a(x)(y)(\nu s)P_1P_2\rightarrow t_2, t_1', t_2'\rangle)$$
$$(\nu s, t_1', t_2')(t_1' : P \mid t_2' : Q \mid \langle t_1 -a(x)(y)(\nu s)P_1P_2\rightarrow t_2, t_1', t_2'\rangle) \twoheadrightarrow t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2$$

When two parallel threads synchronise to establish a new session, two fresh tags are created to uniquely identify the continuations. Moreover, all relevant information is stored in the action memory: the tag t_1 of the initiator (i.e., the thread executing a prefix of the form $\bar{a}(\cdot)$), the tag t_2 of the thread executing the dual action, the tags t_1' and t_2' of their continuations, the shared channel a used for the synchronisation,

the replaced variables x and y, the generated session channel s, and the processes P_1 and P_2 to which substitutions are applied. All such information is exploited in the backward rule to revert this reduction. In particular, the corresponding backward reduction is triggered by the coexistence of the memory described above with two threads tagged t_1' and t_2', all of them within the scope of the session channel s and tags t_1' and t_2' generated by the forward reduction (which, in fact, are removed by the backward one). When considering reachable processes, due to tag uniqueness, processes P and Q coincide with $P_1[\overline{s}/x]$ and $P_2[s/y]$; indeed, as registered in the memory, these latter processes have been tagged with t_1' and t_2' by the forward reduction. Therefore, the fact that two threads tagged with t_1' and t_2' are in parallel with the memory ensures that all actions possibly executed by the two continuations activated by the forward computation have been undone and, hence, we can safely undone the forward computation itself.

Multiple providers scenario. The scenario involving a client and two providers introduced in Section 1 is rendered in *ReSπ* as $(t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2})$, where the client process P_{client} is

$$\overline{a_{login}}(x).x!\langle \mathsf{srv\_req} \rangle.x?(y_{quote}).\,\mathtt{if}\ accept(y_{quote})\ \mathtt{then}\ x \triangleleft l_{acc}.P_{acc} \\ \mathtt{else}\ (\mathtt{if}\ negotiate(y_{quote})\ \mathtt{then}\ x \triangleleft l_{neg}.P_{neg}\ \mathtt{else}\ x \triangleleft l_{rej}.\mathbf{0})$$

while $P_{provider\,i}$ is $a_{login}(y).y?(z_{req}).y!\langle quote_i(z_{req})\rangle.y \triangleright \{l_{acc} : Q_{acc}\ ,\ l_{neg} : Q_{neg}\ ,\ l_{rej} : \mathbf{0}\}$.

If the client contacts the first provider and accepts the proposed quote, the system evolves to $M = (\nu s, \ldots, t_1', t_2')(t_1' : P_{acc}[\overline{s}/x, \mathsf{quote}/y_{quote}] \mid t_2' : Q_{acc}[s/y, \mathsf{srv\_req}/z_{req}] \mid m_1 \mid \ldots \mid m_5) \mid P_{provider2}$, where memories m_i keep track of the computation history. Now, if a problem occurs during the subsequent interactions, the computation can be reverted to allow the client to start a new session with (possibly) another provider: $M \rightsquigarrow^* t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2}$.

Properties of *ReSπ*. We show here that *ReSπ* enjoys standard properties of reversible calculi.

First, we demonstrate that *ReSπ* is a conservative extension of the (session-based) π-calculus. In fact, as most reversible calculi, *ReSπ* is only a decoration of its host calculus. This decoration can be erased by means of the *forgetful map* ϕ, mapping *ReSπ* terms into π-calculus ones by removing memories, tag annotations and tag restrictions. The following lemmas show that each forward reduction of a *ReSπ* process corresponds to a reduction of the corresponding π-calculus process and vice versa.

Lemma 1. *Let M and N be two ReSπ processes. If $M \twoheadrightarrow N$ then $\phi(M) \rightarrow \phi(N)$.*

Lemma 2. *Let P and Q be two π-calculus processes. If $P \rightarrow Q$ then for any ReSπ process M such that $\phi(M) = P$ there exists a ReSπ process N such that $\phi(N) = Q$ and $M \twoheadrightarrow N$.*

Then, we show that *ReSπ* backward reductions are the inverse of the forward ones and vice versa.

Lemma 3 (Loop lemma)**.** *Let M and N be two reachable ReSπ processes. $M \twoheadrightarrow N$ if and only if $N \rightsquigarrow M$.*

We conclude with the *causal consistency* result stating that two sequences of reductions (called *traces* and ranged over by σ), with the same initial state (*coinitial*) and equivalent w.r.t. the standard notion of *causal equivalence* ($\asymp$), lead to the same final state (*cofinal*). Thus, in this case, we can rollback to the initial state by reversing any of the two traces.

Theorem 1. *Let σ_1 and σ_2 be coinitial traces. Then, $\sigma_1 \asymp \sigma_2$ if and only if σ_1 and σ_2 are cofinal.*

4 Discussion on a type discipline

A question that should be answered before defining a static type discipline for a reversible calculus is "*Should we type check the processes stored in the memories?*". The question arises from the fact that we should be able to determine if any *ReSπ* process is well-typed or not. In our case the answer is "*Yes*", otherwise typability would not be preserved under reduction (i.e., Subject Reduction would not be satisfied). It is indeed easy to define a *ReSπ* process (see [9]) containing a memory that, even if consistent, triggers a backward reduction leading to an untypable term (by the type system defined in [10] for the host calculus).

One could wonder now if it is possible to type *ReSπ* processes in a naïve way by separately type checking the term resulting from the application of ϕ and each single memory, by using the type system in [10]. For each memory we would check the term that has triggered the forward reduction generating the memory. In general, this approach does not work, because the term stored in a memory cannot be type checked in isolation without taking into account its context. For example, consider a memory corresponding to a communication along a session s typable under typing $\Delta = \bar{s} : ![\mathsf{int}].\mathsf{end} \cdot s : ?[\mathsf{int}].\mathsf{end}$ and in parallel with $(t_1 : \bar{s}!\langle 1\rangle \mid t_2 : s?(x))$. The term resulting from the corresponding backward reduction is not typable, because the typings of its sub-terms are not composable (indeed, $\Delta \cdot \Delta$ is not defined).

Memory context can be considered by extending the type system in [10] with rules that permits typing (processes stored in) memories and ignoring tag annotations and tag restrictions (see [9] for the definition of this type system). In this way, during type checking, typings of memories and threads must be composed by means of the rule for parallel composition. Thus, e.g., the *ReSπ* process mentioned above is, rightly, untypable. This type system properly works only on a simplified setting, which permits avoiding to deal with dependencies among memories and the threads outside memories, that could cause unwanted conflicts during type checking. Specifically, we consider the class of *ReSπ* processes that, extending Def. 1, are obtained by means of forward and backward reductions from processes with unique tags, no memory, no session initialised, no conditional choices and recursions at top-level, and no delegation. The characteristic of these processes is that, for each memory inside a process, there exists within the process an ancestor memory corresponding to the initialisation of the considered session. The type system only checks this latter kind of memories, which significantly simplifies the theory.

Coming back now to the multiple providers scenario, we can verify that the initial process is well-typed. In particular, the channel a_{login} can be typed by $\langle\, ?[\mathsf{Request}].\,![\mathsf{Quote}].\,\&[l_{acc} : \alpha_{acc}\,,\, l_{neg} : \alpha_{neg}\,,\, l_{rej} : \mathsf{end}]\,\rangle$, where sorts Request and Quote are used to type requests and quotes, respectively. Let us consider now a scenario where the client wills to concurrently submit two different requests to the same provider, which would concurrently serve them. Consider in particular the following specification of the client: $\overline{a_{login}}(x).\,(\,x!\langle \mathsf{srv\_req\_1}\rangle.P_1 \;\mid\; x!\langle \mathsf{srv\_req\_2}\rangle.P_2\,)$. The new specification is clearly not well-typed, due to the use of parallel threads within the same session. This permits avoiding mixing up messages related to different requests and wrongly delivering them. In order to properly concurrently submit separate requests, the client must instantiate separate sessions with the provider, one for each request.

5 Concluding remarks

To bring the benefits of reversible computing to structured communication-based programming, we have defined a theoretical framework based on π-calculus that can be used as formal basis for studying the interplay between (causal-consistent) reversibility and session-based structured interaction.

The type discipline for *ReSπ* is still subject of study. In fact, the type system mentioned in Section 4 is not completely satisfactory, because its use is limited to a restricted class of processes. To consider a broader class, an appropriate static type checking approach for memories has to be devised. For

each memory, we would check a term composed of the threads stored in the memory and of a context composed of threads that have not been generated by the execution of the memory threads.

Concerning the reversible calculus, we plan to investigate the definition of a syntactic characterisation of consistent terms, which statically enforces history consistency in memories (as in [7]). It is worth noticing that the calculus is *fully* reversible, i.e. backward computations are always enabled. Full reversibility provides theoretical foundations for studying reversibility in session-based π-calculus, but it is not suitable for a practical use. In line with [6], we plan to enrich the language with mechanisms to control reversibility. Moreover, we intend to enrich the framework with *irreversible* actions for closing sessions. In this way, computation would go backward and forward, allowing the parties to try different interactions, until the session closes. The closure would act as a commit, because it would be an irreversible action that does not permit undoing the session computation. For instance, the process P_{acc} in our example could terminate by performing the irreversible action $\overline{\mathsf{close}}$, which has to synchronise with the dual action close in Q_{acc}. No backward rule would be defined to revert this interaction. Differently from sessions terminated by $\mathbf{0}$, a session terminated by a close synchronisation is unbacktrackable. The type theory should be tailored to properly deal with this kind of session closure.

As longer-term goals, we intend to apply the proposed approach to other session-based formalisms, which consider, e.g., asynchronous sessions and multiparty sessions. Moreover, we plan to investigate implementation issues that may arise when incorporating the approach into standard programming languages, in particular in case of a distributed setting.

References

[1] Luca Cardelli and Cosimo Laneve. Reversible structures. In *CMSB*, pages 131–140. ACM, 2011.

[2] I. Cristescu, J. Krivine, and D. Varacca. A Compositional Semantics for the Reversible p-Calculus. In *LICS*, pages 388–397. IEEE, 2013.

[3] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.

[4] Vincent Danos and Jean Krivine. Formal Molecular Biology Done in CCS-R. *Electr. Notes Theor. Comput. Sci.*, 180(3):31–49, 2007.

[5] E. Giachino, I. Lanese, C.A. Mezzina, and F. Tiezzi. Causal-Consistent Reversibility in a Tuple-Based Distributed Language. Technical report, 2013. `http://www.cs.unibo.it/~giachino/bib/GLMT.pdf`.

[6] I. Lanese, C.A. Mezzina, A. Schmitt, and J. Stefani. Controlling Reversibility in Higher-Order Pi. In *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.

[7] I. Lanese, C.A. Mezzina, and J. Stefani. Reversing higher-order pi. In *CONCUR*, volume 6269 of *LNCS*, pages 478–493. Springer, 2010.

[8] Iain C. C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2):70–96, 2007.

[9] Francesco Tiezzi and Nobuko Yoshida. Towards Reversible Sessions. Technical report, 2014. `http://cse.lab.imtlucca.it/~tiezzi/reversible_sessions_TR.pdf`.

[10] N. Yoshida and V.T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Session Types for Broadcasting

Dimitrios Kouzapas[1], Ramūnas Gutkovas[2] and Simon J. Gay[1]

[1] University of Glasgow
[2] Uppsala University

Abstract

Up to now session types have been used under the assumptions of point to point communication, to ensure the linearity of session endpoints, and reliable communication, to ensure send/receive duality. In this paper we define a session type theory for broadcast communication semantics that by definition do not assume point to point and reliable communication. Our session framework lies on top of the parametric framework of broadcasting ψ-calculi, giving insights on developing session types within a parametric framework. Our session type theory enjoys the properties of soundness and safety. We further believe that the solutions proposed will eventually provide a deeper understanding of how session types principles should be applied in the general case of communication semantics.

1 Introduction

Session types [5, 7, 6] allow communication protocols to be specified as types and verified by type-checking. Up to now, session type systems have assumed reliable, point to point message passing communication. Reliability is important to maintain send/receive duality, and point to point communication is required to ensure session endpoint linearity.

In this paper we propose a session type system for unreliable broadcast communication. Developing such a system was challenging for two reasons: (i) we needed to extend binary session types to handle unreliability as well as extending the notion of session endpoint linearity, and (ii) the reactive control flow of a broadcasting system drove us to consider typing patterns of communication interaction rather than communication prefixes. The key ideas are (i) to break the symmetry between the s^+ and s^- endpoints of channel s, allowing s^+ (uniquely owned) to broadcast and gather, and s^- to be shared; (ii) to implement (and type) the gather operation as an iterated receive. We retain the standard binary session type constructors.

We use ψ-calculi [1] as the underlying process framework, and specifically we use the extension of the ψ-calculi family with broadcast semantics [2]. ψ-calculi provide a parametric process calculus framework for extending the semantics of the π-calculus with arbitrary data structures and logical assertions. Expressing our work in the ψ-calculi framework allows us to avoid defining a new operational semantics, instead defining the semantics of our broadcast session calculus by translation into a broadcast ψ-calculus. Establishing a link between session types and ψ-calculi is therefore another contribution of our work.

Intuition through Demonstration. We demonstrate the overall intuition by means of an example. For the purpose of the demonstration we imply a set of semantics, which we believe are self explanatory. Assume types $S = !T; ?T; \mathtt{end}$, $\overline{S} = ?T; !T; \mathtt{end}$ for some data type T, and typings $s^+ : S$, $s^- : \overline{S}$, $a : \langle S \rangle$, $v : T$. The session type prefix $!T$ means *broadcast* when used by s^+, and *single destination send* when used by s^-. Dually, $?T$ means *gather* when used by s^+, and *single origin receive* when used by s^-.

Session Initiation through broadcast, creating an arbitrary number of receiving endpoints:

$$\overline{a}s^-.P_0 \mid ax.P_1 \mid ax.P_2 \mid ax.P_3 \longrightarrow P_0 \mid P_1\{s^-/x\} \mid P_2\{s^-/x\} \mid ax.P_3$$

Due to unreliability, $ax.P_3$ did not initiate the session. We denote the initiating and accepting session endpoint as s^+ and s^- respectively.

Session Broadcast from the s^+ endpoint results in multiple s^- endpoints receiving:

$$s^+!\langle v\rangle; P_0 \mid s^-?(x); P_1 \mid s^-?(x); P_2 \mid s^-?(x); P_3 \longrightarrow P_0 \mid P_1\{v/x\} \mid P_2\{v/x\} \mid s^-?(x); P_3$$

Due to unreliability, a process (in the above reduction, process $s^-?(x); P_3$) might not receive a message. In this case the session endpoint that belongs to process $s^-?(x); P_3$ is considered broken, and later we will introduce a recovery mechanism.

Gather: The next challenge is to achieve the sending of values from the s^- endpoints to the s^+ endpoint. The gather prefix $s^+?(x); P_0$ is translated (in Section 4) into a process that iteratively receives messages from the s^- endpoints, non-deterministically stopping at some point and passing control to P_0.

$$s^+?(x); P_0 \mid s^-!\langle v_1\rangle; P_1 \mid s^-!\langle v_2\rangle; P_2 \mid s^-!\langle v_3\rangle; P_3 \longrightarrow^* P_0' \mid P_1 \mid s^-!\langle v_2\rangle; P_2 \mid P_3$$

with $P_0\{\{v_1, v_3\}/x\} \longrightarrow P_0'$.

After two reductions the messages from processes $s^-!\langle v_1\rangle; P_1$ and $s^-!\langle v_3\rangle; P_3$ had been received by the s^+ endpoint. On the third reduction the s^+ endpoint decided not to wait for more messages and proceeded with its session non-deterministically, resulting in a broken sending endpoint ($s^-!\langle v_2\rangle; P_2$), which is predicted by the unreliability of the broadcast semantics. The received messages, v_1 and v_2, were delivered to P_0 as a set.

Prefix Enumeration: The above semantics, although capturing broadcast session initiation and interaction, still violate session type principles due to the unreliability of communication:

$$\begin{array}{rcl} s^+!\langle v_1\rangle; s^+!\langle v_2\rangle; \mathbf{0} \mid s^-?(x); s^-?(y); \mathbf{0} \mid s^-?(x); s^-?(y); \mathbf{0} & \longrightarrow & \\ s^+!\langle v_2\rangle; \mathbf{0} \mid s^-?(y); \mathbf{0} \mid s^-?(x); s^-?(y); \mathbf{0} & \longrightarrow & \mathbf{0} \mid \mathbf{0} \mid s^-?(y); \mathbf{0} \end{array}$$

The first reduction produced a broken endpoint, $s^-?(x); s^-?(y); \mathbf{0}$, while the second reduction reduces the broken endpoint. This situation is not predicted by session type principles. To solve this problem we introduce an enumeration on session prefixes:

$$\begin{array}{l} (s^+, 1)!\langle v_1\rangle; (s^+, 2)!\langle v_2\rangle; \mathbf{0} \mid (s^-, 1)?(x); (s^-, 2)?(y); \mathbf{0} \mid (s^-, 1)?(x); (s^-, 2)?(y); \mathbf{0} \longrightarrow \\ (s^+, 2)!\langle v_2\rangle; \mathbf{0} \mid (s^-, 2)?(y); \mathbf{0} \mid (s^-, 1)?(x); (s^-, 2)?(y); \mathbf{0} \longrightarrow \mathbf{0} \mid \mathbf{0} \mid (s^-, 1)?(x); (s^-, 2)?(y); \mathbf{0} \end{array}$$

The intuitive semantics described in this example are encoded in the ψ-calculi framework. From this it follows that all the operational semantics, typing system and theorems are stated using the ψ-calculus framework.

Contributions. This paper is the first to propose session types as a type meta-theory for the ψ-calculi. Applying session semantics in such a framework meets the ambition that session types can effectively describe general communication semantics. A step further is the development of a session type framework for broadcast communication semantics. It is the first time that session types escape the assumptions of point to point communication and communication reliability. We also consider as a contribution the fact that we use enumerated session prefixes in order to maintain consistency of the session communication. We believe that this technique will be applied in future session type systems that deal with unreliable and/or unpredictable communication semantics.

Related Work. Carbone *et al.* [4] extended binary session types with exceptions, allowing both parties in a session to collaboratively handle a deviation from the standard protocol. Capecchi *et al.* [3] generalized a similar approach to multi-party sessions. In contrast, our recovery processes allow a broadcast sender or receiver to autonomously handle a failure of communication. Although it might be possible to represent broadcasting in multi-party session type systems, by explicitly specifying separate messages from a single source to a number of receivers, all such systems assume reliable communication for every message.

2 Broadcast Session Calculus

We define an intuitive syntax for our calculus. The syntax below will be encoded in the ψ-calculi framework so that it will inherit the operational semantics.

$$\begin{array}{rcl} P, R & ::= & \overline{a}s^-.P \mid ax.P \mid s^+!\langle v\rangle; P \mid s^-!\langle v\rangle; P \mid s^+?(x); P \mid s^-?(x); P \\ & \mid & s^+ \oplus l; P \mid s^-\&\{l_i : P_i\} \mid P \bowtie R \mid \mathbf{0} \mid \mu X.P \mid X \mid P \mid P \mid (\nu\ n)P \end{array}$$

Processes $\overline{a}s^-.P$, $ax.P$ are prefixed with session initiation operators that interact following the broadcast semantics. Processes $s^+!\langle v\rangle; P$, $s^-!\langle v\rangle; P$ define two different sending patterns. For the s^+ endpoint we have a broadcast send. For the s^- endpoint we have a unicast send. Processes $s^+?(x); P$, $s^-?(x); P$ assume gather (i.e. the converse of broadcast send) and unicast receive, respectively. We allow selection and branching $s^+ \oplus l; P$, $s^-\&\{l_i : P_i\}$ only for broadcast semantics from the s^+ to the s^- endpoint. Each process can carry a recovery process R with the operator $P \bowtie R$. The process can proceed non-deterministically to recovery if the session endpoint is broken due to the unreliability of the communication. Process R is carried along as process P reduces its prefixes. The rest of the processes are standard π-calculus processes.

Structural congruence is defined over the abelian monoid defined by the parallel operator ($\mid$) and the inactive process ($\mathbf{0}$) and additionally satisfies the rules:

$(\nu\ n)\mathbf{0} \equiv \mathbf{0}$ $\quad$ $P \mid (\nu\ n)Q \equiv (\nu\ n)(P \mid Q)$ if $n \notin \mathtt{fn}(P)$

3 Broadcast ψ-Calculi

Here we define the parametric framework of ψ-calculi for broadcast. For a detailed description of ψ-calculi we refer the reader to [1].

