university of groningen

University Medical Center Groningen

University of Groningen

Session-based concurrency in Maude

Ramírez Restrepo, Carlos Alberto; Jaramillo, Juan C.; Pérez, Jorge A.

Published in:
The Journal of Logical and Algebraic Methods in Programming

DOI:
10.1016/j.jlamp.2023.100872

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2023

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Ramírez Restrepo, C. A., Jaramillo, J. C., & Pérez, J. A. (2023). Session-based concurrency in Maude: Executable semantics and type checking. *The Journal of Logical and Algebraic Methods in Programming*, *133*, Article 100872. Advance online publication. https://doi.org/10.1016/j.jlamp.2023.100872

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

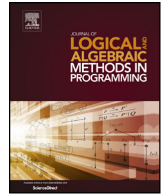
Download date: 01-11-2023

Journal of Logical and Algebraic Methods in Programming 133 (2023) 100872

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



Session-based concurrency in Maude: Executable semantics and type checking

Carlos Alberto Ramírez Restrepo [a,*], Juan C. Jaramillo [b], Jorge A. Pérez [b,*]

[a] Pontificia Universidad Javeriana, Cali, Colombia
[b] University of Groningen, the Netherlands

A R T I C L E I N F O

Article history:
Received 30 September 2022
Received in revised form 1 March 2023
Accepted 31 March 2023
Available online 6 April 2023

Dataset link: https://gitlab.com/calrare1/session-types

A B S T R A C T

Session types are a well-established approach to communication correctness in message-passing processes. Widely studied from a process calculi perspective, here we pursue an unexplored strand and investigate the use of the Maude system for implementing session-typed process languages and reasoning about session-typed process specifications.

We present four technical contributions. First, we develop and implement in Maude an executable specification of the operational semantics of a session-typed π-calculus by Vasconcelos. Second, we also develop an executable specification of its associated algorithmic type checking, and describe how both specifications can be integrated. Third, we show that our executable specification can be coupled with reachability and model checking tools in Maude to detect well-typed but deadlocked processes. Finally, we demonstrate the robustness of our approach by adapting it to a *higher-order* session π-calculus, in which exchanged values include names but also abstractions (functions from names to processes).

All in all, our contributions define a promising new approach to the (semi)automated analysis of communication correctness in message-passing concurrency.

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

This paper presents several executable rewriting semantics for a π-calculus equipped with *session types*. Widely known as the paradigmatic calculus of interaction, the π-calculus [19,26] offers a rigorous platform for reasoning about message-passing concurrency. Session types are arguably the most prominent representative of *behavioral type systems* [14], which can statically ensure that processes respect their ascribed *interaction protocols* and never exhibit errors and mismatches.

The integration of (variants of) the π-calculus with different formulations of session types has received much attention from foundational and applied perspectives. As a result, our understanding about (abstract) communicating processes and their typing disciplines steadily reaches maturity. Despite this progress, rigorous connections with more concrete representation models fall short. In particular, the study of session-typed π-calculi within frameworks and systems like Maude [4] seems to remain unexplored. This gap is an opportunity to investigate the formal systems underlying session-typed π-calculi (reduction semantics and type systems) from a fresh yet rigorous perspective, taking advantage of the concrete representation given by executable semantics in Maude.

* Corresponding authors.
E-mail addresses: carlosalbertoramirez@javerianacali.edu.co (C.A. Ramírez Restrepo), j.a.perez@rug.nl (J.A. Pérez).

https://doi.org/10.1016/j.jlamp.2023.100872
2352-2208/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Looking at session-typed π-calculi from the perspective of Maude is insightful, for several reasons. First, Maude enables the systematic validation of such formal systems and their results, improving over pen-and-paper developments. Second, as there is not a canonical session-typed π-calculus, but actually many different formulations (with varying features and properties), an implementation in Maude could provide a concrete platform for uniformly representing them all. Third, resorting to Maude as a host representation framework for session-typed π-calculi could help in addressing known limitations of static type checking for deadlock detection, leveraging tools already available in Maude.

This paper presents our work on pursuing these directions. Our contributions are as follows:

1. We implement in Maude the session-typed π-calculus developed by Vasconcelos [30], dubbed sπ in the following. Specifically, for this typed language, we first implement its (untyped) reduction semantics as a rewriting semantics, essentially extending prior work on representing the π-calculus in Maude (see below). Then, we implement its associated algorithmic type system, also given in [30].
2. Well-typedness in [30] ensures *fidelity* (i.e., well-typed sπ processes respect at runtime their ascribed protocols) but does not rule out deadlocks and other kinds of insidious circular dependencies. This is a known shortcoming, which we address by leveraging reachability and model checking of formulas in Linear Temporal Logic (LTL) in Maude.
3. Finally, we show that our approach for implementing sπ in Maude extends also to a different session process language, namely the *higher-order* session π-calculus (HOπ), introduced and studied by Kouzapas et al. [17,18]. The typed calculus HOπ is interesting because it allows one to exchange names (as in sπ) as well as abstractions (functions from names to processes). We develop an executable semantics and type checking for HOπ in Maude, the latter based on a new algorithmic type system that we develop here.

The rest of this paper is organized as follows. Next, Section 2 summarizes the syntax and semantics of sπ. Section 3 describes the definition of our rewriting semantics for sπ in Maude. Section 4 addresses types: it presents the rewriting implementation of the algorithmic type checking system for sπ introduced in [30]. Section 5 presents our developments on deadlock detection. Section 6 extends our approach to the higher-order case: we recall the syntax of HOπ, define its rewriting semantics, develop a new algorithmic type checking system, and implement it in Maude. Section 7 discusses related works and collects some concluding remarks. Additional material has been collected in the appendices.

This paper is an extended and revised version of the WRLA'22 paper [23], extended with the treatment of higher-order session processes in Section 6, which is entirely new to this presentation. Our Maude developments are publicly available online:

https://gitlab.com/calrare1/session-types

2. A session-based π-calculus

The process calculus sπ is a variant of the synchronous π-calculus [19] with session constructs [15], formalized by Vasconcelos [30]. Here we summarize its syntax and semantics.