We fix a countably infinite set of names $\mathcal{N}$ ranged over by a, b, x. ψ-calculi are parameterised over three nominal sets: terms ($\mathbf{T}$ ranged over by M, N, L), conditions ($\mathbf{C}$ ranged over by φ), and assertions ($\mathbf{A}$ ranged over by Ψ); and operators: channel equivalence, broadcast output and input connectivity $\leftrightarrow, \dot{\prec}, \dot{\succ}$: $\mathbf{T} \times \mathbf{T} \to \mathbf{C}$, assertion composition $\otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$, unit $\mathbf{1} \in \mathbf{A}$, entailment relation $\vdash \mathbf{A} \times \mathbf{C}$, and a substitution function substituting terms for names for each set. The channel equivalence is required to be symmetric and transitive, and assertion composition forms abelian monoid with $\mathbf{1}$ as the unit element. We do not require output and input connectivity be symmetric, i.e., $\Psi \vdash M \mathrel{\dot{\prec}} N$ is not equivalent to $\Psi \vdash N \mathrel{\dot{\succ}} M$, however for technical reasons require that the names of L should be included in N and M whenever $\Psi \vdash N \mathrel{\dot{\prec}} L$ or $\Psi \vdash L \mathrel{\dot{\succ}} M$. The agents are defined as follows

$$P, Q \ ::= \ M(\lambda\widetilde{a})N.P \mid \overline{M}N.P \mid \mathbf{case}\,\varphi_1 : P_1 \,[]\, \ldots \,[]\, \varphi_n : P_n \mid (\!|\Psi|\!) \mid (\nu a)P \mid P \mid Q \mid\ !P$$

where $\widetilde{a}$ bind into N and P. The assertions in the case and replicated agents are required to be guarded. We abbreviate the case agent as $\mathbf{case}\,\widetilde{\varphi} : \widetilde{P}$; we write $\mathbf{0}$ for $(\!|\mathbf{1}|\!)$, we also write $a\#X$ to intuitively mean that name a does not occur freely in X.

We give a brief intuition behind the communication parameters: Agents unicast whenever their subject of their prefixes are channel equivalent, to give an example, $\overline{M}L.P$ and $N(\lambda\widetilde{a})K.Q$ communicate whenever $\Psi \vdash M \leftrightarrow N$. In contrast, broadcast communication is mediated by a broadcast channel, for example, the agents $\overline{M}N.P$ and $M_i(\lambda\widetilde{a_i})N_i.P_i$ (for $i > 0$) communicate if they can broadcast and receive from the same channel $\Psi \vdash M \mathrel{\dot{\prec}} K$ and $\Psi \vdash K \mathrel{\dot{\succ}} M_i$.

In addition to the standard structural congruence laws of pi-calculus we define the following, with the assumption that $a \# \widetilde{\varphi}, M, N, \widetilde{x}$ and π is permutation of a sequence.

$$(\nu a)\mathbf{case}\,\widetilde{\varphi} : \widetilde{P} \equiv_\Psi \mathbf{case}\,\widetilde{\varphi} : \widetilde{(\nu a)P} \qquad \mathbf{case}\,\widetilde{\varphi} : \widetilde{P} \equiv_\Psi \mathbf{case}\,\pi \cdot (\widetilde{\varphi} : \widetilde{P})$$
$$\overline{M}N.(\nu a)P \equiv_\Psi (\nu a)\overline{M}N.P \qquad M(\lambda\widetilde{x})N.(\nu a)P \equiv_\Psi (\nu a)M(\lambda\widetilde{x})N.P$$

The following is a reduction context with two types of numbered holes (condition hole $\hat{[]}$ and process hole $[]$) such that no two holes of the same type have the same number.

$$C \;::=\; (\mathbf{case}\,\hat{[]}_j : C \,[]\, \widetilde{\varphi} : \widetilde{P}) \mid C \quad | \quad \prod_{k>0} []_{i_k}$$

The filling of the holes is defined in the following way: filling a process (resp. condition) hole with a assertion guarded process (resp. condition) taken from the number position of a given sequence. We denote filling of holes as $C[(\varphi_i)_{i\in I}; (P_j)_{j\in J}; (Q_k)_{k\in K}]$ where the first component is for filling the condition holes and the other two are for filling process holes.

We require that I is equal to the numbering set of condition holes and furthermore J and K are disjoint and their union is equal to the numbering set of context for the process holes. We also require that every J numbered hole is either in parallel with any of the K holes or is parallel to **case** where recursively a K numbered hole can be found. When the numbering is understood we simply write $C[\widetilde{\varphi}; \widetilde{P}; \widetilde{Q}]$.

In the following we define reduction semantics of ψ-calculi, in addition to the standard labelled transition semantics [1]. The two rules describe unicast and broadcast semantics. We identify agents up to structural congruence, that is, we also assume the rule such that two agents reduce if their congruent versions reduce. In the broadcast rule, if for some $a \in \widetilde{a}$, $a \in n(K)$, then $\widetilde{b} = \widetilde{a}$, otherwise $\widetilde{b} = \widetilde{a} \setminus n(N)$. To simplify the presentation we abbreviate $\widetilde{\prod (\!|\Psi|\!)}$ as $(\!|\hat{\Psi}|\!)$ and $\otimes_i \Psi_i$ as $\hat{\Psi}$. We prove that reductions correspond to silent and broadcast transitions.

$$\frac{N' = N[\widetilde{x} := \widetilde{L}] \text{ and } \hat{\Psi} \vdash M \leftrightarrow M' \text{ and } \forall i. \hat{\Psi} \vdash \varphi_i}{(\nu \widetilde{a})(C[\widetilde{\varphi};\ \widetilde{R};\ M(\lambda \widetilde{x})N.P,\ \overline{M'}N'.Q] \mid (\!|\hat{\Psi}|\!)) \quad \to \quad (\nu \widetilde{a})(P[\widetilde{x} := \widetilde{L}] \mid Q \mid \prod \widetilde{R} \mid (\!|\hat{\Psi}|\!))}$$

$$\frac{\hat{\Psi} \vdash M \dot{\prec} K \text{ and } \forall i. \hat{\Psi} \vdash K \dot{\succ} M'_i \text{ and } N'_i[\widetilde{x}_i := \widetilde{L}_i] = N \text{ and } \forall j. \hat{\Psi} \vdash \varphi_j}{(\nu \widetilde{a})(C[\widetilde{\varphi};\ \widetilde{R};\ \overline{M}N.P, \widetilde{(M'(\lambda \widetilde{x})N'.Q)}] \mid (\!|\hat{\Psi}|\!)) \quad \to \quad (\nu \widetilde{b})(P \mid \prod_i Q_i[\widetilde{x}_i := \widetilde{L}_i] \mid \prod \widetilde{R} \mid (\!|\hat{\Psi}|\!))}$$

Theorem 3.1. *Let α be either a silent or broadcast output action. Then, $\mathbf{1} \triangleright P \xrightarrow{\alpha} P'$ iff $P \to P'$*

Proof Sketch. The complicated direction is $\Longrightarrow$. One needs to prove similar results for the other actions, and then demonstrate that they in parallel have the right form. □

4 Translation of Broadcast Calculus to Broadcast ψ-Calculus

The semantics for the broadcast session calculus are given as an instance of the ψ-calculi with broadcast [2]. To achieve this effect we define a translation between the syntax of § 2 and a particular instance of the ψ-calculi. Operational semantics are then inherited by the ψ-calculi framework.

We fix the set of labels $\mathcal{L}$ and ranged over by $l, l_1, l_2 \ldots$. The following are the nominal sets

$$\begin{array}{lcl} \mathbf{T} & = & \mathcal{N} \cup \{*\} \cup \{(n^p, k), (n^p, i), (n^p, k, \mathsf{u}), (n^p, l, k), n \cdot k \;\mid\; n, k \in \mathbf{T} \wedge i \in \mathbb{N} \wedge l \in \mathcal{L} \wedge p \in \{+, -\}\} \\ \mathbf{C} & = & \{t_1 \leftrightarrow t_2, t_1 \dot{\prec} t_2, t_1 \dot{\succ} t_2 \;\mid\; t_1, t_2 \in \mathbf{T}\} \cup \{\mathsf{true}\} \\ \mathbf{A} & = & \mathbf{T} \to \mathbb{N} \end{array}$$

We define the $\otimes$ operator (here defined as multiset union) and the $\vdash$ relation:

$$(f \otimes g)(n) = \begin{cases} f(n) + g(n) & \text{if } n \in dom(f) \cap dom(g) \\ f(n) & \text{if } n \in dom(f) \\ g(n) & \text{if } n \in dom(g) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\begin{array}{lcll} \Psi & \vdash & (s^{p_1}, k, \mathsf{u}) \leftrightarrow (s^{p_2}, j, \mathsf{u}) \text{ iff } \Psi(k) = \Psi(j) \\ \Psi & \vdash & (s^{+}, k) \dot{\prec} (s^{+}, i) \text{ iff } \Psi(k) = i \\ \Psi & \vdash & (s^{+}, i) \dot{\succ} (s^{-}, k) \text{ iff } \Psi(k) = i \\ \Psi & \vdash & \mathsf{true} \qquad \Psi \vdash a \leftrightarrow a \in \mathcal{N} \end{array}$$

It can be easily checked that the definition is indeed a broadcast ψ-calculus. We write $\Sigma_{i\in I}P$ as a shorthand for $\widetilde{\mathbf{case}\ \mathsf{true} : \widetilde{P}}$, and $P + Q$ for $\mathbf{case}\ \mathsf{true} : P \,[]\, \mathsf{true} : Q$

The translation is parameterised by ρ, which tracks the enumeration of session prefixes, represented by multisets of asserted names $(\!|k|\!)$. The replication in $s^+?(x, u^i); P$ implements the iterative broadcast receive. We annotated the prefixes $s^+?(x, u^i);^b P$ and $\mu X^b.P \bowtie R$ with $b \in \{0, 1\}$ to capture their translation as a two step (0 and 1) iterative process. The recovery process can be chosen in a non-deterministic way instead of a s^- prefix. Otherwise it is pushed in the continuation of the translation.

$$
\begin{array}{l}
[\![\overline{a}s^-.P \bowtie R]\!]_\rho = (\nu k)(\overline{a}s^-.[\![P \bowtie R]\!]_{\rho\cup\{s^+:k\}}) \qquad [\![ax.P \bowtie R]\!]_\rho = (\nu k)(a(\lambda x)x.[\![P \bowtie R]\!]_{\rho\cup\{s^-:k\}}) \\
[\![s^+!\langle v\rangle; P \bowtie R]\!]_{\rho\cup\{s^+:k\}} = \overline{(s^+, k)}v.([\![P \bowtie R]\!]_{\rho\cup\{s^+:k\}} \mid (\!|k|\!)) \\
[\![s^-!\langle v\rangle; P \bowtie R]\!]_{\rho\cup\{s^-:k\}} = \overline{(s^-, k, \mathsf{u})}v.([\![P \bowtie R]\!]_{\rho\cup\{s^-:k\}} \mid (\!|k|\!)) + [\![R]\!]_{\rho\cup\{s^-:k\}} \\
\begin{array}{rl}
[\![s^+?(x, u);^0 P \bowtie R]\!]_{\rho\cup\{s^+:k\}} = & (\nu\ n)(\overline{n}u.\mathbf{0} \mid !(n(\lambda x)x.((s^+, k, \mathsf{u})(\lambda y)y.\overline{n}(x \cdot y).\mathbf{0}) \\
& +\tau.([\![P \bowtie R]\!]_{\rho\cup\{s^+:k\}} \mid (\!|k|\!)))) \\
[\![s^+?(x, u);^1 P \bowtie R]\!]_{\rho\cup\{s^+:k\}} = & (\nu\ n)(((s^+, k, \mathsf{u})(\lambda y)y.\overline{n}(u \cdot y).\mathbf{0}) + \tau.([\![P \bowtie R]\!]_{\rho\cup\{s^+:k\}}[x := u] \mid (\!|k|\!)) \\
& \mid !(n(\lambda x)x.((s^+, k, \mathsf{u})(\lambda y)y.\overline{n}(x \cdot y).\mathbf{0}) + \tau.([\![P \bowtie R]\!]_{\rho\cup\{s^+:k\}} \mid (\!|k|\!))))
\end{array} \\
[\![s^-?(x); P \bowtie R]\!]_{\rho\cup\{s^-:k\}} = (s^-, k)(\lambda x)x.([\![P \bowtie R]\!]_{\rho\cup\{s^-:k\}} \mid (\!|k|\!)) + [\![R]\!]_{\rho\cup\{s^-:k\}} \\
[\![s^+ \oplus l; P \bowtie R]\!]_{\rho\cup\{s^-:k\}} = \overline{(s^+, l, k)} * .([\![P \bowtie R]\!]_{\rho\cup\{s^+:k\}} \mid (\!|k|\!)) \qquad [\![(\!|k|\!)]\!]_{\rho\cup\{s^p:k\}} = (\!|k|\!) \\
[\![s^-\&\{l_i : P_i\}_{i\in I} \bowtie R]\!]_{\rho\cup\{s^-:k\}} = \Sigma_{i\in I}(s^-, l_i, k)(\lambda) * .([\![P_i \bowtie R]\!]_{\rho\cup\{s^-:k\}} \mid (\!|k|\!)) + [\![R]\!]_{\rho\cup\{s^-:k\}} \\
[\![\mu X^0.P \bowtie R]\!]_\rho = (\nu\ n)(!(n(\lambda) * .[\![P \bowtie R]\!]_{\rho\cup\{X:n\}}) \mid \overline{n} * .\mathbf{0}) \\
[\![\mu X^1.P \bowtie R]\!]_\rho = (\nu\ n)([\![P \bowtie R]\!]_{\rho\cup\{X:n\}} \mid !(n(\lambda) * .[\![P \bowtie R]\!]_{\rho\cup\{X:n\}})) \\
[\![X]\!]_{\rho\cup\{X:n\}} = \overline{n} * .\mathbf{0} \quad [\![\mathbf{0}]\!]_\rho = \mathbf{0} \quad [\![\mathbf{0} \bowtie R]\!]_\rho = \mathbf{0} \quad [\![P \mid Q]\!]_\rho = [\![P]\!]_\rho \mid [\![Q]\!]_\rho \quad [\![(\nu\ n)P]\!]_\rho = (\nu n)[\![P]\!]_\rho
\end{array}
$$

The encoding respects the following desirable properties.

Lemma 4.1 (Encoding Properties)**.** Let P be a session broadcast process.

1. $[\![P[x := v]]\!] = [\![P]\!][x := v]$
2. $[\![P]\!] \to Q$ implies that for a session broadcast process $P', Q \equiv_\Psi [\![P']\!]$.

5 Broadcast Session Types

Broadcast session types syntax is identical to classic binary session type syntax (cf. [7]), with the exception that we do not allow session channel delegation. We assume the duality relation as defined in [7]. Note that we do not need to carry the session prefix enumeration in the session type system or semantics. Session prefix enumeration is used operationaly only to avoid communication missmatch.

$$
\begin{array}{lcl}
S & ::= & !U; S \mid ?U; S \mid \oplus\{l_i : S_i\}_{i\in I} \mid \&\{l_i : S_i\}_{i\in I} \mid \mathtt{end} \mid \mathsf{X} \mid \mu\mathsf{X}.S \\
U & ::= & \langle S\rangle \mid [U]
\end{array}
$$

Typing judgements are: $\Gamma \vdash P$ read as P is typed under environment Γ, with

$$
\Delta ::= \emptyset \mid \Delta \cdot s^p : S \qquad \Gamma ::= \emptyset \mid \Gamma \cdot a : \langle S\rangle \mid \Gamma \cdot s^p : S \mid \Gamma \cdot \mathsf{X} : \Delta
$$

Δ environments map only session names to session types, while Γ maps shared names to shared types, session names to session types and process variables to Δ mappings.

The rules below define the broadcast session type system:

$$
\Gamma \cdot n : U \vdash n : U\ [\text{Name}] \quad \frac{\Gamma \vdash P \quad s^p \notin \mathtt{fn}(P)}{\Gamma \cdot s^p : \mathtt{end} \vdash P}\ [\text{Weak}] \quad \frac{s \notin \mathtt{dom}(\Gamma)}{\Gamma \vdash \mathbf{0}}\ [\text{Inact}] \quad \frac{\Gamma \vdash R \quad s^p \notin \mathtt{dom}(\Gamma)}{\Gamma \vdash \mathbf{0} \bowtie R}\ [\text{Recov}]
$$

$$\frac{\Gamma \vdash a : \langle S \rangle \quad \Gamma \vdash s^+ : S \quad \Gamma \vdash P}{\Gamma \cdot s^- : \overline{S} \vdash \overline{a} s^-.P} \text{ [BInit]} \quad \frac{\Gamma \vdash a : \langle S \rangle \quad \Gamma \cdot x : \overline{S} \vdash P}{\Gamma \vdash a x.P} \text{ [BAcc]}$$

$$\frac{\Gamma \cdot s^+ : S \vdash P \bowtie R \quad \Gamma \vdash v : \langle S' \rangle}{\Gamma \cdot s^+ : !\langle S' \rangle; S \vdash s^+ ! \langle v \rangle; P \bowtie R} \text{ [BSend]} \quad \frac{\Gamma \cdot s^- : S \vdash P \bowtie R \quad \Gamma \vdash v : \langle S' \rangle \quad s^- \notin \mathtt{dom}(\Gamma)}{\Gamma \cdot s^- : !\langle S' \rangle; S \vdash s^- ! \langle v \rangle; P \bowtie R} \text{ [USend]}$$

$$\frac{\Gamma \cdot s^+ : S \cdot x : \langle S' \rangle \vdash P \bowtie R \quad \Gamma \vdash u : [\langle S' \rangle]}{\Gamma \cdot s^+ : ?\langle S' \rangle; S \vdash s^+ ? (x, u);^b P \bowtie R} \text{ [URcv]} \quad \frac{\Gamma \cdot s^- : S \cdot x : \langle S' \rangle \vdash P \bowtie R \quad s^- \notin \mathtt{dom}(\Gamma)}{\Gamma \cdot s^- : ?\langle S' \rangle; S \vdash s^- ? (x); P \bowtie R} \text{ [BRcv]}$$

$$\frac{\Gamma \cdot s^+ : S_k \vdash P \bowtie R \quad k \in I}{\Gamma \cdot s^+ : \oplus\{l_i : S_i\}_{i \in I} \vdash s^+ \oplus l_k; P \bowtie R} \text{ [Sel]} \quad \frac{\Gamma \cdot s^- : S_i \vdash P_i \bowtie R \quad s^- \notin \mathtt{dom}(\Gamma)}{\Gamma \cdot s^- : \&\{l_i : S_i\}_{i \in I} \vdash s^- \& \{l_i : P_i\}_{i \in I} \bowtie R} \text{ [Bra]}$$

$$\frac{\Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2 \quad s^+ \notin \mathtt{dom}(\Gamma_1) \cap \mathtt{dom}(\Gamma_2)}{\Gamma_1 \cup \Gamma_2 \vdash P_1 \mid P_2} \text{ [Par]} \quad \frac{\Gamma \cdot s^+ : S \cdot \{s^- : \overline{S_i}\}_{i \in I} \vdash P \quad S = S_i}{\Gamma \vdash (\nu\ s) P} \text{ [SRes]}$$

$$\frac{\Gamma \cdot a : \langle S \rangle \vdash P}{\Gamma \vdash (\nu\ a) P} \text{ [ShRes]} \quad \frac{\Gamma \cup \Delta \cdot X : \Delta \vdash P \quad s^p \notin \mathtt{dom}(\Gamma)}{\Gamma \cup \Delta \vdash \mu X^b.P} \text{ [Rec]} \quad \Gamma \cup \Delta \cdot X : \Delta \vdash X \text{ [RVar]}$$

Rule [Recov] types the recovery process. We expect no free session names in a recover process. Rules [BInit], [BAcc], [BSend], [Usend], [BRcv], [BRcv], [Sel] and [Bra] type prefixes in the standard way, i.e. check for object and the subject type match. Rule [URcv] types both binary instances of the unicast receive prefix with the same type. We require that the recovery process is carried and typed inductively in the structure of a process. A recovery process must not (re)use any session endpoints ([Recov]). Also we require the s^- to be the only one in Γ. Multiple s^- endpoints are collected using the [Par] rule. The [Par] rule expects that there is no duplicate s^+ endpoint present inside a process. When restricting a session name we check endpoint s^+ and the set of endpoints s^- to have dual types. The rest of the rules are standard.

5.1 Soundness and Safety

We use the standard notion of a context $\mathcal{C}$ on session types S with a single hole denoted as $[]$. We write $\mathcal{C}[S]$ for filling a hole in C with the type S. We define the set of non-live sessions in a context as $d(\Gamma) = \{s^- : S \mid s^+ : S' \in \Gamma \text{ and } \overline{S} = C[S'] \text{ with } C \neq []\}$ and live $l(\Gamma) = \Gamma \setminus d(\Gamma)$. We say that Γ is well typed iff $\forall s^+ : S \in l(\Gamma)$ then $\{s^- : \overline{S_i}\}_{i \in I} \subset l(\Gamma)$ with $S = S_i$ or $S = ?U; S_i$.

Theorem 5.1 (Subject Congruence)**.** If $\Gamma \vdash P$ with Γ well typed and $P \equiv P'$ then $\Gamma \vdash P'$.

Theorem 5.2 (Subject Reduction)**.** If $\Gamma \vdash P$ with Γ well typed, $dom(\rho) \subseteq dom(\Gamma)$ and $[\![P]\!]_\rho \to Q$, then there is P' such that $[\![P']\!]_\rho \equiv_\Psi Q$, $\Gamma' \vdash P'$ and Γ' well typed with either $\Gamma' = d(\Gamma) \cup l(\Gamma')$ or $\Gamma' = d(\Gamma) \setminus \{s^- : S\} \cup l(\Gamma')$ or $\Gamma' = d(\Gamma) \cup \{s^- : S\} \cup l(\Gamma')$.

Definition 5.1 (Error Process)**.** Let s-prefix processes to have the following form:

1. $s^+ ! \langle v \rangle; P$ 2. $s^+ \oplus l; P$ 3. $s^+ ? (x); P$ 4. $\prod_{i \in I} s^- ? (x); P_i \mid \prod_{j \in J} C_j[s^- ? (x); P_j]$
5. $\prod_{i \in I} s^- ! \langle v_i \rangle; P_i \mid \prod_{k \in K} P_k \mid \prod_{j \in J} C_j[s^- ? (x); P_j]$
   where $\prod_{i \in I} P_i \mid \prod_{k \in K} P_k \mid \prod_{j \in J} C_j[s^- ? (x); P_j]$ forms an s-redex.
6. $\prod_{i \in I} s^- \& \{l_k : P_k\}_{k \in K_i} \mid \prod_{j \in J} C_j[s^- \& \{l_k : P_k\}_{k \in K_j}]$

with $C_j[]$ being a context that contains s^- prefixes.

A valid s-redex is a parallel composition of either s-prefixes 1 and 4, s-prefixes 2 and 6, or s-prefixes 3 and 5. Every other combination of s-prefixes is invalid. An error process is a process of the form $P \equiv (\nu\ \tilde{n})(R \mid Q)$ where R is an invalid s-redex and Q does not contain any other s-prefixes.

Theorem 5.3 (Type Safety)**.** A well typed process will never reduce into an error process.

Proof. The proof is a direct consequence of the Subject Reduction Theorem (5.2) since error process are not well typed. □

6 Conclusion

We have defined a system of session types for a calculus based on unreliable broadcast communication. This is the first time that session types have been generalised beyond reliable point-to-point communication. We defined the operational semantics of our calculus by translation into an instantiation of broadcast ψ-calculi, and proved subject reduction and safety results. The use of the ψ-calculi framework means that we can try to use its general theory of bisimulation for future work on reasoning about session-typed broadcasting systems. The definition of a session typing system is also a new direction for the ψ-calculi framework.

Acknowledgements Kouzapas and Gay are supported by the UK EPSRC project "From Data Types to Session Types: A Basis for Concurrency and Distribution" (EP/K034413/1). This research was supported by a Short-Term Scientific Mission grant from COST Action IC1201 (Behavioural Types for Reliable Large-Scale Software Systems).

References

[1] J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: Mobile processes, nominal data, and logic. In *LICS*, pages 39–48, 2009.

[2] J. Borgström, S. Huang, M. Johansson, P. Raabjerg, B. Victor, J. Åman Pohjola, and J. Parrow. Broadcast psi-calculi with an application to wireless protocols. In G. Barthe, A. Pardo, and G. Schneider, editors, *SEFM*, volume 7041 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2011.

[3] S. Capecchi, E. Giachino, and N. Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 2014. To appear.

[4] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions in session types. In *CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.

[5] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[6] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.

[7] N. Yoshida and V. T. Vasconcelos. Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electr. Notes Theor. Comput. Sci.*, 171(4):73–93, 2007.

Multiparty Session Actors

Rumyana Neykova Nobuko Yoshida
Imperial College London

Actor coordination armoured with a suitable protocol description language has been a pressing problem in the actors community. We study the applicability of multiparty session type (MPST) protocols for verification of actor programs. We incorporate sessions to actors by introducing minimum additions to the model such as the notion of actor roles and protocol mailbox. The framework uses Scribble, which is a protocol description language based on multiparty session types. Our programming model supports actor-like syntax and runtime verification mechanism guaranteeing type-safety and progress of the communicating entities. An actor can implement multiple roles in a similar way as an object can implement multiple interfaces. Multiple roles allow for inter-concurrency in a single actor still preserving its progress property. We demonstrate our framework by designing and implementing a session actor library in Python and its runtime verification mechanism.

1 Introduction

The actor model is (re)gaining attention in the research community and in the mainstream programming languages as a promising concurrency paradigm. Unfortunately, in spite of the importance of message passing mechanisms in the actor paradigm, the programming model itself does not ensure correct sequencing of interactions between different computational processes. This is a serious problem when designing safe concurrent and distributed systems in languages with actors.

To overcome this problem, we need to solve several shortcomings existing in the actor programming models. First, although actors often have multiple states and complex policies for changing states, no general-purpose specification language is currently in use for describing actor protocols. Second, a clear guidance on actor discovery and coordination of distributed actors is missing. As a study published in [13] reveals, this leads to adhoc implementations and mixing the model with other paradigms which weaken its benefits. Third, no verification mechanism (neither static nor dynamic) is proposed to ensure correct sequencing of actor interactions. Most actor implementations provide static typing within a single actor, but the communication between actors – the complex communication patterns that are most likely to deadlock – are not checked.

We tackle the aforementioned challenges by studying applicability of multyparty session types (MPST) verification and its practical incarnation, the protocol description language Scribble, to actor systems. Recent works from [8, 6] prove suitability of MPST for dynamic verification of real world complex protocols [9]. The verification mechanism is applied to large cyberinfrastructure, but checks are restricted only to MPST primitives. In this paper, we take MPST verification one step further and apply it to an actor model by extending it with creation and management of communication contexts (protocols).

Our programming model is grounded on three design ideas: (1) use Scribble protocols and their relation to finite state machines for specification and runtime verification of actor interactions; (2) augment actor messages and their mailboxes dynamically with protocol (role) information; and (3) propose an algorithm based on virtual routers (protocol mailboxes) for the dynamic discovery of actor mailboxes within a protocol. We implement a session actor library in Python to demonstrate the applicability of the approach. To the best of our knowledge, this is the first design and implementation of session types and their dynamic verification toolchain in an actor library.

To appear in EPTCS.

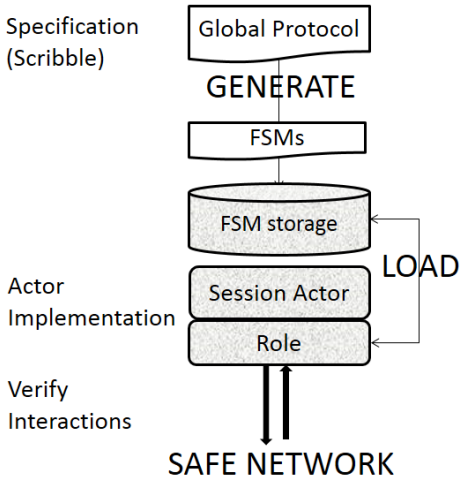


Figure 1: Development methodology

2 Multiparty Session Actor Framework and Programming

2.1 Overview of Multiparty Session Actor Framework

The standard development methodology for MPST verification is illustrated in Fig. 1, and the contributions of this work are darked. Distributed protocols are specified in Scribble, which collectively defines the admissible communication behaviours. Scribble tool can algorithmically project any global conversation to the specifications of its endpoints, generating finite state machines (FSMs). This is different from previous works [8, 6], where Scribble compiler produces local Scribble protocols and FSMs are generated on the fly at runtime. This difference is important for the applicability of our framework to general actor programming.

Distributed components are realised as session actors associated with one or more of the generated FSMs. The association is done through annotating the actor type (class) and receive messages (methods) with protocol information. Annotating the actor type with protocol information results in registering the type for a particular role. When a session is started, a join message is sent (in a round-robin style) to all registered actors. When a join message is received, the generated FSM is loaded into the actor role and all subsequent messages on that protocol (role) are tracked and checked. Message receive is delegated to the appropriate FSM via pattern matching on the protocol id, contained in the message. If all actors messages comply to their assigned FSMs, the whole communication is guaranteed to be safe. If participants do not comply, violations (such as deadlocks, communication mismatch or breakage of a protocol) are detected and delegated to a Policy actor.

2.2 Warehouse Management Protocol in Session Actors

To illustrate and motivate central design decisions of our model, we present the buyer-seller protocol from [5] and extend it to a full warehouse management scenario. A warehouse consists of multiple customers communicating to a warehouse provider. It can be involved in purchase protocol (with customers), but can also be involved in a loaded protocol with dealers to update its storage.

Scribble Protocol. Fig. 2 shows the interactions between the entities in the system written as a Scribble protocol. There are purchase and storeload protocols, involving three (a Buyer (B), a Seller (S) and an Authenticator (A)) and two (a Store (S), a Dealer (D)) parties, respectively. At the start of a purchase session, B sends login details to S, which delegates the request to an Authentication server. After the authentication is completed, B sends a request quote for a product to S and S replies with the

```
1  global protocol Purchase
2    (role B, role S, role A)
3  {
4    login(string:user) from B to S;
5    login(string:user) from S to A;
6    authenticate(string:token) from A to B, S;
7    choice at B
8      {request(string:product) from B to S;
9      (int:quote) from S to B;}
10   or
11     {buy(string:product) from B to S
12      delivery(string) from S to B; }
13   or
14     {quit() from B to S; }}
15 global protocol StoreLoad
16   (role D, role S)
17 {
18   rec Rec{
19   choice at S
20     {request(string:product, int:n) from S to D;)
21      put(string:product, int:n) from D to S;
22      continue Rec;}
23   or
24     {quit() from S to D;
25      acc() from D to S;}}}
```

Figure 2: Global protocols in Scribble

```
1  @protocol(c, Purchase, seller, buyer, auth)
2  @protocol(c1, StoreLoad, seller, dealer)
3  class Warehouse(SessionActor):
4
5    @role(c, buyer)
6    def login(self, user):
7      c.auth.send.login(user)
8
9    @role(c, buyer)
10   def buy(self, product):
11     self.purchaseDB[product]-=1;
12     c.seller.send.delivery(product.details)
13     self.become(update, product)
14
15   @role(c, buyer)
16   def quit(self):
17     c.send.buyer.acc()
18
19   @role(c1, self)
20   def update(self, product):
21     c1.dealer.send.request(product, n)
22
23   @role(c1, dealer)
24   def put(self, c, product):
25     self.purchaseDB[product]+=1:
```

Figure 3: Session Actor for warehouse

product price. Then B has a choice to ask for another product, to proceed with buying the product, or to quit. By buying a product the warehouse decreases the product amount it has in the store. When a product is out of stock, the warehouse sends a product request to a dealer to load the store with n numbers of that product (following the storeload protocol). The reader can refer to [11] for the full specification of Scribble syntax.

Challenges. There are several challenging points to implement the above scenario. First, a warehouse implementation should be involved in both protocols, therefore it can play more than one role. Second, initially the user does not know the exact warehouse it is buying from, therefore the customer learns dynamically the address of the warehouse. Third, there can be a specific restriction on the protocol that cannot be expressed as system constants (such as specific timeout depending on the customer). The next section explains implementations of Session Actors in more details.

Session Actor. Fig. 3 presents an implementation of a warehouse service as a single session actor that keeps the inventory as a state (*self.purchaseDB*). Lines 1- 2 annotate the session actor class with two protocol decorators – *c* and *c1* (for seller and store roles respectively). *c* and *c1* are accessible within the warehouse actor and are holders for mailboxes of the other actors, involved in the two protocols. All message handlers are annotated with a role and for convenience are implemented as methods. For example, the login method (Line 6) is invoked when a *login* message (Line 4, Fig. 2) is sent. The role annotation for *c* (Line 5) specifies the sender to be *buyer*. The handler body continues following Line 5, Fig. 2 - sending a *login* message via the *send* primitive to the session actor, registered as a role *auth* in the protocol of *c*. Value *c.auth* is initialised with the *auth* actor mailbox as a result of the actor discovery mechanism (explained in the next section). The handling of *authenticate* (Line 6, Fig. 2) and *request*

(Line 8, Fig. 2) messages is similar, so we omit it and focus on the *buy* handler (Line 10- 13), where after sending the delivery details (Line 12), the warehouse actor sends a message to itself (Line 13) using the primitive *become* with value *update*. Value *update* is annotated with another role *c1*, but has as a sender *self*. This is the mechanism used for switching between roles within an actor. Update method (Line 20-21, Fig. 3) implements the request branch (Line 20-22, Fig. 2) of the *StoreLoad* protocol - sending a request to the *dealer* and handling the reply via method *put*. The correct order of messages is verified by the FSM attached to *c* and *c1*. As a result, errors such as calling *put* before *update* or executing two consecutive updates, will be detected as invalid.

3 Implementation of Multiparty Session Actors

AMQP in a Nutshell Distributed actor library. We have implemented the multiparty session actors on top of Celery [3] (distributed message queue in Python) with support for distributed actors. Celery uses advanced message queue protocol (AMQP 0-9-1 [2]) as a transport. The reason for choosing AMQP network as base for our framework is that AMQP middleware shares a similar abstraction with the actor programming model, which makes the implementation of distributed actors more natural. AMQP model can be summarised as follow: messages are published by producers to entities, called exchanges (or mailboxes). Exchanges then distribute message copies to queues using rules called bindings. Then AMQP brokers (virtual routers) deliver messages to consumers subscribed to queues. Distributed actors are naturally represented in this context using the abstractions of exchanges. Each actor type is represented in the network as an exchange and is realised as a consumer subscribed to a queue based on a pattern matching on the actor id. Message handlers are implemented as methods on the actor class.

Our distributed actor discovery mechanism draws on the AMPQ abstractions of exchanges, queues and binding, and our extensions to the actor programming model are built using Python advanced abstraction capabilities: two main capabilities are coroutines (for realising the actors inter-concurrency) and decorators (for annotating actor types and methods).

Actor roles. A key idea is each role to run in a virtual thread of an actor (using Python coroutines/green threads). We annotate methods, implementing part of a protocol, with a role decorator. Roles are scheduled cooperatively. This means that at most one role can be active in a session actor at a time. A role is activated when a message is received and ready to be processed. Switching between roles is done via the *become* primitive (as demonstrated in Fig. 3), which is realised as sending a message to the internal queue of the actor.

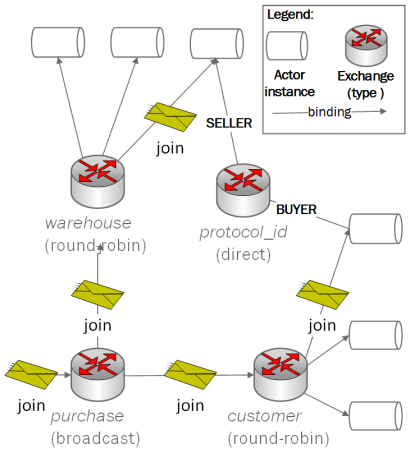


Figure 4: Organising Actors into protocols

Actors discovery. Fig. 4 presents the network setting (in terms of AMQP objects) for realising the actor discovery for *buyer* and *seller* of the protocol *Purchase*. For simplicity, we create the actor exchanges on starting of the actor system – round-robin exchange per actor type (*warehouse* and *customer* in Fig. 4) and broadcast exchange per protocol type (*purchase* in Fig. 4). All spawned actors alternate to receive messages addressed to their type exchange. Session actors are registered for roles via the protocol decorator and as a result their type exchange is bound to the protocol exchange (Line 1 in Fig. 3 binds *warehouse* to *purchase* in Fig. 4).

We explain the workflow for actor discovery. When a protocol is started, a fresh protocol id and an

exchange with that id are created. The type of the exchange is *direct*[1] (*protocol id* in Fig. 4). Then *join* message is sent to the protocol exchange and delivered to one actor per registered role (*join* is broadcasted to *warehouse* and *customer* in Fig. 4). On *join*, an actor binds itself to the *protocol id* exchange with subscription key equal to its role (bindings *seller* and *buyer* in Fig. 4). When an actor sends a message to another actor within the same session (for example *c.buyer.send* in Fig. 3), the message is sent to the protocol id exchange (stored in *c*) and from there delivered to the *buyer* actor.

Order preservation through FSM checking. Whenever a message is received the actor internal loop dispatches the message to the role the message is annotated with. The FSM, generated from the Scribble compiler (as shown in Fig. 1), is loaded when an actor joins a session and messages are passed to the FSMs for checking before being dispatched to their handler. The FSM perform checks message labels (already part of the actor payload) and sender and receiver roles (part of the message binding key due to our extension). An outline of the monitor implementation can be also found in [6]. The monitor mechanism is an incarnation of the session monitor in [6] within actors.

4 Related and Future Work

There are several theoretical works that have studied the behavioural types for verifying actors [7, 4]. The work [4] proposes a behavioural typing system for an actor calculus where a type describes a sequence of inputs and outputs performed by the actor body, while [7] studies session types for a core of Erlang. On the practical side, the work [10] proposes a framework of three layers for actor programming - actors, roles and coordinators, which resembles roles and protocol mailboxes in our setting. Their specifications focus on QoS requirements, while our aim is to describe and ensure correct patterns of interactions (message passing). Scala actor library [1] (studied in [13]) supports FSM verification mechanism (through inheritance) and typed channels. Their channel typing is simple so that it cannot capture structures of communications such as sequencing, branching or recursions. These structures ensured by session types are the key element for guaranteeing deadlock freedom between multiple actors. In addition, in [1], channels and FSMs are unrelated and cannot be intermixed; on the other hand, in our approach, we rely on external specifications based on the choreography (MPST) and the FSMs usage is internalised (i.e. FSMs are automatically generated from a global type), therefore it does not affect program structures. To our best knowledge, no other work is linking FSMs, actors and choreographies in a single framework.

As a future work, we plan to develop (1) extensions to other actor libraries to test the generality of our framework and (2) extensive evaluations of the performance overhead of session actors. The initial micro benchmark shows the overhead of the three main additions to our framework (FSM checking, actor type and method annotation) is negligible per message, see [12] (for example, the overhead of FSM checking is less than 2%). As actor languages and frameworks are getting more attractive, we believe that our framework would become useful for coordinating large-scale, distributed actor programs.

References

[1] Akka - scala actor library. http://akka.io/.
[2] Advanced Message Queuing Protocol homepage. `http://www.amqp.org/`.
[3] Celery. `http://http://www.celeryproject.org//`.
[4] S. Crafa. Behavioural types for actor systems. `arXiv:1206.1687`.
[5] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
[6] R. Hu, R. Neykova, N. Yoshida, and R. Demangeon. Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python. In *RV'13*, volume 8174 of *LNCS*, pages 130–148. Springer, 2013.

[1] A direct type means that messages with routing key R are delivered to actors linked to the exchange with binding R.