Syntax The calculus sπ relies on a base set of *variables*, ranged over by $x, y, \ldots$. Variables denote *channels* (or *names*). Processes interact to exchange values, which can be variables or booleans. Variables can be seen as consisting of (dual) *endpoints* on which interaction takes place. Rather than non-deterministic choices among prefixed processes, there are two complementary operators: one for offering a finite set of alternatives (called *branching*) and one for choosing one of such alternatives (*selection*). More formally, the syntax of *values*, *qualifiers*, and *processes* is presented below:

$$v ::= x \mid \mathsf{true} \mid \mathsf{false} \qquad\qquad q ::= \mathsf{un} \mid \mathsf{lin}$$

$$P ::= \mathbf{0} \quad\mid\quad \overline{x}v.P \quad\mid\quad q\,x(y).P \quad\mid\quad P_1 \mid P_2 \quad\mid\quad (\boldsymbol{\nu}xy)P$$

$$\mid \quad \mathsf{if}\ v\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \quad\mid\quad x \triangleright \{l_i : P_i\}_{i\in I} \quad\mid\quad x \triangleleft l.P$$

The inactive process is denoted as $\mathbf{0}$. The output process $\overline{x}v.P$ sends the value v along x and continues as P. The process $q\ x(y).P$ denotes an input action on x, which prefixes P. The qualifier q is used for inputs, which can be linear (to be executed exactly once) or shared. The process $\mathsf{un}\ x(y).P$ denotes a persistent input action, which corresponds to (input-guarded) replication in the π-calculus. The parallel composition $P_1 \mid P_2$ denotes the concurrent execution of P_1 and P_2. The process $(\boldsymbol{\nu}xy)P$ declares the scope of *co-variables* x and y to be P. These co-variables are intended to be the complementary ends of a communication channel. Given a boolean v, the process $\mathsf{if}\ v\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ continues as P_1 if v is true; otherwise it continues as P_2. The branching process $x \triangleright \{l_i : P_i\}_{i\in I}$ offers multiple alternative branches $P_1, P_2, \ldots$ (each with a *label* $l_1, l_2, \ldots$), along x; it is meant to interact with a selection process $x \triangleleft l.P$, which uses x to indicate the choice of the alternative labeled l and then continues as P.

As usual, $q\ x(y).P$ binds variable y in P and $(\boldsymbol{\nu}xy)P$ binds co-variables x, y in P. The sets of free and bound variables of a process P, denoted $\mathtt{fv}(P)$ and $\mathtt{bv}(P)$, are defined accordingly. Process $P[v/y]$ denotes the capture-avoiding substitution of variable y by value v in process P.

$$
\begin{array}{ll}
P \mid Q \equiv Q \mid P & P \mid \mathbf{0} \equiv P \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R & (\boldsymbol{\nu}xy)\, \mathbf{0} \equiv \mathbf{0} \\
(\boldsymbol{\nu}xy)(\boldsymbol{\nu}wz)P \equiv (\boldsymbol{\nu}wz)(\boldsymbol{\nu}xy)P & (\boldsymbol{\nu}xy)P \mid Q \equiv (\boldsymbol{\nu}xy)(P \mid Q) \quad \text{If } x, y \notin \mathtt{fv}(Q) \\
\text{if true then } P_1 \text{ else } P_2 \equiv P_1 & \text{if false then } P_1 \text{ else } P_2 \equiv P_2
\end{array}
$$

Fig. 1. Structural congruence Rules for sπ.

$$\frac{}{(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \text{lin } y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid R)}\ [\text{R-LinCom}]$$

$$\frac{}{(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \text{un } y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid \text{un } y(z).Q \mid R)}\ [\text{R-UnCom}]$$

$$\frac{j \in I}{(\boldsymbol{\nu}xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q_j \mid R)}\ [\text{R-Case}]$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \frac{P \longrightarrow P'}{(\boldsymbol{\nu}xy)P \longrightarrow (\boldsymbol{\nu}xy)P'}\ [\text{R-Par}]\ [\text{R-Res}]$$

$$\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}\ [\text{R-Struct}]$$

Fig. 2. Reduction semantics for sπ.

Semantics The operational semantics for sπ is given as a *reduction semantics*, which, as customary, relies on a structural congruence relation, the smallest congruence relation on processes that satisfy the axioms in Fig. 1. Structural congruence includes the usual axioms for inaction and parallel composition as well as adapted axioms for scope restriction, scope extrusion, and conditionals.

Armed with structural congruence, the rules of the reduction semantics are presented in Fig. 2. Rules [R-LinCom] and [R-UnCom] induce different patterns for process communication, depending on the qualifier of their corresponding input action. Indeed, processes $\overline{x}v.P$ and $q\ y(z).Q$ can synchronize if x and y are co-variables. This is only possible if both processes are underneath a scope restriction $(\boldsymbol{\nu}xy)$. When this occurs, processes $\overline{x}v.P$ and $q\ y(z).Q$ continue respectively as P and $Q[v/z]$. When $q = \text{un}$ then process $q\ y(z).Q$ remains (Rule [R-UnCom]); otherwise, process $q\ y(z).Q$ disappears (Rule [R-LinCom]). Rule [R-Case] stands for the case synchronization: processes $x \triangleleft l_j.P$ and $y \triangleright \{l_i : Q_i\}_{i\in I}$ can synchronize if they are underneath a scope restriction $(\boldsymbol{\nu}xy)$. Process $x \triangleleft l_j.P$ reduces to process P and process $y \triangleright \{l_i : Q_i\}_{i\in I}$ reduces to process Q_j. Rules for parallel composition, scope restriction, and structurally congruent processes (Rules [R-Par], [R-Res], [R-Struct]) are as usual.

Example 2.1. Consider the process $P = (\boldsymbol{\nu}x_1 y_1)(\boldsymbol{\nu}x_2 y_2)(P_1 \mid P_2 \mid P_3)$ where:

$$P_1 = \text{un } y_1(t).\overline{t}\,\text{false}.\mathbf{0}$$

$$P_2 = \text{lin } y_1(w).\overline{w}\,\text{true}.\mathbf{0}$$

$$P_3 = \overline{x_1}x_2.\text{lin } y_2(z).\overline{a}z.\mathbf{0}$$

Starting from P, there are two possible sequences of reductions depending on the processes involved in the initial synchronization in the co-variables x_1, y_1.

- Suppose P_1 and P_3 synchronize via an application of rule [R-UnCom] over co-variables x_1, y_1 and thereafter rule [R-LinCom] is used over x_2, y_2. Then we would have:

$$P \longrightarrow\longrightarrow (\boldsymbol{\nu}x_1 y_1)(\boldsymbol{\nu}x_2 y_2)(P_1 \mid P_2 \mid \overline{a}\text{false}.\mathbf{0})$$

- On the other hand, if P_2 and P_3 synchronize then rule [R-LinCom] can be applied twice, first over co-variables x_1, y_1 and after over x_2, y_2. In this case, we would have:

$$P \longrightarrow\longrightarrow (\boldsymbol{\nu}x_1 y_1)(\boldsymbol{\nu}x_2 y_2)(P_1 \mid \overline{a}\text{true}.\mathbf{0})$$

The *standard form* of a process, defined in [30], will be crucial for the executable specification of the reduction semantics. We say P is in standard form if it matches the pattern expression $(\boldsymbol{\nu}x_1 y_1)(\boldsymbol{\nu}x_2 y_2)\ldots(\boldsymbol{\nu}x_n y_n)(P_1 \mid P_2 \mid \ldots \mid P_k)$, where each P_i is a process of the form $\overline{x}v.Q$, $q\ x(y).Q$, $x \triangleleft l.Q$ or $x \triangleright \{l_i : Q_i\}_{i\in I}$. Every process is structurally congruent to a process in standard form.