[7] D. Mostrous and V. T. Vasconcelos. Session typing for a featherweight erlang. In *COORDINATION*, volume 6721 of *LNCS*, pages 95–109. Springer, 2011.
[8] R. Neykova, N. Yoshida, and R. Hu. SPY: Local Verification of Global Protocols. In *RV'13*, volume 8174 of *LNCS*, pages 358–363. Springer, 2013.
[9] Ocean Observatories Initiative. `http://www.oceanobservatories.org/`.
[10] S. Ren, Y. Yu, N. Chen, K. Marth, P.-E. Poirot, and L. Shen. Actors, roles and coordinators - a coordination model for open distributed and embedded systems. In *COORDINATION*, volume 4038 of *LNCS*, pages 247–265. Springer, 2006.
[11] Scribble project home page. `http://www.scribble.org`.
[12] Online appendix for this paper. `http://www.doc.ic.ac.uk/~rn710/sactor`.
[13] S. Tasharofi, P. Dinges, and R. E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP*, volume 7920, pages 302–326. Springer, 2013.

Lightening Global Types

Tzu-chun Chen
Department of Informatica, University of Torino, Italy

Abstract

Global session types prevent participants from waiting for never coming messages. Some interactions take place just for the purpose of informing receivers that some message will never arrive or the session is terminated. By decomposing a big global type into several light global types, one can avoid such kind of redundant interactions. Lightening global types gives us cleaner global types, which keep all necessary communications. This work proposes a framework which allows to easily decompose global types into light global types, preserving the interaction sequences of the original ones but for redundant interactions.

1 Introduction. Since cooperating tasks and sharing resources through communications under network infrastructures (e.g. clouds, large-scale distributed systems, etc.) has become the norm and the services for communications are growing with increasing users, it is a need to give programmers an easy and powerful programming language for developing applications of interactions. For this aim, Scribble [11], a communication-based programming language, is introduced building on the theory of global types [12, 1]. However the tool itself cannot ensure an efficient communication programming. The scenario of a communication can be very complicated so it becomes a burden for programmers to correctly code interactions which satisfy protocols (described by global types). At runtime, the cost for keeping all resources ready for a long communication and for maintaining the safety of the whole system can increase a lot.

For example, assume a gift requester needs a *key* (with her identity) to get a wanted gift. In order to get the key, she needs to get a *guide*, which is a map for finding the key. It is like searching for treasures step by step, where a player needs not be always online in *one* session for completing the whole procedure. Instead, the communication protocol can be viewed as *separated but related* sessions which are linked (we use **call**s to switch from a session to another). For example, let a session $\mathcal{A}$ do the interactions for getting *guide*, and another session $\mathcal{B}$ do the interactions for getting *key* with *guide*. Both *guide* and *key* are knowledge gained from these interactions. *guide* bridges $\mathcal{A}$ and $\mathcal{B}$ as it is gained in $\mathcal{A}$ and used in $\mathcal{B}$. The participant then uses *key*, if she successfully got it in $\mathcal{B}$, to gain the wanted gift. Let session $\mathcal{C}$ implements these final interactions.

As standard we use global types [12, 1] to describe interaction protocols, adding a **call** command and type declarations for relating sessions. We call *light global types* the global types written in the extended syntax. Figure 1 simply represents the difference between viewing all interactions as one scenario and viewing interactions as three separated ones. As usual $\texttt{store} \to \texttt{req} : \{\mathit{yes3}(\mathsf{str}).\texttt{end}, \mathit{no3}().\texttt{end}\}$ models a communication where role `store` sends to role `req` either the label *yes3* and a value of type str or the label *no3*, and in both cases `end` finishes the interaction. The construct **call** ℓ_b indicates that the interaction should continue by executing the session described by the light global type associated to the name ℓ_b.

The set of light global types associated to the names ℓ_a, ℓ_b and ℓ_c describe the same protocol given by the global type G. An advantage of lightening is to spare communications needed only to warn participants they will not receive further messages. We call such communications *redundant* ones. In the example we avoid the interactions $\texttt{req} \to \texttt{store} : \mathit{no4}()$ and $\texttt{req} \to \texttt{issuer} : \mathit{no5}().\texttt{req} \to \texttt{store} : \mathit{no6}()$. Moreover, lightening prevents both local participants and global network from wasting resources, e.g. keeping online or waiting for step-by-step permissions.

$$
\begin{array}{lcl}
G & = & \mathtt{req} \to \mathtt{map} : \mathit{req1}(\mathsf{str}). \\
& & \mathtt{map} \to \mathtt{req} : \\
& & \{\ \mathit{yes1}(\mathsf{str}). \\
& & \quad \mathtt{req} \to \mathtt{issuer} : \mathit{req2}(\mathsf{str}). \\
& & \quad \mathtt{issuer} \to \mathtt{req} : \\
& & \quad \{\ \mathit{yes2}(\mathsf{int}). \\
& & \quad\quad \mathtt{req} \to \mathtt{store} : \mathit{req3}(\mathsf{int}). \\
& & \quad\quad \mathtt{store} \to \mathtt{req} : \\
& & \quad\quad \{\ \mathit{yes3}(\mathsf{str}).\mathtt{end}, \mathit{no3}().\mathtt{end}\ \}, \\
& & \quad\quad \mathit{no2}().\mathtt{req} \to \mathtt{store} : \mathit{no4}().\mathtt{end}\ \}, \\
& & \quad \mathit{no1}(). \\
& & \quad \mathtt{req} \to \mathtt{issuer} : \mathit{no5}(). \\
& & \quad \mathtt{req} \to \mathtt{store} : \mathit{no6}().\mathtt{end}\ \}
\end{array}
$$

$$
\begin{array}{lcl}
\ell_a & = & \mathtt{req} \to \mathtt{map} : \mathit{req1}(\mathsf{str}). \\
& & \mathtt{map} \to \mathtt{req} : \\
& & \{\ \mathit{yes1}(\mathsf{str}).\mathbf{call}\ \ell_b, \mathit{no1}().\mathtt{end}\ \} \\
\\
\ell_b & = & \mathtt{req} \to \mathtt{issuer} : \mathit{req2}(\mathsf{str}). \\
& & \mathtt{issuer} \to \mathtt{req} : \\
& & \{\ \mathit{yes2}(\mathsf{int}).\mathbf{call}\ \ell_c, \mathit{no2}().\mathtt{end}\ \} \\
\\
\ell_c & = & \mathtt{req} \to \mathtt{store} : \mathit{req3}(\mathsf{int}). \\
& & \mathtt{store} \to \mathtt{req} : \\
& & \{\ \mathit{yes3}(\mathsf{str}).\mathtt{end}, \mathit{no3}().\mathtt{end}\ \}
\end{array}
$$

Figure 1: Viewing interactions as a whole or as separated but related ones.

```
protocol getGift
         (role req, role map,
          role issuer, role store){
  req1(str identity) from req to map;
  choice at map{
    yes1(str guide) from map to req;
    req2(str guide) from req to issuer;
    choice at issuer{
      yes2(int key) from issuer to req;
      req3(int key) from req to store;
      choice at store{
        yes3(str gift) from store to req;
      } or {
        no3() from store to req;
      }
    } or {
      no2() from issuer to req;
      no4() from req to store;
    }
  } or {
    no1() from map to req;
    no5() from req to issuer;
    no6() from req to store;
  }
}
```

```
protocol getGuide (role req, role map){
  req1(str identity) from req to map;
  choice at map{
    yes1(str guide) from map to req;
    run protocol
          getKey(role req, role issuer) at req;
  } or {
    no1() from map to req;
  }
}
protocol getKey (role req, role issuer){
  req2(str guide) from req to issuer;
  choice at issuer{
    yes2(int key) from issuer to req;
    run protocol
          getGift'(role req, role store) at req;
  } or {
    no2() from issuer to req;
  }
}
protocol getGift' (role req, role store){
  req3(int key) from req to store;
  choice at store{
    yes3(str gift) from store to req;
  } or {
    no3() from store to req;
  }
}
```

Figure 2: Scribble for **protocol**s getGift (LHS) and for getGuide, getKey, and getGift′ (RHS).

Features of lightening become more clear by looking at the Scribble code implementing the global types of Figure 1, see Figure 2. The words in bold are keywords. In Scribble after the **protocol** keyword one writes the protocol's name and declares the roles involved in the session, then one describes the interactions in the body. The left-hand side (LHS) of Figure 2 shows the Scribble code corresponding to the global type G. Although the code is for a simple task, it is involved since there are three nested **choice**s. On the contrary, the right-hand side (RHS) of Figure 2 clearly illustrates the steps for getting a gift in three small protocols (corresponding to the light global types associating to ℓ_a, ℓ_b, ℓ_c respectively), which are separated but linked (by **run** and **at**) for preserving the causality.

The structure of the paper is the following. Section 2 introduces a framework of light global session types, which gives a syntax to easily compose a global type by light global types. Section 3 proposes a function for decomposing a general global type into light global types, e.g. it decomposes G into the types associated to ℓ_a, ℓ_b and ℓ_c. Section 4 proves the soundness of the function, and Section 5 discusses related and future works.

2 Syntax of (light) global types. The syntax for *global types* is standard:

$$G \quad ::= \quad \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).G_j\}_{j\in J} \mid \mu \mathsf{t}.G \mid \mathsf{t} \mid \mathtt{end} \quad \text{where } S \quad ::= \quad \mathsf{unit} \mid \mathsf{nat} \mid \mathsf{str} \mid \mathsf{bool}$$

The branching $\mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).G_j\}_{j\in J}$ says that role $\mathtt{r}_1$ sends a label l_j and a message of type S_j to $\mathtt{r}_2$ by selecting $j \in J$ and then the interaction continues as described in G_j. $\mu \mathsf{t}.G$ is a recursive type, where t is guarded in G in the standard way. `end` means termination of the protocol. We write $l()$ as short for $l(\mathsf{unit})$ and we omit brackets when there is only one branch.

Light global types are global types extended with declarations and the construct **call**:

$$D \quad ::= \quad \overrightarrow{\ell = L} \qquad\qquad L \quad ::= \quad \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L_j\}_{j\in J} \mid \mu \mathsf{t}.L \mid G \mid \mathbf{call}\ \ell$$

Declarations associate names to light global types, for example in Figure 1 the name ℓ_a is associated with the type

$$\mathtt{req} \to \mathtt{map} : \mathit{req1}(\mathsf{str}).\ \ \mathtt{map} \to \mathtt{req} : \{\ \mathit{yes1}(\mathsf{str}).\mathbf{call}\ \ell_b, \mathit{no1}().\mathtt{end}\ \}$$

We denote by $\emptyset$ the empty declaration.

The type **call** ℓ prescribes that the interaction continues by opening a new session with light global type L such that $\ell = L$ belongs to the set of current declarations. For example according to the declarations in Figure 1 **call** ℓ_b asks to open a new session with type:

$$\mathtt{req} \to \mathtt{issuer} : \mathit{req2}(\mathsf{str}).\ \ \mathtt{issuer} \to \mathtt{req} : \{\ \mathit{yes2}(\mathsf{int}).\mathbf{call}\ \ell_c, \mathit{no2}().\mathtt{end}\ \}$$

3 Lightening global types. This section describes a function for removing redundant interactions (Definition 1) from (possibly light) global types. It uses lightening, since it adds **call** constructors and declarations. In the next section we will show that the initial and final protocols describe the same non-redundant interactions.

We consider an interaction redundant when only one label can be sent and the message is not meaningful, i.e. it has type unit and it does not appear under a recursion. More precisely, by defining light global contexts (without recursion) as follows:

$$\mathtt{C} \quad ::= \quad [\,] \mid \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).G_j, l(S).\mathtt{C}\}_{j\in J}$$

we get:

Definition 1 (Redundant interaction)**.** The interaction $\mathtt{r}_1 \to \mathtt{r}_2 : l()$ is *redundant* in

$$\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 :\ l().L\,].$$

Interactions sending only one label and with messages of type unit are needed inside recursions when they terminate the cycle. For example the interaction $\mathtt{r}_2 \to \mathtt{r}_3 : \mathit{stop}()$ cannot be erased in the type

$$\mu \mathsf{t}.\mathtt{r}_1 \to \mathtt{r}_2 : \{\mathit{goon}(\mathsf{int}).\mathtt{r}_2 \to \mathtt{r}_3 : \mathit{goon}(\mathsf{int}).\mathsf{t}, \mathit{stop}().\mathtt{r}_2 \to \mathtt{r}_3 : \mathit{stop}().\mathtt{end}\}$$

We define $\mathbb{L}(L)$ as a function for removing redundant interactions in L by decomposing L into separated light global types. The basic idea is that, in order to erase redundant communications, the roles which are the receivers of these communications must get the communications belonging to other branches in separated types. Therefore the result of $\mathbb{L}(L)$ is a new light global type and a set of declarations with fresh names.

The function $\mathbb{L}(L)$ uses the auxiliary function $L \Downarrow \mathtt{r}$ which searches inside the branches of L the first communications with receiver $\mathtt{r}$ and replaces these communications (and all the following types) by **call**s to newly created names, which are associated by declarations to the corresponding types. Therefore also the result of $L \Downarrow \mathtt{r}$ is a new light global type and a set of declarations with fresh names.

Definition 2 (The function $\Downarrow$)**.** The function $L \Downarrow \mathtt{r}$ is defined by induction on L:

$$\mathtt{end} \Downarrow \mathtt{r} = (\mathtt{end}, \emptyset) \qquad \mathsf{t} \Downarrow \mathtt{r} = (\mathsf{t}, \emptyset) \qquad \mathbf{call}\ \ell \Downarrow \mathtt{r} = (\mathbf{call}\ \ell, \emptyset) \qquad \mu \mathsf{t}.L \Downarrow \mathtt{r} = (\mu \mathsf{t}.L, \emptyset)$$

$$(\mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L_j\}_{j\in J}) \Downarrow \mathtt{r} = \begin{cases} (L_j, \emptyset) & \text{if } \mathtt{r}_2 = \mathtt{r} \text{ and } S_j = \mathsf{unit} \text{ and } J = \{j\} \\ (\mathbf{call}\ \ell, \ell = \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L_j\}_{j\in J}) & \text{if } \mathtt{r}_2 = \mathtt{r} \text{ and } S_j \neq \mathsf{unit} \text{ or } J \neq \{j\} \\ \quad \text{where } \ell \text{ is a fresh name} & \\ (\mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L'_j\}_{j\in J}, \bigcup_{j\in J} D_j) & \text{otherwise} \\ \quad \text{where } L_j \Downarrow \mathtt{r} = (L'_j, D_j) \text{ for } j \in J. & \end{cases}$$

We remark that the first case of the definition of $\Downarrow$ for a branching type allows this function to eliminate a redundant interaction. Therefore one application of function $\mathbb{L}$ can get rid of more than one redundant interaction, as exemplified below.

The function $\mathbb{L}$ is defined by induction on the context in which the redundant interaction appears.

Definition 3 (The function $\mathbb{L}$)**.** The application of the function $\mathbb{L}$ to $\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 :\ l().L\ ]$ for eliminating the interaction $\mathtt{r}_1 \to \mathtt{r}_2 :\ l().L$ is defined by induction on $\mathtt{C}$:

$$\mathbb{L}(\mathtt{r}_1 \to \mathtt{r}_2 :\ l().L\ ) \quad = \quad (L, \emptyset)$$

$$\mathbb{L}(\mathtt{r}'_1 \to \mathtt{r}'_2 : \{l_j(S_j).L_j, l'(S).\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 :\ l().L\ ]\}_{j\in J}) =$$

$$\begin{cases} (\mathtt{r}'_1 \to \mathtt{r}'_2 : \{l_j(S_j).L_j, l'(S).\mathtt{C}[L]\}_{j\in J}, \emptyset) & \text{if } \mathtt{r}_2 = \mathtt{r}'_1 \text{ or } \mathtt{r}_2 = \mathtt{r}'_2, \\ (\mathtt{r}'_1 \to \mathtt{r}'_2 : \{l_j(S_j).L'_j, l'(S).L'\}_{j\in J}, \bigcup_{j\in J} D_j \cup D) & \text{if } \mathtt{r}_2 \neq \mathtt{r}'_1 \text{ and } \mathtt{r}_2 \neq \mathtt{r}'_2, \\ \quad \text{where } L_j \Downarrow \mathtt{r}_2 = (L'_j, D_j) \text{ for } j \in J & \\ \quad \text{and } \mathbb{L}(\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 :\ \ l().L\ ]) = (L', D) & \end{cases}$$

The branching case of the above definitions needs some comments. If the receiver $\mathtt{r}_2$ is also the sender or the receiver of the top branching, then she is aware of the choice of the label l' and

the redundant interaction can simply be erased. Otherwise $\mathtt{r}_2$ must receive a communication in all branches l_j and in this case these communications need to be replaced by **call**s to fresh names of light global types. For this reason we compute $L_j \Downarrow \mathtt{r}_2$ for all $j \in J$. Moreover we recursively call the mapping $\mathbb{L}$ on $\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : \ l().L\ ]$.

We now show two applications of the lightening function, first to the global type G of Figure 1 and then to the light global type obtained as a result of the first application. The application of $\mathbb{L}$ for eliminating the interaction $\mathtt{req} \to \mathtt{store} : no4()$ gives as a result $(L'_a, \ell_c = L_c)$, where L_c is the type associated to ℓ_c in Figure 1 and:

$$\begin{array}{rcl} L'_a & = & \mathtt{req} \to \mathtt{map} : req1(\mathsf{str}).\mathtt{map} \to \mathtt{req} : \\ & & \{\ yes1(\mathsf{str}).\mathtt{req} \to \mathtt{issuer} : req2(\mathsf{str}).\mathtt{issuer} \to \mathtt{req} : \\ & & \quad \{\ yes2(\mathsf{int}).\mathbf{call}\ \ell_c, no2().\mathtt{end}\}, \\ & & \quad no1().\mathtt{req} \to \mathtt{issuer} : no5().\mathtt{end}\} \end{array}$$

Notice that also the redundant interaction $\mathtt{req} \to \mathtt{store} : no6()$ is erased in L'_a. The same result can be obtained by applying $\mathbb{L}$ for eliminating the interaction $\mathtt{req} \to \mathtt{store} : no6()$. Now if we apply $\mathbb{L}$ to L'_a for eliminating $\mathtt{req} \to \mathtt{issuer} : no5()$ we get $(L_a, \ell_b = L_b)$, where L_a and L_b are the types associated to ℓ_a and ℓ_b in Figure 1. So we get all declarations shown in Figure 1.

4 Safety of the lightening function. In order to discuss the correctness of our lightening function, following [8] we view light global types with relative sets of declarations as denoting languages of interactions which can occur in multi-party sessions. The only difference is that recursive types do not reduce, the reasons being that our lightening function does not modify them. More formally the following definition gives a labelled transition system for light global types with respect to a fixed set of declarations. As usual τ means a silent move, and $\checkmark$ session termination.

Definition 4 (LTS)**.**

$$\frac{\ell = L \in D}{\mathbf{call}\ \ell \xrightarrow{\tau}_D L}\ [\textsc{call}] \qquad \mu\mathrm{t}.L \xrightarrow{\checkmark}_D\ [\textsc{rec}] \qquad \mathtt{r}_1 \to \mathtt{r}_2 : \ l().L \xrightarrow{\tau}_D L\ [\textsc{red}] \qquad \mathtt{end} \xrightarrow{\checkmark}_D\ [\textsc{end}]$$

$$\frac{J \neq \{j\} \text{ or } S_j \neq \mathsf{unit}}{\mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L_j\}_{j\in J} \xrightarrow{\mathtt{r}_1 \to \mathtt{r}_2 : l_j(S_j)}_D L_j}\ [\textsc{act}]$$

We convene that λ ranges over $\mathtt{r}_1 \to \mathtt{r}_2 : l(S)$ and $\checkmark$, and that σ ranges over sequences of λ. Using the standard notation:

$$\begin{array}{rcl} L \overset{\lambda}{\Longrightarrow}_D L' & \text{if} & L \xrightarrow{\tau}{}^*_D L_1 \xrightarrow{\lambda}_D L_2 \xrightarrow{\tau}{}^*_D L' \text{ for some } L_1, L_2 \\ L \overset{\lambda\cdot\sigma}{\Longrightarrow}_D L' & \text{if} & L \overset{\lambda}{\Longrightarrow}_D L_1 \overset{\sigma}{\Longrightarrow}_D L' \text{ for some } L_1 \\ L \overset{\sigma\cdot\checkmark}{\Longrightarrow}_D & \text{if} & L \overset{\sigma}{\Longrightarrow}_D L_1 \overset{\checkmark}{\Longrightarrow}_D \text{ for some } L_1 \end{array}$$

The language generated by L and relative to D is the set of sequences σ obtained by reducing L using D. We take this language as the meaning of L relative to D (notation $[\![L]\!]_D$).

Definition 5 (Semantics)**.** $[\![L]\!]_D = \{\sigma \mid L \overset{\sigma}{\Longrightarrow}_D L' \text{ for some } L' \text{ or } L \overset{\sigma}{\Longrightarrow}_D\}$.

Soundness of lightening then amounts to show that the function $\mathbb{L}$ preserves the meaning of light global types with respect to the relative sets of declarations. A first lemma shows soundness of the function $\Downarrow$.

Lemma 1. Let L be a light global type with declaration D. If $L \Downarrow \mathtt{r} = (L', D')$, then $[\![L]\!]_D = [\![L']\!]_{D' \cup D}$.

Proof. The proof is by induction on L and by cases on Definition 2. The only interesting case is $L = \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L_j\}_{j \in J}$.

1. If $L \Downarrow \mathtt{r} = (L_j, \emptyset)$, then $S_j = \mathsf{unit}$ and $J = \{j\}$. By rule [RED] we have $L \xrightarrow{\tau}_D L_j$, thus $[\![L]\!]_D = [\![L_j]\!]_D = [\![L']\!]_D$.

2. If $L \Downarrow \mathtt{r} = (\mathbf{call}\ \ell, \ell = \mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L_j\}_{j \in J})$, then by rule [CALL] we have $L' \xrightarrow{\tau}_{\{\ell = L\} \cup D} L$. Thus $[\![L]\!]_D = [\![L']\!]_{\{\ell = L\} \cup D}$.

3. If $L \Downarrow \mathtt{r} = (\mathtt{r}_1 \to \mathtt{r}_2 : \{l_j(S_j).L'_j \Downarrow \mathtt{r}\}_{j \in J}, \bigcup_{j \in J} D_j)$, then $L_j \Downarrow \mathtt{r} = (L'_j, D_j)$ for $j \in J$. By Definition 5 and rule [ACT] $[\![L]\!]_D = \bigcup_{j \in J} \{\ \mathtt{r}_1 \to \mathtt{r}_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L_j]\!]_D\ \}$ and $[\![L']\!]_{\bigcup_{j \in J} D_j \cup D} = \bigcup_{j \in J} \{\ \mathtt{r}_1 \to \mathtt{r}_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L'_j]\!]_{\bigcup_{j \in J} D_j \cup D}\ \}$. Since by induction $[\![L_j]\!]_D = [\![L'_j]\!]_{D_j \cup D}$, we conclude $[\![L]\!]_D = [\![L']\!]_{D' \cup D}$.

□

Theorem 1 (Soundness)**.** Let L be a light global type with set of declarations D. If $\mathbb{L}(L) = (L', D')$, then $[\![L]\!]_D = [\![L']\!]_{D' \cup D}$.

Proof. The proof is by induction on L and by cases on Definition 3.

1. If $L = \mathtt{r}_1 \to \mathtt{r}_2 : l().L''$, then $\mathbb{L}(L) = (L'', \emptyset)$. We conclude since by Definition 5 and by rule [RED] $[\![L]\!]_D = [\![L'']\!]_D$.

2. Let $L = \mathtt{C}_0[\mathtt{r}_1 \to \mathtt{r}_2 : l().L'']$ where $\mathtt{C}_0 = \mathtt{r}'_1 \to \mathtt{r}'_2 : \{l_j(S_j).L_j, l'(S).\mathtt{C}\}]\}_{j \in J}$. By Definition 5 and by rule [ACT]

$$\begin{array}{rcl} [\![L]\!]_D & = & \bigcup_{j \in J} \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L_j]\!]_D\ \} \cup \\ & & \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l'(S) \cdot \sigma \mid \sigma \in [\![\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : l().L'']]\!]_D\ \} \end{array}$$

   (a) If $\mathtt{r}_2 = \mathtt{r}'_1$ or $\mathtt{r}_2 = \mathtt{r}'_2$, then

$$\begin{array}{rcl} [\![L']\!]_D & = & \bigcup_{j \in J} \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L_j]\!]_D\ \} \cup \\ & & \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l'(S) \cdot \sigma \mid \sigma \in [\![\mathtt{C}[L'']]\!]_D\ \} \end{array}$$

   We conclude since rule [RED] $[\![\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : l().L'']]\!]_D = [\![\mathtt{C}[L'']]\!]_D$.

   (b) If $\mathtt{r}_2 \neq \mathtt{r}'_1$ and $\mathtt{r}_2 \neq \mathtt{r}'_2$, let us assume $L_j \Downarrow \mathtt{r}_2 = (L'_j, D_j)$ for $j \in J$ and $\mathbb{L}(\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : \ \ l().L\ ]) = (L'_0, D_0)$. Then

$$\begin{array}{rcl} [\![L']\!]_{\bigcup_{j \in J} D_j \cup D_0 \cup D} & = & \bigcup_{j \in J} \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l_j(S_j) \cdot \sigma \mid \sigma \in [\![L'_j]\!]_{\bigcup_{j \in J} D_j \cup D_0 \cup D}\ \} \cup \\ & & \{\ \mathtt{r}'_1 \to \mathtt{r}'_2 : l'(S) \cdot \sigma \mid \sigma \in [\![L'_0]\!]_{\bigcup_{j \in J} D_j \cup D_0 \cup D}\ \} \end{array}$$

   We conclude since by Lemma 1 $[\![L_j]\!]_D = [\![L'_j]\!]_{D_j \cup D}$ and by induction

$$[\![\mathtt{C}[\mathtt{r}_1 \to \mathtt{r}_2 : l().L'']]\!]_D = [\![L'_0]\!]_{D_0 \cup D}.$$

□

In the running example we get $[\![G]\!]_\emptyset = [\![L'_a]\!]_{\ell_c = L_c} = [\![L_a]\!]_{\ell_b = L_b\ \ell_c = L_c}$.

5 Related works and conclusion. In the recent literature global types have been enriched in various directions by making them more expressive through roles [10] or logical assertions [3] or monitoring [2], more safe through security levels for data and participants [6, 5] or reputation systems [4].

The more related papers are [9] and [7]. Demangeon and Honda introduce nesting of protocols, that is, the possibility to define a subprotocol independently of its parent protocol, which calls the subprotocol explicitly. Through a call, arguments can be passed, such as values, roles and other protocols, allowing higher-order description. Therefore global types in [9] are much more expressive than our light types. Carbone and Montesi [7] propose to merge together protocols interleaved in the same choreography into a single global type, removing costly invitations. Their approach is opposite to ours, and they deal with implementations, while we deal with types.

In this paper we show how to decompose interactions among multiple participants in order to remove redundant interactions, by preserving the meaning of (light) global types. We plan to implement our lightening function in order to experiment its practical utility in different scenarios.

Acknowledgements We are grateful to Mariangiola Dezani-Ciancaglini for her valuable comments and feedbacks. We also thank PLACES reviewers for careful reading, since we deeply revised this article following their suggestions.

REFERENCES.

[1] 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*, *LNCS* 5201, Springer, pp. 418–433.

[2] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda & Nobuko Yoshida (2013): *Monitoring Networks through Multiparty Session Types.* In: *FMOODS/FORTE*, *LNCS* 7892, Springer, pp. 50–65.

[3] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): *A Theory of Design-by-contract for Distributed Multiparty Interactions.* In: *CONCUR*, LNCS, Springer, pp. 162–176.

[4] Viviana Bono, Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2012): *A Reputation System for Multirole Sessions.* In: *TGC*, *LNCS* 7173, Springer, pp. 1–24.

[5] Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2011): *Information Flow Safety in Multiparty Sessions.* In: *EXPRESS*, *EPTCS* 64, pp. 16–31.

[6] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Tamara Rezk (2010): *Session Types for Access and Information Flow Control.* In: *CONCUR*, *LNCS* 6269, Springer, pp. 237–252.

[7] Marco Carbone & Fabrizio Montesi (2012): *Merging Multiparty Protocols in Multiparty Choreographies.* In: *PLACES*, *EPTCS* 109, pp. 21–27.

[8] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2012): *On Global Types and Multi-Party Sessions. Log. Meth. Comp. Scie.* 8, pp. 1–45.

[9] Romain Demangeon & Kohei Honda (2012): *Nested Protocols in Session Types.* In: *CONCUR*, *LNCS* 7454, Springer, pp. 272–286.

[10] Pierre-Malo Deniélou & Nobuko Yoshida (2011): *Dynamic Multirole Session Types.* In: *POPL*, ACM, pp. 435–446.

[11] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen & Nobuko Yoshida (2011): *Scribbling Interactions with a Formal Foundation.* In: *ICDCIT*, *LNCS* 6536, pp. 55–75.

[12] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types.* In: *POPL*, ACM, pp. 273–284.

Verifying Parallel Loops with Separation Logic*

Stefan Blom, Saeed Darabi and Marieke Huisman

University of Twente, Netherlands
s.c.c.blom,s.darabi,m.huisman@utwente.nl

Abstract

This paper proposes a technique to specify and verify whether a loop can be parallelised. Our approach can be used as an additional step in a parallelising compiler to verify user annotations about loop dependences. Essentially, our technique requires each loop iteration to be specified with the locations it will read and write. From the loop iteration specifications, the loop (in)dependences can be derived. Moreover, the loop iteration specifications also reveal where synchronisation is needed in the parallelised program. The loop iteration specifications can be verified using permission-based separation logic.

1 Introduction

Parallelising compilers can detect loops that can be executed in parallel. However, this detection is not perfect. Therefore developers can typically also use a pragma to declare that a loop is parallel. Any loop annotated with such a pragma will be assumed to be parallel by the compiler.

This paper addresses the problem of how to verify that loops that are declared parallel by a developer can indeed safely be parallelised. The solution is to add specifications to the program that when verified guarantee that the program can be parallelised without changing its meaning. Our specifications stem from permission-based separation logic [4, 5], an extension of Hoare logic. This has the advantage that we can easily combine the specifications related to parallelisation with functional correctness properties.

We illustrate our approach on the PENCIL programming language [1]. This is a high-level programming language to simplify using many-core processors, such as GPUs, to accelerate computations. It is currently under development as a part of the CARP project[1]. However, our approach also applies to other languages that use the concept of parallel loops, such as OpenMP [6]. In order to simplify the presentation in this paper, we limit ourselves to single loops. At the end of this paper, we will briefly discuss how to extend our approach to nested loops.

Below, we first present some background information, and then we introduce the specification language for parallel loops. Next, we sketch how we can implement automated verification of the specifications. Finally, we conclude with future work.

2 Background

Parallel Hardware. Modern hardware offers many different ways of parallelising code. Most main processors nowadays are multi-core. Additionally, they often have a set of vector instructions that can operate on small vectors instead of just a single value at once. Moreover, graphics processing units (GPUs) nowadays also can be used for general-purpose programming. Writing and tuning software for such accelerated hardware can be a very time-consuming task.

*This work is supported by the EU FP7 STREP project CARP (project nr. 287767).

[1] See http://www.carpproject.eu/.

The PENCIL Language. The PENCIL programming language is developed as a part of the CARP project. It is designed to be a high-level programming language for accelerator programming, providing support for efficient compilation. Its core is a subset of sequential C, imposing strong limitations on pointer-arithmetic. In addition to traditional C, it allows loops to be specified with two pragmas: *independent* and *ivdep*, indicating that a loop can be parallelised, because it is independent, or only contains forward dependences, respectively.

Loop Dependences. Several kinds of loop dependences can be identified. There exists a *loop-carried dependence* from statement S_{src} to statement S_{sink} in the body of a loop if there exist two iterations i and j of that loop, such that:

- Iteration i is before iteration j, *i.e.*, $i < j$.
- Statements S_{src} on iteration i and S_{sink} on iteration j access the same memory location.
- At least one of these accesses is a write.

When S_{src} syntactically appears before S_{sink} (or if they are the same) there is a *forward loop-carried dependence*, otherwise there is a *backward loop-carried dependence*. The distance between two dependent iterations i and j is defined as the *distance of dependence*.

On the right, we show examples of first a forward and then a backward loop carried dependence. In both cases there is a dependence between S_1 and S_2. In the first loop, the read in S_2 reads the value written in S_1 in the previous iteration of the loop. In the second loop, the read in S_2 must be done before the value is overwritten in S_1 during the next iteration.

```
for(int i=1;i<=N;i++){
    S1: a[i] = c[i] + 1;
    S2: c[i] = a[i-1] + 2;
}

for(int i=0;i<N;i++){
    S1: a[i] = c[i] + 1;
    S2: c[i] = a[i+1] + 2;
}
```

The distinction between forward and backward dependences is important. Independent parallel execution of a loop with dependences is always unsafe, because it may change the result. However, a loop with forward dependences can be parallelised by inserting an appropriate synchronisation in the code, while loops with backward dependences cannot be parallelised.

Separation Logic. Our approach to reason about loop (in)dependences uses permission-based separation logic to specify which variables are read and written by a loop iteration. Separation logic [9] was originally developed as an extension of Hoare logic to reason about pointer programs, as it allows to reason explicitly about the heap. This makes it also suited to reason modularly about concurrent programs [8]: two threads that operate on disjoint parts of the heap do not interfere, and thus can be verified in isolation. The basis of our work is a separation logic for C [10], but extended with permissions [5], to denote either the right to read from or to write to a location. The set of permissions that a thread holds are often known as its *resources*. We write access permissions as $\mathbf{perm}(e, \pi)$, where e is an expression denoting a memory location and $\pi \in (0, 1]$ is a fraction, where any value permits reading and 1 provides write permission. The logic prevents the sum of permissions for a location over all threads to exceed 1, which prevents data races. In earlier work, we have shown that this logic is suitable to reason about kernel programs [3].

```
for(i=0;i<N;i+=1)
/*@ requires perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2);
    ensures  perm(a[i],1) ** perm(c[i],1) ** perm(b[i],1/2); @*/
  { S1: a[i] = b[i] + 1;
    S2: c[i] = a[i] + 2; }
```

Listing 1: Specification of an Independent Loop

3 A Specification Language for Loop Dependence

The classical way to specify the effect of a loop is by means of an invariant that has to hold before and after the execution of each iteration in the loop. Unfortunately, this offers no insight into possible parallel execution of the loop. Instead we will consider every iteration of the loop in isolation. To be able to handle dependences, we specify restrictions on how the execution of the statements for each iteration is scheduled. In particular, each iteration is specified by its own contract, *i.e.*, its *iteration contract*. In the iteration contract, the precondition specifies resources that a particular iteration requires and the postcondition specifies the resources which are released after the execution of the iteration. In other words, we treat each iteration as a specified block [7].

Listing 1 gives an example of an *independent loop*, specified by its iteration contract. The contract requires that at the start of iteration i, permission to write both c[i] and a[i] is available, as well as permission to read b[i]. The contract also ensures that these permissions are returned at the end of iteration i. The iteration contract implicitly requires that the separating conjunction of all iteration preconditions holds before the first iteration of the loop, and that the separating conjunction of all iteration postconditions holds after the last iteration of the loop. In Listing 1, the loop iterates from 0 to $N-1$, so the contract implies that before the loop, permission to write the first N elements of both a and c must be available, as well as permission to read the first N elements of b. The same permissions are ensured to be available after termination of the the loop.

To specify *dependent loops*, in addition we need the ability to specify what happens when the computations have to synchronise due to a dependence. During such a synchronisation, permissions should be transferred from the iteration containing the source of a dependence to the iteration containing the sink of that dependence. To specify a *permission transfer* we introduce the **send** keyword:

```
//@ send φ to L, d;
```

This specifies that the permissions and properties expressed by the separation logic formula ϕ are transferred to the statement labelled L in the iteration $i+d$, where i is the current iteration and d is the distance of dependence.

Below, we will give two examples that illustrate how loops are specified with **send** clauses. The **send** clause alone completely specifies both how permissions are provided and used by the iterations. However, for readability, we also mark the place where the permission are used with a corresponding **receive** statement as a comment. Listing 2 gives a specified program with a forward dependence, similar to our earlier example, while Listing 3 gives an example of a program with a backward dependence.

We discuss the annotations of the first program in some detail. Each iteration i starts with write permission on a[i] and c[i]. The first statement is a write to a[i], which needs write permission. The second statement reads a[i−1], which is not allowed unless read permission is

```
for(int i=1;i<=N;i++)
/*@ requires i==1 ==> perm(a[i-1],1/2);
    requires perm(c[i],1) ** perm(a[i],1);
    ensures  perm(c[i],1) ** perm(a[i],1/2) ** perm(a[i-1],1/2);
    ensures  i==N ==> perm(a[i],1/2); @*/
{
    S1: a[i] = c[i]*CONST +a[i]*(1-CONST);
    //@ send perm(a[i],1/2) to S2,1;
    // if (i>1) receive perm(a[i-1],1/2);
    S2: c[i] = min(a[i],a[i-1]);
}
```

Listing 2: Specification of a Forward Loop-Carried Dependence

```
for( i=0;i<N;i++)
/*@ requires i==0 ==> perm(a[i],1/2);
    requires perm(c[i],1) ** perm(a[i],1/2) ** perm(a[i+1],1/2);
    ensures  perm(c[i],1) ** perm(a[i],1);
    ensures  i==N-1 ==> perm(a[i+1],1/2); @*/
{
   // if (i>0) receive perm(a[i],1/2);
   S1: a[i] = c[i]*CONST + a[i]*(1-CONST);
   S2: c[i] = min(a[i+1],a[i]);
   /*@ send perm(a[i+1],1/2) to S1, 1; *@/
}
```

Listing 3: Specification of a Backward Loop-Carried Dependence

available. For the first iteration, this read permission is available. For all subsequent iterations, permission must be transferred. Hence a **send** annotation is specified after the first assignment that transfers a read permission on a[i] to the next iteration (and in addition, keeps a read permission itself). The postcondition of the iteration contract reflects this: it ensures that the original permission on c[i] is released, as well as the read permission on a[i], which was not sent, and also the read permission on a[i−1], which was received. Finally, since the last iteration cannot transfer a read permission on a[i], the iteration contract's postcondition also specifies that the last iteration returns this non-transferred read permission on a[i].

The specifications in both listings are valid. Hence every execution order of the loop bodies that respects the order implied by the **send** annotations yields the same result as sequential execution. In the case of the forward dependence example, this can be achieved by adding appropriate synchronisation in the parallelised code. All parallel iterations should synchronise each **send** annotation with the location of the specified label to ensure proper permission transfer. For the backward dependence example, only sequential execution respects the ordering.

4 Verifying Dependence Annotations

To verify an iteration contract, we encode it as a standard method contract that can be verified using the VerCors tool set [2]. Suppose we have a loop specified with an iteration contract as below:

```
S_pre;
for(int i=0;i<N;i++)
/*@ requires pre(i);
    ensures  post(i); @*/
{   S; }
S_post;
```

To prove that this program respects its annotations, the following proof obligations have to be discharged:

- after S_{pre}, the separating conjunction of all of the iteration preconditions holds;
- the loop body S respects the iteration contract; and
- the statement S_{post} can be proven correct, assuming that the separating conjunction of the postconditions holds.

To generate these proof obligations, we encode the original program by generating several annotated procedures by the following steps:

1. We replace every loop in the program with a call to a procedure loop_main, whose arguments are the free variables occurring in the loop. The contract of this procedure requires the separating conjunction of all preconditions and ensures the separating conjunction of all postconditions. After this replacement, we can verify the program with existing tools to discharge the first and the last proof obligations.
2. To discharge the remaining proof obligation, we generate a procedure loop_body, whose arguments are the loop variable i plus the same arguments as loop_main. The contract of this procedure is the iteration contract of the loop body, preceded by a requirement that states that the value of the iteration variable is within the bounds of the loop.

The result of this encoding is as follows:

```
void block(){
    S_pre;
    loop_main(N, free(S));
    S_post;
}

/*@ requires (\forall* int i;0<=i && i<N; pre(i));
    ensures  (\forall* int i;0<=i && i<N; post(i)); @*/
loop_main(int N, free(S));

/*@ requires (0<=i && i<N) ** pre(i);
    ensures  post(i); @*/
loop_body(int i, int N, free(S)){ S; }
```

Verification of the **send** instruction is done by replacing the **send** annotation with a procedure call send_phi(i); and by inserting a procedure call recv_phi(i); at the location of the label L. The contracts of these methods encode the transfer of the resources specified by $\phi(i)$ from the sending iteration to the receiving iteration, subject to two conditions:

1. Permissions can only be transferred to future iterations ($d > 0$).
2. Transfer only happens if both the sending and the receiving iterations exist.