3. Rewriting semantics for sπ

3.1. Syntax

Our rewriting semantics for sπ adapts the one by Thati et al. [28], which is defined for an untyped π-calculus without sessions. There is a direct correspondence between the syntactic categories (values, variables, qualifiers, and terms) and Maude sorts (Value, Chan, Qualifier, and Trm, respectively). We also have some auxiliary sorts such as Guard, Choice, and Choiceset.

```
sorts Value Chan Qualifier Trm Guard Choice Choiceset .
subsort Choice < Choiceset .
subsort Chan < Value .

op _{_} : Qid Nat -> Chan [prec 1] .
ops lin un : -> Qualifier [ctor] .
ops True False : -> Value [ctor] .
op  __(_) : Qualifier Chan Qid -> Guard [ctor prec 5] .
op _<_> : Chan Value -> Guard [ctor prec 6] .
op nil : -> Trm [ctor] .
op new[__]_ : Qid Qid Trm -> Trm [ctor prec 10] .
op _|_  :  Trm Trm -> Trm [ctor assoc comm prec 12 id: nil] .
op if_then_else_fi : Value Trm Trm -> Trm [ctor prec 8] .
op _ << _._ : Chan Qid Trm -> Trm [ctor prec 15] .
op _ >> {_} : Chan Choiceset -> Trm [ctor prec 17] .
op  _._ : Guard Trm -> Trm [ctor prec 7] .
op _:_ : Qid Trm -> Choice .
op empty : -> Choiceset [ctor] .
op __ : Choiceset Choiceset -> Choiceset [ctor assoc comm id: empty] .
```

Following the syntax in Section 2, values can be variables or booleans. We represent booleans as the constructors True and False whereas we distinguish variables (sort Chan) as values through the subsort relation. The only constructor for variables _{_} takes a Qid and a natural number. Each production rule for processes is represented using a constructor, as expected. Notice that the constructor for input guards ___(_) is preceded by a qualifier. Process **0** is denoted as nil and a single guarded term is represented by the constructor _._. The constructor for scope restriction new[__]_ uses two instances of Qid, since it declares a pair of co-variables. The constructor for conditionals is parametric on an instance of Value. We add constructors for selection and branching process terms; their definition is as expected. In particular, the constructor for branching processes relies on instances of Choiceset, which consists of sets of pairs of Qid and process terms. We use instances of Qid to represent labels.

Substitutions As we have seen, the semantics of sπ relies on substitutions of variables with values. To deal with substitutions in Maude, we follow Thati et al.'s approach [28] and use Stehr's CINNI calculus [25], an explicit substitution calculus, which provides a mechanism to implement α-conversion at the language level. The idea behind CINNI is to syntactically associate each use of a variable x to an index, which acts as a counter of the number of binders for x that are found before it is used. In CINNI, there are three types of substitution operations:

Type	Meaning
Simple substitution	$[\mathtt{a} := \mathtt{x}]\ \mathtt{a\{0\}} \mapsto \mathtt{x}$ $\quad [\mathtt{a} := \mathtt{x}]\ \mathtt{a\{n+1\}} \mapsto \mathtt{a\{n\}}$ $[\mathtt{a} := \mathtt{x}]\ \mathtt{b\{m\}} \mapsto \mathtt{b\{m\}}$
Shift substitution	$\uparrow_{\mathtt{a}} \mathtt{a\{n\}} \mapsto \mathtt{a\{n+1\}}$ $\quad \uparrow_{\mathtt{a}} \mathtt{b\{m\}} \mapsto \mathtt{b\{m\}}$
Lift substitution	$\Uparrow_{\mathtt{a}} (\mathtt{S})\ \mathtt{a\{0\}} \mapsto \mathtt{a\{0\}}$ $\quad \Uparrow_{\mathtt{a}} (\mathtt{S})\ \mathtt{a\{n+1\}} \mapsto \uparrow_{\mathtt{a}} (\mathtt{S}\ \mathtt{a\{n\}})$ $\Uparrow_{\mathtt{a}} (\mathtt{S})\ \mathtt{b\{m\}} \mapsto \uparrow_{\mathtt{a}} (\mathtt{S}\ \mathtt{b\{m\}})$

A simple substitution of a variable a for a variable x takes place if the index of x is 0; the index is decreased by 1 otherwise. A shift substitution over a increases by 1 the index and a substitution S can be lifted to skip one index. Any substitution over a variable a has no effect on other variables.

We now present the definition of explicit substitutions for sπ using an approach similar to Stehr's. We first present the definition of the variable substitutions. We use the sort Subst and the substitution application is performed by the operator ___, which takes a substitution and a variable. We define equations for the three substitutions above:

```
sort Subst .
op [_:=_] : Qid Value -> Subst .
op [shiftup_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Chan -> Chan .
```

```
eq  [ a := v ] a{0} = v .
eq  [ a := v ] a{s(n)} = a{n} .
ceq [ a := v ] b{n} = b{n}  if a =/= b .
eq  [ a := v ] val(a) = v .
eq  [ shiftup a ] a{n} = a{s(n)} .
ceq [ shiftup a ] b{n} = b{n}  if a =/= b .
eq  [ lift a S ] a{0} = a{0} .
eq  [ lift a S ] a{s(n)} = [ shiftup a ] S a{n} .
ceq [ lift a S ] b{n} = [ shiftup a ] S b{n} if a =/= b .
```

Equipped with these elements, we adapt to the syntax of sπ the equations associated to the explicit substitutions for the process terms as follows:

```
op  __ :  Subst Trm -> Trm [prec 3] .
op  subst-aux : Subst Choiceset -> Choiceset .
eq  S nil = nil .
eq  S (new [x y] P) = new [x y] ([lift x S] [lift y S] P) .
eq  S (q a(y) . P ) = q (S a)(y) . ([lift y S] P) .
eq  S (a < b > . P) = (S a) < (S b) > . (S P) .
ceq S (a < v > . P) = (S a) < v > . (S P) if v == True or v == False .
ceq S (if v then P else Q fi) = if v then (S P) else (S Q) fi
                               if v == True or v == False .
eq  S (if val(x) then P else Q fi) = if (S val(x)) then (S P) else (S Q) fi .
eq  S (a >> {CH}) = (S a) >> { subst-aux(S, CH) } .
eq  S (a << x . P) = (S a) << x . (S P) .
eq  S (P | Q) = (S P) | (S Q) .
eq  subst-aux(S, empty) = empty .
eq  subst-aux(S, (x : P) CH) = (x : (S P)) subst-aux(S, CH) .
```

In each equation, we deal with a specific production rule for process terms. In each process, the substitution `S` is applied in each variable and each subprocess, as expected. In particular, a lift substitution is performed over `x`, `y`, and `S` to skip the index 0 and perform the substitution in the remaining indices for the scope restriction operator. In this way, the substitution `S` has the expected effect.

Structural congruence To represent the rules in Fig. 1, we exploit the Maude equational attributes `assoc`, `comm`, and `id` to declare the associative, commutative, and identity axioms for parallel composition, with process `nil` acting as its identity. This suffices to cover the rules on the two first lines of Fig. 1. The remaining rules are explicitly declared via the equations below:

```
eq new[x y] nil = nil .
eq if True then P else Q fi = P .
eq if False then P else Q fi = Q .
ceq P | new[x y] Q = new [x y] (Q | [shiftup x] [shiftup y] P)
    if P =/= nil /\ Q =/= nil /\ CS := free\mathit{vars}(P) /\ x{0} in CS and y{0} in CS .
ceq P | new[x y] Q = new [x y] (Q | [shiftup x] P)
    if P =/= nil /\ Q =/= nil /\ CS := free\mathit{vars}(P) /\ x{0} in CS and not y{0} in CS .
ceq P | new[x y] Q = new [x y] (Q | [shiftup y] P)
    if P =/= nil /\ Q =/= nil /\ CS := free\mathit{vars}(P) /\ not x{0} in CS and y{0} in CS .
ceq P | new[x y] Q = new [x y] (Q | P)
    if P =/= nil /\ Q =/= nil /\ CS := free\mathit{vars}(P) /\ not x{0} in CS /\ not y{0} in CS .
```

Scope extrusion is represented through four equations corresponding to the four cases in the presence of `x`, `y` in the free variables of process `P`. Function `freevars` stands for the Maude implementation of the function `fv` over processes.

3.2. Operational semantics

Combined, the Maude rewriting rules, the equational attributes, and the explicit equations associated to variables of sort `Trm` can appropriately express the reduction semantics of sπ and manipulate terms in a compositional fashion. A process is reduced to a simpler equivalent form by virtue of the equational theory; a process is rewritten as long as it satisfies the structure required for a rule wherever the process is located. As a consequence, subprocesses are also rewritten and we do not need to explicitly represent the contextual rules ([R-Par] and [R-Res]).

A process is converted into standard form using the explicit congruence rules. This way, the scope of every unguarded occurrence of the `new` operator is extended to the top level.

Process interaction in sπ can only occur through co-variables; therefore, processes involved in a synchronization must be underneath a scope restriction over such co-variables. Nonetheless, since in the standard form the order of the unguarded occurrences of the `new` operator is irrelevant, it would be necessary to explicitly look for the processes that are enabled to interact, which would affect the efficiency of the rewriting specification. To counter this, we include an auxiliary operator,

dubbed `new*`, which declares a list of pairs of new co-variables, rather than just a single pair. This is equivalent to using nested `new` operators, i.e., the term `new* [x1 y1 x2 y2 ... xn yn] P` is equivalent to the term

`new [x1 y1] new [x2 y2] ... new [xn yn] P.`

We declare the constructor for the sort `QidSet` with the equational attribute `comm` to ensure that the order among the pairs of new co-variables is irrelevant.

This way, the entire process will be underneath a scope restriction `new*` and interaction will be poss

```
sorts QidPair QidSet . subsort QidPair < QidSet .
op __ : Qid Qid -> QidPair [ctor] .
op mt : -> QidSet [ctor] .
op __ : QidSet QidSet -> QidSet [ctor comm assoc id: mt] .
op new* [_] _ : QidSet Trm -> Trm [ctor] .
```

Given a process P, let us write $[\![P]\!]$ to denote its representation in Maude. A reduction rule $P \longrightarrow Q$ can be associated to a rewriting rule $l : [\![P]\!]$ => $[\![Q]\!]$. The reduction rules can be stated as follows:

```
crl [FLAT] : P => P' if P' := flatten(P) /\ P =/= P' .
rl [LINCOM] : new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R =>
              new* [(x y) nl] P | [z := v] Q | R .
rl [UNCOM] : new* [(x y) nl] x{N} < v > . P | un y{N}(z) . Q | R =>
             new* [(x y) nl] P | [z := v] Q | un y{N}(z) . Q | R .
rl [CASE] : new* [(x y) nl] (x{N} << w . P) | (y{N} >> { (w : Q) CH }) | R =>
            new* [(x y) nl] P | Q | R .
```

Rule `FLAT` normalizes the whole process. Hence, in addition to the implicit rewriting performed by the equations associated to the congruence rules, the nested `new` declarations are stated as a flat declaration `new*`. We use an auxiliary operation `flatten`, which is defined as follows:

```
op flatten : Trm -> Trm .
eq flatten(new [x y] P) = flatten(new* [x y] P) .
eq flatten(new* [nl] new [x y] P) = flatten(new* [nl x y] P) .
eq flatten(new* [nl] new* [nl'] P) = flatten(new* [nl nl'] P) .
eq flatten(P) = P [owise] .
```

Rules `LINCOM`, `UNCOM` and `CASE` correspond to the specification of the reduction rules related to synchronization in the calculus semantics (see Fig. 2). In these rules, `nl` stands for the additional co-variables being declared. Rules `LINCOM` and `UNCOM` substitute the variable `z` for the value `v`. It is worth to highlight that these rules require the process term is underneath an operator `new*` whereby synchronization is only possible when the communication pattern occurs at the top level of the process and in this way we prevent the rewriting of guarded subprocesses.

We also include some equations which capture natural equivalences for processes involving the auxiliary operator `new*`.

```
eq new* [nl] nil = nil .
eq new* [x y nl] y{N} < v > . P | q x{N}(z) . Q | R =
   new* [y x nl] y{N} < v > . P | q x{N}(z) . Q | R .
eq new* [x y nl] (y{N} << w . P) | (x{N} >> { CH }) | R =
   new* [y x nl] (y{N} << w . P) | (x{N} >> { CH }) | R .
```

Given a pair of co-variables `x y`, we assume that the first action on `x` is an output or a selection and the first action on `y` is an input or a branching. When this is not the case, the last two equations swap `x` and `y` to enable the execution of the rewriting rules. The type system detailed in Section 4 ensures that for typable process this swap occurs at most once in a pair of co-variables.

Our rewriting specification enables us to directly execute a possible sequence of reductions over a process using the Maude command '`rew`'. In this way, we can obtain a stable (final) reachable process, which cannot reduce further. Moreover, we can use the reachability command '`search`' to: (i) perform all possible sequence of reductions of a process and obtain every possible stable process and (ii) check whether a process that fits some pattern is reachable or if a specific process is reachable. In Section 4, we leverage commands '`search`' and '`modelCheck`' to detect deadlocked $s\pi$ processes.