The existence of iteration i is expressed by the predicate is_iteration(i), whose definition is derived from the loop bounds. For example, the loop **for**(**int** i=0;i<N;i++) gives rise to

```
boolean is_iteration(int i){return 0 <= i && i < N;}
```

Using this notation the generated (abstract) methods and contracts are:

```
/*@ requires is_iteration(i+d) ==> φ(i);
 @*/
void send_phi(int i);

/*@ ensures  is_iteration(i-d) ==> φ(i-d);
@*/
void recv_phi(int i);
```

Note that instead of a constant d, we may use any invertible function $d(i)$.

5 Conclusion and Future Work

This paper sketches how to verify parallel loops, even in the presence of dependences from one loop iteration to the next. The idea is to specify each iteration of a loop with its own iteration contract and to use the **send** annotation to transfer permission between iteration if needed. We conjecture that if verification of a loop is possible without using **send** then it is correct to tag the loop as independent, *i.e.*, an iteration never reads a location that was written by a different iteration. Moreover, if **send** is used with labels occurring after the statement then it is correct to use PENCIL's **ivdep** tag to indicate parallelisability.

The method described is modular in the sense that it allows us to treat any parallel loop as a statement, thus nested loops can be dealt with simply by giving them their own iteration contract. Alternatively one iteration contract can be used for several nested loops.

It is future work to provide a formal proof for our conjecture, as well as to develop fully automated tool support for discharging the proof obligations. We also plan to link our PENCIL specifications with our kernel logic [3] and to define compilation of PENCIL specifications.

Another possible direction for future work is to extend our approach to reason about the correctness of OpenMP [6] pragmas in parallel C programs. From the point of view of verification, many concepts in OpenMP and PENCIL are the same. For example, the **simd** pragma in OpenMP is used in the same way as PENCIL uses **ivdep**. In general, our method can be applied for verification of any high-level parallel programming language which uses compiler directives for parallelisation.

Finally, we will also investigate how the iteration contracts for the verifier and parallelisation pragmas for the compiler can support each other. We believe this support can work in both ways. First of all, the parallelising compiler can use verified annotations to know about dependences without analysing the code itself. In particular, the PENCIL language has a feature, called *function summaries*, that allows the programmer to tell the compiler which memory locations are written and/or read by a function by writing a fake function that assigns to the writable locations and reads form the readable locations. Such summaries are easily extracted from specifications, and thus in this way specifications can help to produce better code. Conversely, if the compiler performs an analysis then it could emit its findings as a specification template for the code, from which a complete specification can be derived.

References

[1] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. *CoRR*, abs/1302.5586, 2013.

[2] S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. to appear in FM2014.

[3] S. Blom, M. Huisman, and M. Mihelčić. Specification and verification of GPGPU programs, 2013. Accepted to appear in SCP.

[4] R. Bornat, C. Calcagno, P.W. O'Hearn, and M.J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[5] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[6] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[7] E.C.R. Hehner. Specified blocks. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005.

[8] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[9] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[10] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 97–108. ACM, 2007.

Towards Composable Concurrency Abstractions*

Janwillem Swalens, Stefan Marr, Joeri De Koster and Tom Van Cutsem

Software Languages Lab, Vrije Universiteit Brussel, Belgium
{jswalens,smarr,jdekoste,tvcutsem}@vub.ac.be

Abstract

In the past decades, many different programming models for managing concurrency in applications have been proposed, such as the actor model, Communicating Sequential Processes, and Software Transactional Memory. The ubiquity of multi-core processors has made harnessing concurrency even more important. We observe that modern languages, such as Scala, Clojure, or F#, provide not one, but *multiple* concurrency models that help developers manage concurrency. Large end-user applications are rarely built using just a single concurrency model. Programmers need to manage a responsive UI, deal with file or network I/O, asynchronous workflows, and shared resources. Different concurrency models facilitate different requirements. This raises the issue of how these concurrency models interact, and whether they are *composable*. After all, combining different concurrency models may lead to subtle bugs or inconsistencies.

In this paper, we perform an in-depth study of the concurrency abstractions provided by the Clojure language. We study all pairwise combinations of the abstractions, noting which ones compose without issues, and which do not. We make an attempt to abstract from the specifics of Clojure, identifying the general properties of concurrency models that facilitate or hinder composition.

1 Introduction

A typical interactive computer program with a graphical user interface needs *concurrency*: a number of activities need to be executed simultaneously. For example, a web browser fetches many files over the network, renders multiple documents in separate tabs, runs plug-ins in the background, and needs to keep the user interface responsive. Since the last decade multi-core processors have become ubiquitous: servers, laptops, and smartphones contain multi-core processors. This has made it possible to perform these concurrent activities simultaneously.

To express the interactions between these activities, a number of concurrency models have been developed. For example, the Actor Model [1] introduces actors to represent components of a system that communicate using asynchronous messages, while Software Transactional Memory (STM) [7, 4] coordinates the access of concurrent activities to shared memory using transactions.

Because of the varying and extensive requirements of interactive end-user programs, developers choose to combine different models [8]. For example, a web browser might use actors to run separate tabs in parallel, and manipulate the DOM (Document Object Model) of a web page using STM. We also see that many programming languages support a number of these models, either through built-in language support or using libraries. For example, the Akka library[1] for Java and Scala provides STM, futures, actors, and agents. Similarly, Clojure[2] has built-in support for atoms, STM, futures, and agents.

*An appendix with a complete discussion of results is available at http://soft.vub.ac.be/~jswalens/PLACES2014.pdf

[1] http://akka.io/

[2] http://clojure.org/

However, the concurrency models are not necessarily well-integrated: programmers may experience subtle problems and inconsistencies when multiple concurrency models interact. In this paper, we analyze some problems that arise when combining different concurrency models and outline potential directions for *composable concurrency abstractions*.

2 Concurrency Models

In this paper, we study the combination of atomics, STM, futures and promises, as well as approaches for *communicating threads* (actors and CSP). To illustrate the usefulness of combining these approaches, we consider how they could be used in a modern email application.

Atomics Atomics are variables that support a number of low-level atomic operations, e. g., compare-and-swap. Compare-and-swap compares the value of an atomic variable with a given value and, only if they are the same, replaces it with a new value. This is a single atomic operation. Operations affecting multiple atomic variables are not coordinated, consequently when modifying two atomic variables race conditions can occur. Atomics are typically used to share independent data fragments that do not require coordinated updates. For example, a mail client might use an atomic variable to represent the number of unread mails.

In Clojure, `atom`s are atomic references. Their value can be read using `deref`. Atoms are modified using `swap!`, which takes a function that is evaluated with the current value of the atom, and replaces it with the result only if it has not changed concurrently, using compare-and-swap to compare to the original value. If it did change, the `swap!` is automatically retried.

Software Transactional Memory (STM) STM [7, 4] is a concurrency model that allows many concurrent tasks to write to shared memory. Each task accesses the shared memory within a transaction, to manage conflicting operations. If a conflict is detected, a transaction can be retried. A mail client can use STM to keep information about mails consistent while updates from the server, different devices, and the user are processed at the same time.

Clojure's STM provides `ref`s, which can only be modified within a transaction. They are read using `deref` and modified using `ref-set`. Transactions are represented by a `dosync` block. Outside a `dosync` block, refs can only be read.

Futures and Promises Futures and promises[3] are placeholders for values that are only known later. A future executes a given function in a new thread. Upon completion, the future is resolved to the function's result. Similarly, a promise is a value placeholder, but it can be created independently and its value is 'delivered' later by an explicit operation. Reading a future or a promise blocks the reading thread until the value is available. A mail client can use futures to render a preview of an attachment in a background thread, while the mail body is shown immediately. When the future completes, the preview can be added to the view.

In Clojure, futures are created using `future`. Promises are created using `promise` and resolved using `deliver`. Futures and promises are read using `deref`, which potentially blocks.

Communicating Threads We classify concurrency models that use structured communication in terms of messages, instead of relying on shared memory, as *communicating threads*. Often, each thread has only private memory, ensuring that all communication is done via messages. This, combined with having each thread process at most one message at a time, avoids race conditions. However, models and implementations vary the concrete properties to account for a wide range of trade-offs. We distinguish between models that use asynchronous messages, such as actors [1] or agents (as in Clojure and Akka), and models that communicate

[3] Literature uses various different definitions of futures and promises. We use those of amongst others Clojure and Scala here.

	atoms	agents	STM	futures	promises	core.async
create	atom	agent	ref	future	promise	chan
read	deref	deref	deref	deref ⊗	deref ⊗	<! ⊗
write	reset!		ref-set		deliver	>! ⊗
	swap! ↻	send	alter			
block			dosync ↻			go
other		await ⊗				<!! >!! ⊗
						take! put!

Figure 1: Operations supported by Clojure's concurrency models. ⊗ indicates a (potentially) blocking operation, ↻ an operation that might be re-executed automatically.

using synchronous messages, such as CSP [5]. In a mail client, typical use cases include the event loop of the user interface as well as the communication with external systems with strict communication protocols, such as mail servers.

Clojure `agent`s represent state that can be changed via asynchronous messages. `send` sends a message that contains a function which takes the current state of the agent and returns a new state. The current state of an agent can be read using a non-blocking `deref`. The `await` function can be used to block until all messages sent to the agent so far have been processed. Clojure's core.async library implements the CSP model. A new thread is started using a `go` block, channels are created using `chan`. Inside a `go` block, values are put on a channel using `>!` and taken using `<!`. Outside `go` blocks, `>!!` and `<!!` can be used. These operations block until their complementary operation is executed on another thread.

All operations described above for Clojure are summarized in figure 1, highlighting blocking and re-execution.

3 Integration Problems of Concurrency Models

3.1 Criteria for composability

We study pairwise combinations of the five concurrency models described in the previous section. Two correctness properties are evaluated: safety and liveness [6]. For each combination we study whether additional safety or liveness issues can arise, emerging from the interactions between the two models. We consider two models *composable* if combining them cannot produce new safety or liveness issues.

Safety Safety means that, given a correct input, a program will not produce an incorrect result. In our context, many concurrency models are designed to avoid race conditions to achieve safety. They do this by managing shared resources: STM only allows shared memory to be accessed through transactions, while CSP or the actor model only allow threads to share data through explicit message passing.

When combining two models, new races could be introduced unexpectedly. For example, some implementations of STM have been proven linearizable [7]: every concurrent execution is equivalent to a legal sequential execution. However, this assumes that all shared resources are managed by the STM system, which is not true if a thread communicates with other threads, e. g., using CSP. This can cause unexpected interleavings that eventually lead to race conditions. This is not dissimilar to the problem of feature interaction [2], where several features that each function correctly separately, might behave incorrectly when combined.

Concretely, we study whether the combination of two concurrency models can introduce race conditions: incorrect results caused by unexpected interleavings.

	Safety					Liveness				
in \ used	atoms	agents	refs	futures promises	channels	atoms	agents	refs	futures promises	channels
atoms	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗
agents	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗
refs	✗	✓	✓	✗	✗	✓	✓	✓	✓	✗
futures promises	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗
channels	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗

Figure 2: This table shows when safety and liveness issues can arise by combining two models in Clojure. The model in the column is used in the model in the row.

Liveness Liveness guarantees a program will terminate if its input is correct. In our context, two problems can occur: deadlocks and livelocks. Deadlocks are introduced by operations that block, waiting until a certain condition is satisfied, but the condition is continually not satisfied. Livelocks appear when code is re-executed under a certain condition, and this condition is continually satisfied. Some concurrency models have proven liveness properties, for instance, STMs are usually non-blocking [7]. Others try to confine the problem by limiting blocking to a small set of operations. For example, futures only provide one blocking operation, reading, which waits until the future is resolved. As long as the future eventually resolves, no deadlocks will happen.

Again, when concurrency models are combined, unexpected deadlocks and livelocks might arise. An STM transaction that uses blocking operations of another model, such as CSP, is not guaranteed to be non-blocking anymore. Or, a future that contains blocking operations from another model might not ever be resolved, in other words, combining futures with another model can introduce unexpected deadlocks.

We study whether the combination of two concurrency models can introduce new deadlocks, by studying the blocking operations offered by a model, and/or new livelocks, by studying the operations that can cause re-execution.

3.2 Integration Problems in Clojure

We examine these combination issues specifically for Clojure. Each of the five concurrency models from section 2 is embedded in each of the models (including itself). The complete set of results is shown in the table in figure 2. For example, we embed a `send` to an agent (1) in an atom's `swap!` block (figure 3a), (2) in another agent action, (3) in an STM transaction (figure 3b), (4) in a future, and (5) in a `go` block (these results form the second column of figure 2). Even though some of these individual results are already known, we systematically study each pairwise combination of models, in an attempt to provide a comprehensive overview of safety and liveness issues. A discussion of the most interesting results is given below, a complete discussion of all results in the table is given in the online appendix.

Safety We first look at the possibility of race conditions (left side of table 2). Race conditions are caused by an incorrect interleaving between two models.

When using any concurrency model in the function given to an atom's `swap!` operation (first row of the table), race conditions are possible, because the function might be re-executed if the atom changed concurrently. For example, when this function sends a message to an agent, it could be sent twice (figure 3a). Moreover, because operations on multiple atoms are not coordinated, their updates are inherently racy.

```
(def notifications (agent '()))
(def unread-mails (atom 0))

(swap! unread-mails
  (fn [n]
    (send notifications
      (fn [msgs] (cons "New mail!" msgs)))
    (inc n)))
```

(a) Sending a message to an agent, in `swap!`. Send may happen more than once.

```
(def notifications (agent '()))
(def mail (ref {:subject "Hi" :archived false}))

(dosync
  (ref-set mail (assoc @mail :archived true))
  (send notifications
    (fn [msgs] (cons (str "Archived mail " (:subject
        @mail)) msgs))))
```

(b) Sending a message to an agent, in a `dosync`. Send is delayed until the transaction is committed.

Figure 3: Sending a message to an agent, in a block that might re-execute (`swap!` and `dosync`).

For STM, actions inside a `dosync` block are re-executed if the transaction is retried, and therefore no irrevocable operations should happen inside `dosync`. However, there are two safe combinations. Firstly, when a message is sent to an agent inside a `dosync` block (figure 3b), Clojure does not send this message immediately. Instead, it delays the send until the transaction is successfully committed. Secondly, embedding one `dosync` block in another means the inner transaction will be merged with the outer one, and as a result transactions are combined safely.

Based on these results we conclude that *if the 'outer' model might re-execute code it is given, and the 'inner' model might perform irrevocable actions, unexpected interleavings can happen and therefore safety is not guaranteed.*

Liveness Next, we look at the liveness property (right side of table 2): deadlocks and livelocks.

Deadlocks are introduced by blocking operations (indicated in figure 1). CSP relies heavily on blocking for communication, and as such deadlocks are possible when it is embedded in another model (see last column). This is particularly problematic when embedded in `swap!` (atoms) or `dosync` (STM): synchronous communication is irrevocable and should not be re-executed.

Reading a future or a promise blocks until its value is available. This can cause a deadlock when a promise is embedded in an agent (fourth column), because one thread might send an action to an agent, which reads a promise that is delivered in a later action sent to that same agent, possibly from another thread. Reading futures inside another future can also cause a deadlock when mutually recursive futures are allowed, as is the case in Clojure.

Finally, the agents' blocking `await` operation can cause deadlocks in a `go` block or a future (second column). In agent actions and STM transactions this situation is prevented by raising an exception. We conclude that *if the 'inner' model might block, and the 'outer' model does not expect this, a deadlock is possible.*

Livelocks appear when code is re-executed (operations that can cause re-execution are indicated in figure 1). An STM transaction is retried when it conflicts with another one, causing a livelock if the conflict would consistently occur. However, Clojure's STM prevents such deadlocks and livelocks dynamically [3]. In general, *a livelock can appear when a model that re-executes code is combined with a model that causes this re-execution to continually happen.* However, this occurs in none of the examined cases in Clojure.

4 Solutions and Open Questions

In the discussion of the previous section, we already pointed out some places where bad interactions between models are avoided by Clojure. Specifically: (1) Sending a message to an agent in an STM transaction is delayed until the transaction has been committed. (2) `await` is not

allowed in STM transactions, nor in actions sent to agents. (3) A `dosync` block embedded in another will not start a new transaction: the inner transaction is merged into the outer one. These mechanisms could be replicated in some other cases: similar to sending a message to an agent in a transaction, delivering a promise could also be delayed until the transaction has been committed. The combination of futures and transactions could further be improved by canceling futures started in a transaction if the transaction is aborted.

A solution to deadlocks caused by the combination of agents and futures/promises is to disallow reading a future/promise in an agent action, an operation that potentially blocks the agent. Instead, the future/promise would need to be read before sending the message.

To remove the unsafe combinations of atoms with any other model, some of the mechanisms of combinations with STM could be replicated (e. g., delaying sends to agents). However, this can be considered contrary to the purpose of atoms: they are low-level and uncoordinated, and accordingly offer no safety guarantees. Similarly, the CSP model uses synchronous, blocking operations by design, and therefore liveness issues are inherent to this model. Trying to avoid these would be contradictory to the nature of the model.

In general, in future research we would like to decompose several concurrency models into their components, or "building blocks". For example, we observe that some common elements are: (1) most models supply a way to create new threads (e. g., `agent`, `future`, `go`); (2) agents, actors, futures, promises and CSP provide message passing (asynchronous or synchronous); (3) agents, actors, and CSP have private memory per thread, while (4) atomics and STM provide a way to manage shared memory.

We want to extract such common elements and provide a way to compose them safely and efficiently. For example, different threads could have some private memory, communicate using message passing (as in the actor model and CSP), but also share a section of memory (e. g., using STM). Using these composable concurrency abstractions, it should be possible to express existing concurrency models as well as combinations of them correctly. In the end, it should be possible to write complex applications, such as the email client example of section 2, using a combination of concurrency models, without introducing new safety or liveness issues caused by interactions between the models.

5 Conclusion and Future Work

There exist various different concurrency models, and in many large-scale applications these are combined. However, subtle problems and inconsistencies can appear in the interactions between these models. In this paper, we studied the safety and liveness issues that can appear when the various concurrency models available for Clojure are combined.

We identified four reasons for conflicts between models. Firstly, when a model re-executes code, and this code uses another concurrency model that performs irrevocable actions, safety is not guaranteed. Clojure takes some special precautions in some of these cases. Secondly, when a model re-executes code, and this code can cause the re-execution to continually happen, a livelock is possible. In Clojure, this is prevented in the studied cases. Thirdly, when a model that supports blocking operations is embedded in a model that does not expect this, deadlocks become possible. Again, Clojure prevents this in some cases but not in others. Lastly, some models do not provide safety or liveness guarantees by design.

In future work, we aim to work towards composable concurrency abstractions: we will decompose existing concurrency models into more primitive "building blocks", and provide a way to compose these safely and efficiently.

References

[1] Gul A. Agha. *Actors: a model of concurrent computation in distributed systems.* PhD thesis, MIT, 1985.

[2] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[3] Chas Emerick, Brian Carper, and Christophe Grand. *Clojure Programming.* O'Reilly, 2012.

[4] Tim Harris, Simon Marlow, Simon P. Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '05*, pages 48–60, New York, New York, USA, 2005. ACM Press.

[5] C. A. R. Hoare. Communicating sequential processes. *Comm. of the ACM*, 21(8):666–677, 1978.

[6] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[7] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC'95: Proc. of the fourteenth annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[8] Samira Tasharofi, Peter Dinges, and Ralph Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *Proc. of ECOOP'13*, Montpellier, France, 2013.

Session Type Isomorphisms

Mariangiola Dezani-Ciancaglini[1], Luca Padovani[1] and Jovanka Pantovic[2]

[1] Università di Torino, Italy
[2] Univerzitet u Novom Sadu, Serbia

1 Introduction. We have all experienced, possibly during a travel abroad, using an ATM that behaves differently from the ones we are familiar with. Although the information requested for accomplishing a transaction is essentially always the same – the PIN, the amount of money we want to withdraw, whether or not we want a receipt – we may be prompted to enter such information in an unexpected order, or we may be asked to dismiss sudden popup windows containing informative messages – "charges may apply" – or commercials. Subconsciously, we *adapt* our behavior so that it matches the one of the ATM we are operating, and we can usually complete the transaction provided that the expected and actual behaviors are *sufficiently similar*. An analogous problem arises during software development or execution, when we need a component that exhibits some desired behavior while the components we have at hand exhibit similar, but not exactly equal, behaviors which could nonetheless be adapted to the one we want. In this article, we explore one particular way of realizing such adaptation in the context of binary sessions, where the behavior of components is specified as session types.

There are two key notions to be made precise in the previous paragraph: first of all, we must clarify what it means for two behaviors to be "similar" to the point that one can be adapted into the other; second, as for the "subconscious" nature of adaptation, we translate this into the ability to synthesize the adapter automatically – *i.e.* without human intervention – just by looking at the differences between the required and actual behaviors of the component. Clearly we have to find a trade-off: the coarser the similarity notion is the better, for this means widening the range of components we can use; at the same time, it is reasonable to expect that the more two components differ, the harder it gets to automatically synthesize a sensible adapter between them. The methodology we propose in this work is based on the theory of *type isomorphisms* [10]. Intuitively, two types T and S are isomorphic if there exist two adapters $A : T \to S$ and $B : S \to T$ such that A transforms a component of type (or, that behaves like) T into one of type S, and B does just the opposite. It is required that these transformations must not *lose any information*. This can be expressed saying that if we compose A and B in any order they annihilate each other, that is we obtain adapters $A \rceil\lceil B : T \to T$ and $B \rceil\lceil A : S \to S$ that are equivalent to the "identity" trasformations on T and S respectively.

In the following we formalize these concepts: we define syntax and semantics of processes as well as a notion of process equivalence (Section 2). Next, we introduce a type system for processes, the notion of session type isomorphism, and show off samples of the transformations we can capture in this framework (Section 3). We conclude with a quick survery of related works and open problems (Section 4).

2 Processes. We let $m, n, \ldots$ range over integer numbers; we let c range over the set $\{\mathtt{l}, \mathtt{r}\}$ of *channels* and ℓ range over the set $\{\mathtt{inl}, \mathtt{inr}\}$ of *selectors*. We define an involution $\overline{\cdot}$ over channels such that $\overline{\mathtt{l}} = \mathtt{r}$. We assume a set of *basic values* $\mathsf{v}, \ldots$ and *basic types* $t, s, \ldots$ that include the unitary value $\mathtt{()}$ of type $\mathtt{unit}$, the booleans $\mathtt{true}$ and $\mathtt{false}$ of type $\mathtt{bool}$, and the integer numbers of type $\mathtt{int}$. We write $\mathsf{v} \in t$ meaning that v has type t. We use a countable set of *variables* $x, y, \ldots$; *expressions* $\mathsf{e}, \ldots$ are either variables or values or the equality $\mathsf{e}_1 = \mathsf{e}_2$ between two expressions. Additional expression forms can be added without substantial issues. *Processes* are defined by the grammar

$$P \ ::= \ \mathbf{0} \ \mid \ \mathsf{c}?(x:t).P \ \mid \ \mathsf{c}!\langle \mathsf{e} \rangle.P \ \mid \ \mathsf{c} \triangleleft \ell.P \ \mid \ \mathsf{c} \triangleright \{P, Q\} \ \mid \ \mathtt{if}\ \mathsf{e}\ \mathtt{then}\ P\ \mathtt{else}\ Q \ \mid \ P \rceil\lceil Q$$

which includes the terminated process $\mathbf{0}$, input $\mathsf{c}?(x:t).P$ and output $\mathsf{c}!\langle \mathsf{e} \rangle.P$ processes, as well as labeled-driven selection $\mathsf{c} \triangleleft \ell.P$ and branching $\mathsf{c} \triangleright \{P, Q\}$, the conditional process $\mathtt{if}\ \mathsf{e}\ \mathtt{then}\ P\ \mathtt{else}\ Q$,

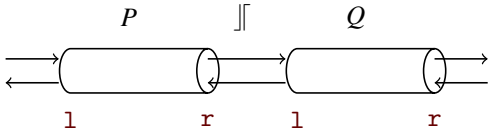
Table 1: Reduction relation.

$$\text{[R-COMM 1]}\quad \frac{\mathsf{e}\downarrow \mathsf{v} \qquad \mathsf{v}\in t}{\mathtt{r}!\langle \mathsf{e}\rangle.P \mathbin{⫿} \mathtt{l}?(x:t).Q \longrightarrow P \mathbin{⫿} Q\{\mathsf{v}/x\}} \qquad \text{[R-COMM 2]}\quad \frac{\mathsf{e}\downarrow \mathsf{v} \qquad \mathsf{v}\in t}{\mathtt{r}?(x:t).P \mathbin{⫿} \mathtt{l}!\langle \mathsf{e}\rangle.Q \longrightarrow P\{\mathsf{v}/x\} \mathbin{⫿} Q}$$

$$\text{[R-CHOICE 1]}\quad \mathtt{r}\triangleleft\ell.P \mathbin{⫿} \mathtt{l}\triangleright\{Q_{\mathtt{inl}}, Q_{\mathtt{inr}}\} \longrightarrow P \mathbin{⫿} Q_\ell \qquad \text{[R-CHOICE 2]}\quad \mathtt{r}\triangleright\{P_{\mathtt{inl}}, P_{\mathtt{inr}}\} \mathbin{⫿} \mathtt{l}\triangleleft\ell.Q \longrightarrow P_\ell \mathbin{⫿} Q$$

$$\text{[R-COND]}\quad \frac{\mathsf{e}\downarrow \mathsf{v} \qquad \mathsf{v}\in \mathtt{bool}}{\mathtt{if}\ \mathsf{e}\ \mathtt{then}\ P_{\mathtt{true}}\ \mathtt{else}\ P_{\mathtt{false}} \longrightarrow P_{\mathsf{v}}} \qquad \text{[R-CONTEXT]}\quad \frac{P \longrightarrow Q}{\mathscr{C}[P] \longrightarrow \mathscr{C}[Q]} \qquad \text{[R-STRUCT]}\quad \frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}$$

and parallel composition $P \mathbin{⫿} Q$. The peculiarity of the calculus is that communication occurs only between adjacent processes. Such communication model is exemplified by the diagram below which depicts the composition $P \mathbin{⫿} Q$. Each process sends and receives messages through the channels $\mathtt{l}$ and $\mathtt{r}$. Messages sent by P on $\mathtt{r}$ are received by Q from $\mathtt{l}$, and messages sent by Q on $\mathtt{l}$ are received by P from $\mathtt{r}$. Therefore, unlike more conventional parallel composition operators, $⫿$ is associative but not symmetric in general. Intuitively, $P \mathbin{⫿} Q$ models a binary session where P and Q are the processes accessing the two endpoints of the session. By compositionality, we can also represent more complex scenarios like $P \mathbin{⫿} A \mathbin{⫿} Q$ where the interaction of the same two processes P and Q is mediated by an adapter A that filters and/or transforms the messages exchanged between P and Q. In turn, A may be the parallel composition of several simpler adapters.



The operational semantics of processes is formalized as a reduction relation closed by reduction contexts and a structural congruence relation. *Reduction contexts* $\mathscr{C}$ are defined by the grammar

$$\mathscr{C} \ ::= \ [\,] \ \mid \ \mathscr{C} \mathbin{⫿} P \ \mid \ P \mathbin{⫿} \mathscr{C}$$

and, as usual, we write $\mathscr{C}[P]$ for the process obtained by replacing the hole in $\mathscr{C}$ with P.

Structural congruence is the least congruence defined by the rules

$$\mathbf{0} \mathbin{⫿} \mathbf{0} \equiv \mathbf{0} \qquad P \mathbin{⫿} (Q \mathbin{⫿} R) \equiv (P \mathbin{⫿} Q) \mathbin{⫿} R$$

while *reduction* is the least relation $\longrightarrow$ defined by the rules in Table 1. The rules are familiar and therefore unremarkable. We assume a deterministic *evaluation* relation $\mathsf{e}\downarrow\mathsf{v}$ expressing the fact that v is the value of e. We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$ and $P \not\longrightarrow$ if there is no Q such that $P \longrightarrow Q$. With these notions we can characterize the set of correct processes, namely those that complete every interaction and eventually reduce to $\mathbf{0}$:

Definition 1 (correct process)**.** We say that a process P is *correct* if $P \longrightarrow^* Q \not\longrightarrow$ implies $Q \equiv \mathbf{0}$.

A key ingredient of our development is a notion of process equivalence that relates two processes P and Q whenever they can be completed by the same contexts $\mathscr{C}$ to form a correct process. Formally:

Definition 2 (equivalence)**.** We say that two processes P and Q are *equivalent*, notation $P \approx Q$, whenever for every $\mathscr{C}$ we have that $\mathscr{C}[P]$ is correct if and only if $\mathscr{C}[Q]$ is correct.

Table 2: Typing rules for expressions and processes.

$$\text{[T-VAR]}\ \ \Gamma, x:t \vdash x:t \qquad \text{[T-VALUE]}\ \ \frac{\mathsf{v} \in t}{\Gamma \vdash \mathsf{v}:t} \qquad \text{[T-EQ]}\ \ \frac{\Gamma \vdash \mathsf{e}_1:t \qquad \Gamma \vdash \mathsf{e}_2:t}{\Gamma \vdash \mathsf{e}_1 = \mathsf{e}_2 : \mathtt{bool}}$$

$$\text{[T-INPUT]}\ \ \frac{\Gamma, x:t \vdash P \blacktriangleright \{\mathsf{c}:T, \overline{\mathsf{c}}:S\}}{\Gamma \vdash \mathsf{c}?(x:t).P \blacktriangleright \{\mathsf{c}:?t.T, \overline{\mathsf{c}}:S\}}$$

$$\text{[T-OUTPUT]}\ \ \frac{\Gamma \vdash \mathsf{e}:t \qquad \Gamma \vdash P \blacktriangleright \{\mathsf{c}:T, \overline{\mathsf{c}}:S\}}{\Gamma \vdash \mathsf{c}!\langle \mathsf{e} \rangle.P \blacktriangleright \{\mathsf{c}:!t.T, \overline{\mathsf{c}}:S\}} \qquad \text{[T-BRANCH]}\ \ \frac{\Gamma \vdash P_i \blacktriangleright \{\mathsf{c}:T_i, \overline{\mathsf{c}}:S\}^{\ (i=1,2)}}{\Gamma \vdash \mathsf{c} \triangleright \{P_1, P_2\} \blacktriangleright \{\mathsf{c}:T_1 + T_2, \overline{\mathsf{c}}:S\}}$$

$$\text{[T-SELECT LEFT]}\ \ \frac{\Gamma \vdash P \blacktriangleright \{\mathsf{c}:T_1, \overline{\mathsf{c}}:S\}}{\Gamma \vdash \mathsf{c} \triangleleft \mathtt{inl}.P \blacktriangleright \{\mathsf{c}:T_1 \oplus T_2, \overline{\mathsf{c}}:S\}} \qquad \text{[T-SELECT RIGHT]}\ \ \frac{\Gamma \vdash P \blacktriangleright \{\mathsf{c}:T_2, \overline{\mathsf{c}}:S\}}{\Gamma \vdash \mathsf{c} \triangleleft \mathtt{inr}.P \blacktriangleright \{\mathsf{c}:T_1 \oplus T_2, \overline{\mathsf{c}}:S\}}$$

$$\text{[T-IDLE]}\ \ \Gamma \vdash \mathbf{0} \blacktriangleright \{\mathtt{l}:\mathtt{end}, \mathtt{r}:\mathtt{end}\}$$

$$\text{[T-CONDITIONAL]}\ \ \frac{\Gamma \vdash \mathsf{e}:\mathtt{bool} \qquad \Gamma \vdash P_i \blacktriangleright \{\mathtt{l}:T, \mathtt{r}:S\}^{\ (i=1,2)}}{\Gamma \vdash \mathtt{if}\ \mathsf{e}\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \blacktriangleright \{\mathtt{l}:T, \mathtt{r}:S\}} \qquad \text{[T-PARALLEL]}\ \ \frac{\Gamma \vdash P \blacktriangleright \{\mathtt{l}:T, \mathtt{r}:T'\} \qquad \Gamma \vdash Q \blacktriangleright \{\mathtt{l}:\overline{T}', \mathtt{r}:S\}}{\Gamma \vdash P \between Q \blacktriangleright \{\mathtt{l}:T, \mathtt{r}:S\}}$$

Note that the relation $\approx$ differs from more conventional equivalences between processes. In particular, $\approx$ is insensitive to the exact time when visible actions are made available on the two interfaces of a process. For example, we have

$$\mathtt{l}?(x:\mathtt{int}).\mathtt{r}!\langle \mathtt{true} \rangle.\mathtt{l}?(y:\mathtt{unit}) \approx \mathtt{l}?(x:\mathtt{int}).\mathtt{l}?(y:\mathtt{unit}).\mathtt{r}!\langle \mathtt{true} \rangle \tag{1}$$

despite the fact that the two processes perform visible actions in different orders. Note that the processes in (1) are not (weakly) bisimilar.

3 Type System and Isomorphisms. *Session types* $T, S, \ldots$ are defined by the grammar

$$T \ ::= \ \mathtt{end} \ \mid \ ?t.T \ \mid \ !t.T \ \mid \ T + S \ \mid \ T \oplus S$$

and are fairly standard, except for branching $T + S$ and selection $T \oplus S$ which are binary instead of n-ary operators, consistently with the process language. As usual, we denote by $\overline{T}$ the *dual* of T, namely the session type obtained by swapping inputs with outputs and selections with branches in T.

We let Γ range over *environments* which are finite maps from variables to types of the form $x_1 : t_1, \ldots, x_n : t_n$. The typing rules are given in Table 2. Judgments have the form $\Gamma \vdash \mathsf{e} : t$ stating that e is well typed and has type t in the environment Γ and $\Gamma \vdash P \blacktriangleright \{\mathsf{c}:T, \overline{\mathsf{c}}:S\}$ stating that P is well typed in the environment Γ and uses channel c according to T and $\overline{\mathsf{c}}$ according to S.

Theorem 1. If $\vdash P \blacktriangleright \{\mathtt{l}:\mathtt{end}, \mathtt{r}:\mathtt{end}\}$, then P is correct.

To have an isomorphism between two session types T and S, we need a process A that behaves according to $\overline{T}$ on its left interface and according to S on its right interface. In this way, the process "transforms" T into S. Symmetrically, there must be a process B that performs the inverse transformation. Not all of these transformations are isomorphisms, because we also require that these transformations must not entail any *loss of information*. Given a session type T, the simplest process with this property is the *identity* process id_T defined below:

$$\mathsf{id}_{\mathtt{end}} = \mathbf{0} \qquad \begin{array}{ll} \mathsf{id}_{!t.T} = \mathtt{l}?(x:t).\mathtt{r}!\langle x \rangle.\mathsf{id}_T & \mathsf{id}_{T \oplus S} = \mathtt{l} \triangleright \{\mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_T, \mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_S\} \\ \mathsf{id}_{?t.T} = \mathtt{r}?(x:t).\mathtt{l}!\langle x \rangle.\mathsf{id}_T & \mathsf{id}_{T+S} = \mathtt{r} \triangleright \{\mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_T, \mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_S\} \end{array}$$

Notice that $\vdash \mathsf{id}_T \blacktriangleright \{\mathtt{l} : \overline{T}, \mathtt{r} : T\}$. We can now formalize the notion of session type isomorphism:

Definition 3 (isomorphism)**.** We say that the session types T and S are *isomorphic*, notation $T \cong S$, if there exist two processes A and B such that $\vdash A \blacktriangleright \{\mathtt{l} : \overline{T}, \mathtt{r} : S\}$ and $\vdash B \blacktriangleright \{\mathtt{l} : \overline{S}, \mathtt{r} : T\}$ and $A \mathbin{⟦} B \approx \mathsf{id}_T$ and $B \mathbin{⟦} A \approx \mathsf{id}_S$.

Example 1. Let $T \stackrel{\text{def}}{=} \mathtt{!int}.\mathtt{!bool}.\mathtt{end}$ and $S \stackrel{\text{def}}{=} \mathtt{!bool}.\mathtt{!int}.\mathtt{end}$ and observe that T and S differ in the order in which messages are sent. Then we have $T \cong S$. Indeed, if we take

$$A \stackrel{\text{def}}{=} \mathtt{l?}(x : \mathtt{int}).\mathtt{l?}(y : \mathtt{bool}).\mathtt{r!}\langle y\rangle.\mathtt{r!}\langle x\rangle.\mathbf{0} \quad \text{and} \quad B \stackrel{\text{def}}{=} \mathtt{l?}(x : \mathtt{bool}).\mathtt{l?}(y : \mathtt{int}).\mathtt{r!}\langle y\rangle.\mathtt{r!}\langle x\rangle.\mathbf{0}$$

we derive $\vdash A \blacktriangleright \{\mathtt{l} : \overline{T}, \mathtt{r} : S\}$ and $\vdash B \blacktriangleright \{\mathtt{l} : \overline{S}, \mathtt{r} : T\}$ and moreover $A \mathbin{⟦} B \approx \mathsf{id}_T$ and $B \mathbin{⟦} A \approx \mathsf{id}_S$. ■

Example 2. Showing that two session types are *not* isomorphic is more challenging since we must prove that there is no pair of processes A and B that turns one into the other without losing information. We do so reasoning by contradiction. Suppose for example that $\mathtt{!int.end}$ and $\mathtt{end}$ are isomorphic. Then, there must exist $\vdash A \blacktriangleright \{\mathtt{l} : \mathtt{?int.end}, \mathtt{r} : \mathtt{end}\}$ and $\vdash B \blacktriangleright \{\mathtt{l} : \mathtt{end}, \mathtt{r} : \mathtt{!int.end}\}$. The adapter B is suspicious, since it must send a message of type $\mathtt{int}$ on channel $\mathtt{r}$ without ever receiving such a message from channel $\mathtt{l}$. Then, it must be the case that B "makes up" such a message, say it is n (observe that our calculus is deterministic, so B will always output the same integer n). We can now unmask B showing a context that distinguishes $\mathsf{id}_{\mathtt{!int.end}}$ from $A \mathbin{⟦} B$. Consider

$$\mathscr{C} \stackrel{\text{def}}{=} \mathtt{r!}\langle n+1\rangle.\mathbf{0} \mathbin{⟦} [\,] \mathbin{⟦} \mathtt{l?}(x : \mathtt{int}).\mathtt{if}\ x = n+1\ \mathtt{then}\ \mathbf{0}\ \mathtt{else}\ \mathtt{r!}\langle \mathtt{false}\rangle.\mathbf{0}$$

and observe that $\mathscr{C}[\mathsf{id}_{\mathtt{!int.end}}]$ is correct whereas $\mathscr{C}[A \mathbin{⟦} B]$ is not because

$$\mathscr{C}[A \mathbin{⟦} B] \longrightarrow^* \mathbf{0} \mathbin{⟦} \mathtt{if}\ n = n+1\ \mathtt{then}\ \mathbf{0}\ \mathtt{else}\ \mathtt{r!}\langle \mathtt{false}\rangle.\mathbf{0} \longrightarrow \mathbf{0} \mathbin{⟦} \mathtt{r!}\langle \mathtt{false}\rangle.\mathbf{0} \not\longrightarrow$$

This means that $A \mathbin{⟦} B \not\approx \mathsf{id}_{\mathtt{!int.end}}$, contradicting the hypothesis that A and B were the witnesses of the isomorphism $\mathtt{!int.end} \cong \mathtt{end}$. ■

Example 3. Another interesting pair of non-isomorphic types is given by $T \stackrel{\text{def}}{=} \mathtt{?int.!bool.end}$ and $S \stackrel{\text{def}}{=} \mathtt{!bool.?int.end}$. A lossless transformation from S to T can be realized by the process

$$B \stackrel{\text{def}}{=} \mathtt{l?}(x : \mathtt{bool}).\mathtt{r?}(y : \mathtt{int}).\mathtt{r!}\langle x\rangle.\mathtt{l!}\langle y\rangle.\mathbf{0},$$

which reads one message from each interface and forwards it to the opposite one. The inverse transformation from T to S is unachieavable without loss of information. Such process necessarily sends at least one message (of type $\mathtt{int}$ or of type $\mathtt{bool}$) on one interface *before* it receives the message of the same type from the opposite interface. Therefore, just like in Example 2, such process must guess the message to send, and in most cases such message does not coincide with the one the process was supposed to forward. ■

Table 3 gathers the session type isomorphisms that we have identified. There is a perfect duality between the odd-indexed axioms (about outputs, on the left) and the even-indexed axioms (about inputs, on the right), so we briefly discuss the odd-indexed axioms only. Axiom [A1] is a generalization of the isomorphism discussed in Example 1 and is proved by a similar adapter. Axiom [A3] distributes the *same* output on a selection. Basically, this means that the moment of selection is irrelevant with respect to other adjacent output operations. Axiom [A5] shows that sending the unitary value provides no information and therefore is a superfluous operation. Axiom [A7] shows that sending a boolean value is equivalent to making a selection, provided that the continuation does not depend on the particular

Table 3: Session type isomorphisms.