Specification correctness The transition system associated to our rewrite theory in Maude can be shown to coincide with the reduction semantics in Section 2. This operational correspondence result is detailed in Appendix A.

4. Algorithmic type checking for sπ

4.1. Type syntax

We present a Maude implementation of the algorithmic type checking given in [30]. The type system considers *typing contexts*, denoted Γ, which associate each variable to a specific type, denoted T. Typing contexts and types are defined inductively as follows:

$$\Gamma ::= \emptyset \mid \Gamma, x : T \qquad q ::= \mathsf{lin} \mid \mathsf{un}$$

$$p ::= ?T.T \mid !T.T \mid \&\{l_i : T_i\}_{i\in I} \mid \oplus\{l_i : T_i\}_{i\in I}$$

$$T ::= \mathsf{bool} \mid \mathsf{end} \mid q\,p \mid a \mid \mu a.T$$

Above, q stands for qualifiers and p stands for pretypes; Moreover, x denotes a variable, each l_i denotes a label and a denotes a general variable. For simplicity, we assume a single basic type for values (bool). For a given type T, we also use the predicate un(T), which holds whenever T= *bool* or T= un p, for a pretype p.

Each variable is associated to a (session) type, which represents its intended protocol. In the above grammar, these types correspond to qualified pretypes. The pretype $?T_1.T_2$ (resp. $!T_1.T_2$) is assigned to a variable that first receives (resp. sends) a value of type T_1 and then proceeds to type T_2. The pretype $\&\{l_i : T_i\}_{i\in I}$ (resp. $\oplus\{l_i : T_i\}_{i\in I}$) is assigned to a variable that can offer (resp. select) l_i options and continues with type T_i depending on the label selected. The type end (empty sequence) denotes the type of a variable where no interaction can occur. Recursive types can express infinite sequences of actions; in the type $\mu a.T$, a corresponds to a type variable that must occur guarded in T. Following [30], type equality is defined equi-recursively; thus, instead of dealing directly with recursive types of the form $\mu a.T$, an equivalent type is taken. *Unfolding* is the mechanism used to obtain the new type, where substitutions $T[\mu a.T/a]$ are performed until the type does not start with the binder 'μa'.

We encode session types in Maude by associating the non-terminals context, qualifiers, pretypes, and types to sorts Context, Qualifier, Pretype, and Type.

```
sorts Pretype Type Context ChoiceT ChoiceTset .
subsort ChoiceT < ChoiceTset .
op ?_._ : Type Type -> Pretype .  op !_._ : Type Type -> Pretype .
op +{_} : ChoiceTset -> Pretype . op &{_} : ChoiceTset -> Pretype .
ops bool end : -> Type .          op __ : Qualifier Pretype -> Type .
op u [_] _ : Qid Type -> Type .   op var : Qid -> Type .
ops nil invalid-context : -> Context .
op _:_ : Value Type -> Context .
op _,_ : Context Context -> Context [ctor assoc comm id: nil] .
op _:_ : Qid Type -> ChoiceT . op empty : -> ChoiceTset .
op __ : ChoiceTset ChoiceTset -> ChoiceTset [assoc comm id: empty] .
```

Each production rule is given as a specific constructor. In particular, constructors +{_} and &{_} represent the pretypes $\oplus\{l_i : T_i\}_{i\in I}$ and $\&\{l_i : T_i\}_{i\in I}$, respectively. The pairs of labels l_i and subtypes T_i are defined as instances of the sort ChoiceTset. The recursive type $\mu a.T$ is given as the constructor u [_] _ and the type variables are given as the constructor var.

Typing contexts are defined as expected. An empty context is denoted as nil whereas a single context is associated to the constructor _:_. General contexts are provided by the constructor _,_, which is annotated with the equational attributes assoc, comm and id since the order is irrelevant in typing contexts and the construction is associative. Finally, we added a constant invalid-context to denote an error in type checking.

4.2. Algorithmic type checking

We follow the system for algorithmic type checking proposed (and proven correct) in [30]. This type system enables type checking of the sπ processes from Section 2, with a minor caveat: algorithmic type checking uses processes in which the restriction operator has a corresponding type annotation, i.e., it uses $(\boldsymbol{\nu}xy : T)P$ instead of $(\boldsymbol{\nu}xy)P$. Consequently, we add a constructor for the sort Trm in the Maude specification:

```
op  new[__:_]_ : Qid Qid Type Trm -> Trm [ctor prec 28] .
```

Following [30], we implement the type checking algorithm by relying on some auxiliary functions for type duality (i.e., compatibility), type equality, and context update and difference, among others. They are implemented by means of functions and equations in Maude. The details of the Maude implementation for type duality (function mclo-dual), context update (function +), and the context difference (function \) can be found in Appendix B.

$$\Gamma \vdash \text{true} : \text{bool}; \Gamma \quad [\text{A-True}] \qquad \Gamma_1, x : \text{lin } p, \Gamma_2 \vdash x : \text{lin } p; (\Gamma_1, \Gamma_2) \quad [\text{A-LinVar}]$$

$$\Gamma \vdash \text{false} : \text{bool}; \Gamma \quad [\text{A-False}] \qquad \frac{\text{un}(T)}{\Gamma_1, x : T, \Gamma_2 \vdash x : T; (\Gamma_1, x : T, \Gamma_2)} \quad [\text{A-UnVar}]$$

Fig. 3. Typing rules for values, $\Gamma \vdash v : T; \Gamma$.

$$\Gamma \vdash \mathbf{0} : \Gamma; \emptyset \qquad \frac{\Gamma_1 \vdash P : \Gamma_2; L_1 \qquad \Gamma_2 \div L_1 \vdash Q : \Gamma_3; L_2}{\Gamma_1 \vdash P \mid Q : \Gamma_3; L_2} \qquad [\text{A-Inact}]\ [\text{A-Par}]$$

$$\frac{\Gamma_1, x : T, y : \overline{T} \vdash P : \Gamma_2; L}{\Gamma_1 \vdash (\boldsymbol{\nu} xy : T)\ P : \Gamma_2 \div \{x, y\}; L \backslash \{x, y\}} \quad [\text{A-Res}]$$

$$\frac{\Gamma_1 \vdash v : q \text{ bool}; \Gamma_2 \qquad \Gamma_2 \vdash P : \Gamma_3; L \qquad \Gamma_2 \vdash Q : \Gamma_3; L}{\Gamma_1 \vdash \text{if } v \text{ then } P \text{ else } Q : \Gamma_3; L} \quad [\text{A-If}]$$

$$\frac{\Gamma_1 \vdash x : q!T.U; \Gamma_2 \qquad \Gamma_2 \vdash v : T; \Gamma_3 \qquad \Gamma_3 + x : U \vdash P : \Gamma_4; L}{\Gamma_1 \vdash \overline{x}v.P : \Gamma_4; L \cup (\text{if } q = \text{lin then } \{x\} \text{ else } \emptyset)} \quad [\text{A-Out}]$$

$$\frac{\Gamma_1 \vdash x : q_2?T.U; \Gamma_2 \quad (\Gamma_2, y : T) + x : U \vdash P : \Gamma_3; L \quad q_1 = \text{un} \Rightarrow L \backslash \{y\} = \emptyset}{\Gamma_1 \vdash q_1 x(y).P : \Gamma_3 \div \{y\}; L \backslash \{y\} \cup (\text{if } q_2 = \text{lin then } \{x\} \text{ else } \emptyset)} \quad [\text{A-In}]$$

$$\frac{\Gamma_1 \vdash x : q\&\{l_i : T_i\}_{i \in I}; \Gamma_2 \quad \Gamma_2 + x : T_i \vdash P_i : \Gamma_3; L_i \quad \forall_{i \in I, j \in I}\ L_i \backslash \{x\} = L_j \backslash \{x\}}{\Gamma_1 \vdash x \triangleright \{l_i : P_i\}_{i \in I} : \Gamma_3; L \cup (\text{if } q = \text{lin then } \{x\} \text{ else } \emptyset)} \quad [\text{A-Branch}]$$

$$\frac{\Gamma_1 \vdash x : q \oplus \{l_i : T_i\}_{i \in I}; \Gamma_2 \qquad \Gamma_2 + x : T_j \vdash P : \Gamma_3; L \qquad j \in I}{\Gamma_1 \vdash x \triangleleft l_j.P : \Gamma_3; L \cup (\text{if } q = \text{lin then } \{x\} \text{ else } \emptyset)} \quad [\text{A-Sel}]$$

Fig. 4. Typing rules for processes, $\Gamma \vdash P : \Gamma; L$.

Algorithmic type checking is expressed by sequents of the form $\Gamma_1 \vdash v : T; \Gamma_2$ (for values) and $\Gamma_1 \vdash P : \Gamma_2; L$ (for processes). These sequents have an input-output reading: sequent $\Gamma_1 \vdash v : T; \Gamma_2$ denotes an algorithm that takes Γ_1 and v as input and returns T and Γ_2 as output; similarly, sequent $\Gamma_1 \vdash P : \Gamma_2; L$ denotes an algorithm that takes Γ_1 and P as input and produces Γ_2 and L as output. While Γ_2 is a residual context, the set L collects linear variables occurring in subject position. Intuitively, L tracks the linear variables that are used in P to prevent their reuse in another process. Both algorithms are given by typing rules, which we specify in Maude as an equational theory.

Fig. 3 shows the typing rules for values, which correspond to the rules in [30]. The rules for boolean values (Rules [A-True] and [A-False]) produce as results the type bool and the input context Γ remains unaltered. There are two rules for a variable x: if x has a linear type lin p then the entry $x : \text{lin } p$ is removed from the returned context (Rule [A-LinVar]); otherwise, if $x : \text{un}\, p$ for a pretype p, then the entry $x : \text{un}\, p$ is kept in the returned context (Rule [A-UnVar]). The algorithm for type checking of values is then implemented as a function `type-value`, which is defined as follows:

```
op type-value : Context Value -> TupleTypeContext .
eq type-value(C, True) = [C bool] .                       ---[A-TRUE]
eq type-value(C, False) = [C bool] .                      ---[A-FALSE]
ceq type-value(((a : T), C), a) = [((a : T), C) unfold(T)] ---[A-UNVAR]
    if unrestricted(T) .
eq type-value(((a : lin p), C), a) = [C (lin p)] .        ---[A-LINVAR]
eq type-value(((a : u [x] T), C), a) =
   type-value(((a : unfold(u [x] T)), C), a) .            ---[A-LINVAR]
eq type-value(C, v) = ill-typed [owise] .
```

Function `type-value` produces an instance of the sort `TupleTypeContext`. This sort groups a context and a type or a set of variables and it has only one constructor `[_ _]`. The equations related to the typing of boolean values arise as expected, according to the corresponding typing rule. In those cases, a tuple that contains the unmodified context and the type `bool` is produced. For unrestricted variables, given that some types are infinite, the unrestricted types are *unfolded* before the update (cf. the `unfold` operation). If a type T is a recursive type $\mu a.U$ then the substitution $U[\mu a.U/a]$ is performed. Otherwise, the type T remains unaltered. For linear variables, we also unfold the type when necessary and the linear type is returned and removed from the context.

Fig. 4 shows the typing rules for sπ processes; they largely correspond to the rules in [30]. Rule [A-Inact] proceeds as expected. Process $\mathbf{0}$ is well-typed and the typing context Γ remains unaltered and the set of linear variables is empty. Rule [A-Par] handles parallel composition: to check a process $P \mid Q$ over a context Γ_1, the type of P is checked and the resulting context Γ_2 is used to type-check process Q, making sure that the linear variables used for P are first removed by using the context difference function ($\Gamma_2 \div L_1$). This ensures that free linear variables are used only once. The output

of the algorithm for Q (context Γ_3 and set L_2) then corresponds to the output of the entire process $P \mid Q$. Rule [A-Res] type-checks a process $(\boldsymbol{\nu}xy : T)P$ in a context Γ_1: it first checks the type of sub-process P in the context Γ_1 extended with the association of variables x, y to the type T and its dual type, denoted $\overline{T}$. It is expected that if type T ($\overline{T}$) is linear then it should not be in the resulting context Γ_2; otherwise, if type T ($\overline{T}$) is unrestricted then it will appear in Γ_2. We require that variables x, y are deleted from the residual context ($\Gamma_2 \div \{x, y\}$) and from the set L of linear variables.

Rule [A-If] verifies that type of value v is bool in the context Γ_1, and requires that the typecheck of P and Q in the context Γ_2 generate the same residual context Γ_3 and the same set L, since both processes should use the same linear variables.

Rule [A-Out] handles output processes: it uses the incoming context Γ_1 to check the type of x, which should be of the form $q!T.U$. Then, it checks that the type of v in the residual context Γ_2 is T. The type of the continuation P is checked in the context $\Gamma_3 + x : U$, which stands for the context Γ_3 *updated* with the association of x and the continuation type U. The context update '+' operator enforces that types $q!T.U$ and U must be equivalent when x is unrestricted (i.e., $q = \mathsf{un}$). The rule returns a context Γ_4 and a set of variables L joined with x, if linear. Rule [A-In] presents some minor modifications with respect to the one in [30]. We require that in the case of replication there are no (free) subjects on linear variables in process P except possibly the input variable y. Other than this, this rule is similar to Rule [A-Out].