[A1]	$!t.!s.T \cong !s.!t.T$	[A2]	$?t.?s.T \cong ?s.?t.T$
[A3]	$!t.(T \oplus S) \cong !t.T \oplus !t.S$	[A4]	$?t.(T + S) \cong ?t.T + ?t.S$
[A5]	$!\mathtt{unit}.T \cong T$	[A6]	$?\mathtt{unit}.T \cong T$
[A7]	$!\mathtt{bool}.T \cong T \oplus T$	[A8]	$?\mathtt{bool}.T \cong T + T$
[A9]	$T \oplus S \cong S \oplus T$	[A10]	$T + S \cong S + T$
[A11]	$(T_1 \oplus T_2) \oplus T_3 \cong T_1 \oplus (T_2 \oplus T_3)$	[A12]	$(T_1 + T_2) + T_3 \cong T_1 + (T_2 + T_3)$

Table 4: Adapters for type isomorphism.

$A_1 = \mathtt{l}?(x:t).\mathtt{l}?(y:s).\mathtt{r}!\langle y\rangle.\mathtt{r}!\langle x\rangle.\mathsf{id}_T$	$B_1 = \mathtt{l}?(x:s).\mathtt{l}?(y:t).\mathtt{r}!\langle y\rangle.\mathtt{r}!\langle x\rangle.\mathsf{id}_T$
$A_2 = \mathtt{r}?(x:t).\mathtt{r}?(y:s).\mathtt{l}!\langle y\rangle.\mathtt{l}!\langle x\rangle.\mathsf{id}_T$	$B_2 = \mathtt{r}?(x:s).\mathtt{r}?(y:t).\mathtt{l}!\langle y\rangle.\mathtt{l}!\langle x\rangle.\mathsf{id}_T$
$A_3 = \mathtt{l}?(x:t).\mathtt{l} \triangleright \{\mathtt{r} \triangleleft \mathtt{inl}.\mathtt{r}!\langle x\rangle.\mathsf{id}_T, \mathtt{r} \triangleleft \mathtt{inr}.\mathtt{r}!\langle x\rangle.\mathsf{id}_S\}$	
$B_3 = \mathtt{l} \triangleright \{\mathtt{l}?(x:t).\mathtt{r}!\langle x\rangle.\mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_T, \mathtt{l}?(x:t).\mathtt{r}!\langle x\rangle.\mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_S\}$	
$A_4 = \mathtt{r} \triangleright \{\mathtt{r}?(x:t).\mathtt{l}!\langle x\rangle.\mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_T, \mathtt{r}?(x:t).\mathtt{l}!\langle x\rangle.\mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_S\}$	
$B_4 = \mathtt{r}?(x:t).\mathtt{r} \triangleright \{\mathtt{l} \triangleleft \mathtt{inl}.\mathtt{l}!\langle x\rangle.\mathsf{id}_T, \mathtt{l} \triangleleft \mathtt{inr}.\mathtt{l}!\langle x\rangle.\mathsf{id}_S\}$	
$A_5 = \mathtt{l}?(x:\mathtt{unit}).\mathsf{id}_T$	$B_5 = \mathtt{r}!\langle ()\rangle.\mathsf{id}_T$
$A_6 = \mathtt{l}!\langle ()\rangle.\mathsf{id}_T$	$B_6 = \mathtt{r}?(x:\mathtt{unit}).\mathsf{id}_T$
$A_7 = \mathtt{l}?(x:\mathtt{bool}).\mathtt{if}\ x\ \mathtt{then}\ \mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_T\ \mathtt{else}\ \mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_T$	$B_7 = \mathtt{l} \triangleright \{\mathtt{r}!\langle \mathtt{true}\rangle.\mathsf{id}_T, \mathtt{r}!\langle \mathtt{false}\rangle.\mathsf{id}_T\}$
$A_8 = \mathtt{r} \triangleright \{\mathtt{l}!\langle \mathtt{true}\rangle.\mathsf{id}_T, \mathtt{l}!\langle \mathtt{false}\rangle.\mathsf{id}_T\}$	$B_8 = \mathtt{r}?(x:\mathtt{bool}).\mathtt{if}\ x\ \mathtt{then}\ \mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_T\ \mathtt{else}\ \mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_T$
$A_9 = \mathtt{l} \triangleright \{\mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_T, \mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_S\}$	$B_9 = \mathtt{l} \triangleright \{\mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_S, \mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_T\}$
$A_{10} = \mathtt{r} \triangleright \{\mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_S, \mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_T\}$	$B_{10} = \mathtt{r} \triangleright \{\mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_T, \mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_S\}$
$A_{11} = \mathtt{l} \triangleright \{\mathtt{l} \triangleright \{\mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_{T_1}, \mathtt{r} \triangleleft \mathtt{inr}.\mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_{T_2}\}, \mathtt{r} \triangleleft \mathtt{inr}.\mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_{T_3}\}$	
$B_{11} = \mathtt{l} \triangleright \{\mathtt{r} \triangleleft \mathtt{inl}.\mathtt{r} \triangleleft \mathtt{inl}.\mathsf{id}_{T_1}, \mathtt{l} \triangleright \{\mathtt{r} \triangleleft \mathtt{inl}.\mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_{T_2}, \mathtt{r} \triangleleft \mathtt{inr}.\mathsf{id}_{T_3}\}\}$	
$A_{12} = \mathtt{r} \triangleright \{\mathtt{l} \triangleleft \mathtt{inl}.\mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_{T_1}, \mathtt{r} \triangleright \{\mathtt{l} \triangleleft \mathtt{inl}.\mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_{T_2}, \mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_{T_3}\}\}$	
$B_{12} = \mathtt{r} \triangleright \{\mathtt{r} \triangleright \{\mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_{T_1}, \mathtt{l} \triangleleft \mathtt{inr}.\mathtt{l} \triangleleft \mathtt{inl}.\mathsf{id}_{T_2}\}, \mathtt{l} \triangleleft \mathtt{inr}.\mathtt{l} \triangleleft \mathtt{inr}.\mathsf{id}_{T_3}\}$	

boolean value that is sent. In general, any data type with finitely many values can be encoded as possibly nested choices. Axiom [A9], corresponding to the commutativity of $\oplus$ wrt $\cong$, shows that the actual label used for making a selection is irrelevant, only the continuation matters. Axiom [A11], corresponding to the associativity for $\oplus$ wrt $\cong$, generalizes the irrelevance of labels seen in [A9] to nested selections. Since $\cong$ is a congruence, the axioms in Table 3 can also be closed by transitivity and arbitrary session type contexts.

Table 4 gives all the adapters of the axioms in Table 3. Then the soundness of the axioms in Table 3 amounts to prove:

$$\vdash A_i \blacktriangleright \{\mathtt{l} : \overline{T_i}, \mathtt{r} : S_i\} \qquad \vdash B_i \blacktriangleright \{\mathtt{l} : \overline{S_i}, \mathtt{r} : T_i\} \tag{2}$$

$$A_i \rceil\!\lceil B_i \approx \mathsf{id}_{T_i} \qquad B_i \rceil\!\lceil A_i \approx \mathsf{id}_{S_i} \tag{3}$$

where T_i is the l.h.s. and S_i is the r.h.s. of the axiom [Ai] for $1 \leq i \leq 12$.

Point 2 can be easily shown by cases on the definitions of A_i and B_i taking into account that $\vdash \mathsf{id}_T \blacktriangleright \{\mathtt{l} : \overline{T}, \mathtt{r} : T\}$ for all types T.

For Point 3 we define a *symbolic reduction relation* which preserves equivalence of closed and typed processes (Theorem 2). This is enough since we will reduce parallel compositions of adapters

Table 5: Symbolic reduction relation.

$$[\text{SR-UP 1}]\ \mathtt{l}?(x:t).P \between Q \rightsquigarrow \mathtt{l}?(x:t).(P \between Q) \qquad [\text{SR-UP 2}]\ P \between \mathtt{r}?(x:t).Q \rightsquigarrow \mathtt{r}?(x:t).(P \between Q)$$
$$[\text{SR-UP 3}]\ \mathtt{l}!\langle x\rangle.P \between Q \rightsquigarrow \mathtt{l}!\langle x\rangle.(P \between Q) \qquad [\text{SR-UP 4}]\ P \between \mathtt{r}!\langle x\rangle.Q \rightsquigarrow \mathtt{r}!\langle x\rangle.(P \between Q)$$
$$[\text{SR-UP 5}]\ \mathtt{l} \triangleright \{P_{\mathtt{inl}}, P_{\mathtt{inr}}\} \between Q \rightsquigarrow \mathtt{l} \triangleright \{P_{\mathtt{inl}} \between Q, P_{\mathtt{inr}} \between Q\} \qquad [\text{SR-UP 7}]\ \mathtt{l} \triangleleft \ell.P \between Q \rightsquigarrow \mathtt{l} \triangleleft \ell.(P \between Q)$$
$$[\text{SR-UP 6}]\ P \between \mathtt{r} \triangleright \{Q_{\mathtt{inl}}, Q_{\mathtt{inr}}\} \rightsquigarrow \mathtt{r} \triangleright \{P \between Q_{\mathtt{inl}}, P \between Q_{\mathtt{inr}}\} \qquad [\text{SR-UP 8}]\ P \between \mathtt{r} \triangleleft \ell.Q \rightsquigarrow \mathtt{r} \triangleleft \ell.(P \between Q)$$
$$[\text{SR-UP 9}]\ (\mathtt{if}\ x\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) \between Q \rightsquigarrow \mathtt{if}\ x\ \mathtt{then}\ (P_1 \between Q)\ \mathtt{else}\ (P_2 \between Q)$$
$$[\text{SR-UP 10}]\ P \between (\mathtt{if}\ x\ \mathtt{then}\ Q_1\ \mathtt{else}\ Q_2) \rightsquigarrow \mathtt{if}\ x\ \mathtt{then}\ (P \between Q_1)\ \mathtt{else}\ (P \between Q_2)$$
$$[\text{SR-SWAP 1}]\ \mathtt{c}?(x:t).\overline{\mathtt{c}}?(y:s).P \rightsquigarrow \overline{\mathtt{c}}?(y:s).\mathtt{c}?(x:t).P \qquad [\text{SR-SWAP 2}]\ \mathtt{c}!\langle x\rangle.\overline{\mathtt{c}}!\langle y\rangle.P \rightsquigarrow \overline{\mathtt{c}}!\langle y\rangle.\mathtt{c}!\langle x\rangle.P$$
$$[\text{SR-SWAP 3}]\ \mathtt{c}?(x:t).\overline{\mathtt{c}}!\langle y\rangle.P \leftrightsquigarrow \overline{\mathtt{c}}!\langle y\rangle.\mathtt{c}?(x:t).P \quad x \neq y$$
$$[\text{SR-SWAP 4}]\ \mathtt{c}?(x:t).\overline{\mathtt{c}} \triangleleft \ell.P \leftrightsquigarrow \overline{\mathtt{c}} \triangleleft \ell.\mathtt{c}?(x:t).P$$
$$[\text{SR-SWAP 5}]\ \mathtt{c}!\langle x\rangle.\overline{\mathtt{c}} \triangleleft \ell.P \leftrightsquigarrow \overline{\mathtt{c}} \triangleleft \ell.\mathtt{c}!\langle x\rangle.P$$
$$[\text{SR-SWAP 6}]\ \mathtt{c}?(x:t).\overline{\mathtt{c}} \triangleright \{P,Q\} \leftrightsquigarrow \overline{\mathtt{c}} \triangleright \{\mathtt{c}?(x:t).P, \mathtt{c}?(x:t).Q\}$$
$$[\text{SR-SWAP 7}]\ \mathtt{c}!\langle x\rangle.\overline{\mathtt{c}} \triangleright \{P,Q\} \leftrightsquigarrow \overline{\mathtt{c}} \triangleright \{\mathtt{c}!\langle x\rangle.P, \mathtt{c}!\langle x\rangle.Q\}$$
$$[\text{SR-SWAP 8}]\ \mathtt{c} \triangleright \{\overline{\mathtt{c}} \triangleleft \ell.P, \overline{\mathtt{c}} \triangleleft \ell.Q\} \leftrightsquigarrow \overline{\mathtt{c}} \triangleleft \ell.\mathtt{c} \triangleright \{P,Q\}$$
$$[\text{SR-SWAP 9}]\ \mathtt{c} \triangleleft \ell.\overline{\mathtt{c}} \triangleleft \ell'.P \leftrightsquigarrow \overline{\mathtt{c}} \triangleleft \ell'.\mathtt{c} \triangleleft \ell.P$$
$$[\text{SR-SWAP 10}]\ \mathtt{c} \triangleright \{\overline{\mathtt{c}} \triangleright \{P_1,Q_1\}, \overline{\mathtt{c}} \triangleright \{P_2,Q_2\}\} \leftrightsquigarrow \overline{\mathtt{c}} \triangleright \{\mathtt{c} \triangleright \{P_1,P_2\}, \mathtt{c} \triangleright \{Q_1,Q_2\}\}$$
$$[\text{SR-COND}]\ \mathtt{if}\ x\ \mathtt{then}\ \mathtt{c}!\langle \mathtt{true}\rangle.P\ \mathtt{else}\ \mathtt{c}!\langle \mathtt{false}\rangle.P \rightsquigarrow \mathtt{c}!\langle x\rangle.P$$
$$[\text{SR-COMM 1}]\ \mathtt{r}!\langle y\rangle.P \between \mathtt{l}?(x:t).Q \rightsquigarrow P \between Q\{y/x\} \qquad [\text{SR-COMM 2}]\ \mathtt{r}?(x:t).P \between \mathtt{l}!\langle y\rangle.Q \rightsquigarrow P\{y/x\} \between Q$$
$$[\text{SR-CHOICE 1}]\ \mathtt{r} \triangleleft \ell.P \between \mathtt{l} \triangleright \{Q_{\mathtt{inl}}, Q_{\mathtt{inr}}\} \rightsquigarrow P \between Q_\ell \qquad [\text{SR-CHOICE 2}]\ \mathtt{r} \triangleright \{P_{\mathtt{inl}}, P_{\mathtt{inr}}\} \between \mathtt{l} \triangleleft \ell.Q \rightsquigarrow P_\ell \between Q$$
$$[\text{SR-ID}]\ \mathsf{id}_T \between \mathsf{id}_T \rightsquigarrow \mathsf{id}_T \qquad [\text{SR-CONTEXTS}]\ \frac{P \rightsquigarrow Q}{\mathscr{E}[P] \rightsquigarrow \mathscr{E}[Q]}$$

(Theorem 3). The rules of this relation are given in Table 5, where $\leftrightsquigarrow$ stands for reduction in both directions and *symbolic reduction contexts* $\mathscr{E}$ are defined by:

$$\mathscr{E} \quad ::= \quad [\,] \;\mid\; \mathtt{c}?(x:t).\mathscr{E} \;\mid\; \mathtt{c}!\langle \mathsf{e}\rangle.\mathscr{E} \;\mid\; \mathtt{c} \triangleleft \ell.\mathscr{E} \;\mid\; \mathtt{c} \triangleright \{\mathscr{E}, Q\} \;\mid\; \mathtt{c} \triangleright \{P, \mathscr{E}\}$$
$$\mid \quad \mathtt{if}\ \mathsf{e}\ \mathtt{then}\ P\ \mathtt{else}\ \mathscr{E} \;\mid\; \mathtt{if}\ \mathsf{e}\ \mathtt{then}\ \mathscr{E}\ \mathtt{else}\ Q$$

We call this a symbolic reduction relation because it also reduces processes with free variables. We notice that this reduction applied to two parallel processes:

1. moves up the communications/selections/branchings on the left channel of the left process and the communications/selections/branchings on the right channel of the right process and the conditionals,
2. executes the communications/selections/branchings between the right channel of the left process and the left channel of the right process when possible,
3. eliminates superfluous identities,
4. swaps communications/selections/branchings on different channels when this is not forbidden by bound variables.

The more interesting rule is [SR-COND], that transforms a conditional in an output.

Theorem 2. If P is a closed and typed process and $P \rightsquigarrow^* Q$, then $P \approx Q$.

Theorem 3. $A_i \between B_i \rightsquigarrow^* \mathsf{id}_{T_i}$ and $B_i \between A_i \rightsquigarrow^* \mathsf{id}_{S_i}$ for $1 \leq i \leq 12$.

Point 3 is a straightforward consequence of Theorems 2 and 3.

4 Concluding remarks. Type isomorphisms have been mainly studied for various λ-calculi [10]. In this paper we investigate a notion of isomorphism for session types that can be used for automatically adapting behaviors when their differences do not entail any loss of information. Adaptation in general [3] is much more permissive than in our approach, where we require adapters to be invertible. Moreover we only adapt processes as in [2, 11], while other works like [1, 8, 7] deal with adaptation of whole choreographies.Our approach shares many similarities with [5, 13] where *contracts* (as opposed to session types) describe the behavior of clients and Web services and filters/orchestrators mediate their interaction. The theory of orchestrators in [13] allows not only permutations of subsequent inputs and subsequent outputs, but also permutations between inputs and outputs if these have no causal dependencies. The induced morphism is therefore coarser than $\cong$, but it may entail some loss of information.

There are some open problems left for future research. The obvious ones are whether and how our theory extends to recursive and higher-order session types. Also, we do not know yet whether the set of axioms in Table 3 is *complete*. The point is that in the case of arrow, product and sum types or of arrow, intersection, union types, it is known that the set of isomorphisms is not finitely axiomatizable [12, 9, 6]. Despite the fact that session types incorporate constructs that closely resemble product and sum types, it may be the case that the particular structure of the type language allows for a finite axiomatization. A natural question is to what extend our results are a consequence of the presence of just two channels in the process language, or whether they would carry over to calculi with arbitrary channel names. A more interesting research direction is to consider this notion of session type isomorphism in relation to the work on session types and linear logic [4, 14].

REFERENCES.

[1] G. Anderson and J. Rathke. Dynamic Software Update for Message Passing Programs. In *APLAS'12*, LNCS 7705, pages 207–222. Springer, 2012.

[2] M. Bravetti, C. Di Giusto, J. A. Pérez, and G. Zavattaro. Adaptable Processes. *Log. Meth. Comp. Scie.*, 8(4), 2012.

[3] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. A Conceptual Framework for Adaptation. In *FASE'12*, LNCS 7212, pages 240–254. Springer, 2012.

[4] L. Caires and F. Pfenning. Session Types as Intuitionistic Linear Propositions. In *CONCUR'10*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.

[5] G. Castagna, N. Gesbert, and L. Padovani. A Theory of Contracts for Web Services. *ACM Trans. on Prog. Lang. and Syst.*, 31(5), 2009.

[6] M. Coppo, M. Dezani-Ciancaglini, I. Margaria, and M. Zacchi. Isomorphism of Intersection and Union Types, 2013. `http://www.di.unito.it/~dezani/papers/cdmz.pdf`.

[7] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Self-Adaptive Monitors for Multiparty Sessions. In *PDP'14*, 2014. to appear.

[8] M. dalla Preda, I. Lanese, J. Mauro, M. Gabbrielli, and S. Giallorenzo. Safe Run-time Adaptation of Distributed Systems, 2013. `http://www.cs.unibo.it/~lanese/publications/fulltext/safeadapt.pdf.gz`.

[9] M. Dezani-Ciancaglini, R. D. Cosmo, E. Giovannetti, and M. Tatsuta. On Isomorphisms of Intersection Types. *ACM Trans. on Comp. Logic*, 11(4):1–22, 2010.

[10] R. Di Cosmo. *Isomorphisms of Types: From Lambda-Calculus to Information Retrieval and Language Design*. Birkhauser Boston, 1995.

[11] C. Di Giusto and J. A. Pérez. Disciplined Structured Communications with Consistent Runtime Adaptation. In *SAC'13*, pages 1913–1918. ACM Press, 2013.

[12] M. Fiore, R. Di Cosmo, and V. Balat. Remarks on Isomorphisms in Typed Lambda Calculi with Empty and Sum Types. *Ann. Pure App. Logic*, 141(1–2):35–50, 2006.

[13] L. Padovani. Contract-Based Discovery of Web Services Modulo Simple Orchestrators. *Theor. Comp. Sci.*, 411:3328–3347, 2010.

[14] P. Wadler. Propositions as Sessions. In *ICFP'12*, pages 273–286. ACM, 2012.