Rule [A-Sel] looks the type of x in the incoming context Γ_1. This type must be of the form $q \oplus \{l_i : T_i\}_{i\in I}$. Subsequently, the continuation P is type-checked under the resulting context Γ_2 updated with a new assumption for x, which is associated to a type T_j. In this way, when $q = \mathsf{un}$ we must have $\oplus\{l_i : T_i\}_{i\in I} = T_j$. This rule produces as result the context Γ_3 and the set of linear variables L is augmented with x if linear. Context Γ_3 and set L also corresponds to the output of the type checking of process P. Finally, we have Rule [A-Branch], which has some minor modifications with respect to the rule in [30]. More precisely, this rule has been adjusted to require that the sets of (free) subjects on linear variables L_i only differ in the input variable y. Other than this, this rule is quite similar to Rule [A-Sel].

Example 4.1. As an example of type checking, if $T = \mathsf{lin}\ !\mathsf{bool}.\mathsf{lin}\ ?\mathsf{bool}.\mathsf{end}$ then we can establish the following sequent:

$$a : \mathsf{bool} \vdash (\boldsymbol{\nu}x_1 y_1 : T)(\mathsf{lin}\ y_1(v).\overline{y_1}v.\mathbf{0} \mid \overline{x_1}a.\mathsf{lin}\ x_1(z).\mathbf{0}) : (a : \mathsf{bool}); \{x_1, y_1\}$$

We implement the type-checking algorithm as a function `type-term` that receives an instance of the sort `Context` and an instance of the sort `Trm`. Moreover, it produces an instance of the sort `TupleTypeContext` that groups the resulting typing context and the set L of linear variables that were collected during type-checking. Each rule is implemented by an equation:

```
op type-term : Context Trm -> TupleTypeContext .
eq type-term(C, nil) = [C mt] .                                             --- [A-INACT]
ceq type-term(C, P | Q) = [C2 L2]                                           --- [A-PAR]
    if [C1 L1] := type-term(C, P) /\ [C2 L2] := type-term(C1 / L1, Q) .
ceq type-term(C, new [x y : T] P) =                                         --- [A-RES]
      [(C1 / (x{0} y{0})) remove(remove(L1, x{0}), y{0})]
  if [C1 L1] := type-term((C, (x{0} : T), (y{0} : mclo-dual(T))), P) .
ceq type-term(C, if v then P else Q fi) = [C2 L1]                           --- [A-IF]
  if [C1 bool] := type-value(C, v) /\
     [C2 L1] := type-term(C1, P) /\ [C2 L1] := type-term(C1, Q) .
ceq type-term(C, a < v > . P) =                                             --- [A-OUT]
         [C3 (if q == lin then (L1 a) else L1 fi)]
    if [C1 (q ! T . U)] := type-value(C, a) /\
       [C2 T']  := type-value(C1, v) /\ /\ equal(T, T')
       [C3 L1] := type-term((C2 + a : U), P) .
ceq type-term(C, un a(y) . P) = [(C2 / y{0}) mt]                            --- [A-IN]
    if [C1 (un ? T . U)] := type-value(C, a) /\
       [C2 L] := type-term((C1, (y{0} : T)) + a : U, P) /\
       remove(L, y{0}) == mt .
ceq type-term(C, lin a(y) . P) = [(C2 / y{0}) (remove(L, y{0})              --- [A-IN]
                                  (if q == lin then a else mt fi))]
    if [C1 (q ? T . U)] := type-value(C, a) /\
       [C2 L] := type-term((C1, (y{0} : T)) + a : U, P) .
ceq type-term(C, a>>{CH}) = check-branch(C1, a, CH, CHT,q)                  --- [A-BRANCH]
    if [C1 (q & { CHT })] := type-value(C, a) .
ceq type-term(C, a << x . P) =                                              --- [A-SEL]
        [C2 (if q == lin then (L1 a) else L1 fi)]
    if [C1 (q + { (x : T) CHT })] := type-value(C, a) /\
       [C2 L1] := type-term((C1 + a : T), P) .
eq type-term(C, P) = ill-typed [owise] .
```

When type checking is successful, function `type-term` produces an outgoing type context and a set of variables. Those elements are grouped using the constructor `[__]`, which is associated to the sort `TupleTypeContext`. We use a Maude comment to annotate each equation with the corespondent typing rule. The correspondence is quite intuitive; we highlight

some important details. An empty set of variables is represented with the constant `mt`. We remark that the operator `/` stands for the context difference operation that removes some variables of a type context, whereas operator '`remove`' drops a variable of a variable set. In the equation for Rule [A-OUT], we do not use the same variable `T` in the type associated to variable `a` and the type associated to value `v` as it would be expected, since the types are possibly infinite and there are many possible representations for the same infinite type. Instead, we use another variable `T'` and we check that `T` and `T'` are equivalent, using function `equal`.

We split Rule [A-IN] in two different equations for linear and unrestricted inputs. In the linear case, it is possible that the type of the subject `a` is linear or unrestricted; when the variable is linear it must be included in the returned set of linear variables. In the unrestricted case, the type of subject `a` is required to be unrestricted in as much as the attempt to use a linear variable in an unrestricted fashion must be rejected. Moreover, we require that the only free linear variable used in process `P` is `y{0}` (condition `remove(L, y{0}) == mt`).

Example 4.2. The process

$$(\boldsymbol{\nu} x_1 x_2)(\overline{x_1} x_1.\mathbf{0} \mid \mathsf{un}\, x_2(z).(P \mid \overline{x_1} z.\mathbf{0}))$$

defines P as a replicated process (see [30]). For simplicity, we will assume $P = \overline{w}\,\mathsf{true}.\mathbf{0}$. Additionally, we assume $w : \mu b.\mathsf{un}\ !\mathsf{bool}.b$ and $x_1 : \mu a.\mathsf{un}\ !a.a$. The Maude implementation of this process is as follows:

```
op Replication : Trm .
ops TypeP Typex1 : -> Type .
eq TypeP = u['b']un ! bool . var('b')) .
eq Typex1 = u['a'] un ! var('a') . var('a')) .
Replication = new['x1' 'x2' : Typex1]
               ('x1'{0}< 'x1'{0} > . nil | un 'x2'{0}('z') . ('P'{0}< True > . nil |
                'x1'{0}< 'z'{0} > . nil))) .
```

Using the function `type-term` we obtain the following result for type-checking:

```
reduce in TEST : type-term('w'{0} : TypeP, Replication) .
rewrites: 174 in 0~ms cpu (0~ms real) (~ rewrites/second)
result TupleTypeContext: [('w'{0} : un ! bool . u['b']un ! bool . var('b')) (mt).Chanset]
```

As output we obtain a term of sort `TupleTypeContext` whose first component has no linear types and there are no linear channels in subject position (`(mt).Chanset`), which tells us that the process is well typed.

4.3. Type soundness, now executable in Maude

Vasconcelos [30] established that the type system for sπ is *sound*: a closed, well-typed process is guaranteed to have a well-defined behavior according to the ascribed protocols and the reduction semantics of the calculus. Also, the algorithmic type checking, as implemented in this section, is proven correct. With these two elements, we can integrate (i) the rewriting specification of the operational semantics and (ii) the implementation of the algorithmic type checking. This way, we only execute well-typed processes. For this purpose, we use two auxiliary functions `well-typed` and `erase`. The former checks whether a process does not have typing errors:

```
op well-typed : Trm -> Bool .
eq well-typed(P) = (type-term(nil, P) =/= ill-typed) .
```

Function `well-typed` applies the type checking algorithm `type-term` over a process `P` and returns `true` when type-checking is successful, i.e. when the result is not `ill-typed`. Function `erase` proceeds inductively on the structure of a process; when it reaches an annotated subprocess '`new [x y : T] P`', it removes the annotation to produce '`new [x y] P`'—see Appendix B for details.

Correspondingly, we extend our specification of the reduction semantics to enable the execution of annotated processes, i.e., processes that use the operator $(\boldsymbol{\nu} xy : T)P$ instead of the operator $(\boldsymbol{\nu} xy)P$:

```
rl [TYPED] : new [x y : T] P => if well-typed(new [x y : T] P)
                 then erase(new [x y] P) else ill-typed-process fi .
```

We check whether process `new [x y : T] P` is well-typed; if so, we rewrite it as an equivalent process in which each occurrence of `new [x y : T]` is replaced by `new [x y]` through the function `erase`. Otherwise, process `new [x y : T] P` is rewritten as `ill-typed-process`, a constant that denotes that the process has a typing error and cannot be executed.

Example 4.3. The process

$$P_1 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3)(\overline{x_3}\,\mathsf{true}.\overline{x_1}\,\mathsf{true}.\overline{x_2}\,\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_3(z).\mathsf{lin}\ y_2(x).\mathsf{lin}\ y_1(w).\mathbf{0})$$

can be implemented in Maude as:

```
op P1 : -> Trm .
eq P1 = new['y1' 'x1' : lin ? bool . end]
          new['y2' 'x2' : lin ? bool . end]
           new ['y3' 'x3' : lin ? bool . end]
             ('x3'{0} < True > . 'x1'{0} < True > . 'x2'{0} < False > . nil |
              lin 'y3'{0}('z') . lin 'y2'{0}('x') . lin 'y1'{0}('w') . nil) .
```

Then, by using the function `well-typed` we can confirm that P_1 is well-typed:

```
reduce in TEST : well-typed(P1) .
rewrites: 143 in 0~ms cpu (0~ms real) (~ rewrites/second)
result Bool: true
```

5. Lock and deadlock detection in Maude

Although the type system for sπ given in [30] enables us to statically detect processes whose variables are used according to their ascribed protocols (expressed as session types), there are processes that are well-typed but that still exhibit unwanted behaviors, in particular deadlocks.

Example 5.1. As a simple example, consider the process

$$P = \overline{x_3}\,\mathsf{true}.\overline{x_1}\,\mathsf{true}.\overline{y_2}\,\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_3(z).\mathsf{lin}\ x_2(w).\mathsf{lin}\ y_1(t).\mathbf{0}$$

P is well-typed in a context $x_1 : \mathsf{lin}\ !\mathsf{bool.end}$, $y_1 : \mathsf{lin}\ ?\mathsf{bool.end}$, $x_2 : \mathsf{lin}\ ?\mathsf{bool.end}$, $y_2 : \mathsf{lin}\ !\mathsf{bool.end}$, $x_3 : \mathsf{lin}\ !\mathsf{bool.end}$, $y_3 : \mathsf{lin}\ ?\mathsf{bool.end}$. Process $(\boldsymbol{\nu} x_1 y_1 x_2 y_2 x_3 y_3)P$ can reduce but becomes deadlocked after such a synchronization, due to a circular dependency on variables x_1, y_1, x_2, y_2.

In this section we characterize deadlocks in sπ and we show how to use the rewrite specification of the operational semantics and the Maude tools for detecting processes with deadlocks.

5.1. Definitions

We follow the formulation of deadlock and lock freedom given by Padovani [20], which uses the notion of *pending communication*. We start by defining the reduction contexts $\mathcal{C}$:

$$\mathcal{C} ::= [\,] \ \mid\ (\mathcal{C} \mid P) \ \mid\ (\boldsymbol{\nu} xy)\mathcal{C}$$

The notion of pending communication in a process P with respect to variables x, y is defined with the following auxiliary predicates:

$$\begin{aligned}
\mathsf{in}(x, P) &\overset{\text{def}}{\Longleftrightarrow} P \equiv \mathcal{C}[\mathsf{lin}\ x(y).Q\,] \wedge x \notin \mathrm{bv}(\mathcal{C}) \\
\mathsf{in}^*(x, P) &\overset{\text{def}}{\Longleftrightarrow} P \equiv \mathcal{C}[\mathsf{un}\ x(y).Q\,] \wedge x \notin \mathrm{bv}(\mathcal{C}) \\
\mathsf{out}(x, P) &\overset{\text{def}}{\Longleftrightarrow} P \equiv \mathcal{C}[\overline{x}v.Q\,] \wedge x \notin \mathrm{bv}(\mathcal{C}) \\
\mathsf{sync}(x, y, P) &\overset{\text{def}}{\Longleftrightarrow} (\mathsf{in}(x, P) \vee \mathsf{in}^*(x, P)) \wedge \mathsf{out}(y, P) \\
\mathsf{wait}(x, y, P) &\overset{\text{def}}{\Longleftrightarrow} (\mathsf{in}(x, P) \vee \mathsf{out}(y, P)) \wedge \neg\mathsf{sync}(x, y, P)
\end{aligned}$$

where we assume the extension of function $\mathrm{bv}(.)$ to reduction contexts. Intuitively, the first three predicates express the existence of a pending communication on a variable x. More in details:

- Predicate $\mathsf{in}(x, P)$ holds if x is free in P and there is a subprocess of P that is able to make a linear input on x. Predicate $\mathsf{in}^*(x, P)$ is its analog for unrestricted inputs.
- Predicate $\mathsf{out}(x, P)$ holds if x is free in P and a subprocess of P is waiting to send a value v.
- Predicate $\mathsf{sync}(x, y, P)$ denotes a pending input on x for which a synchronization on y is immediately possible.
- Predicate $\mathsf{wait}(x, y, P)$ denotes a pending input/output for which a synchronization on x, y is not immediately possible.

Let us write $\longrightarrow^*$ to denote the reflexive, transitive closure of $\longrightarrow$. Also, write $P \not\longrightarrow$ if there is no Q such that $P \longrightarrow Q$. With these elements, we may now characterize the deadlock and lock freedom properties. We say process P is

- *deadlock free* if for every Q such that $P \longrightarrow^* (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)\ldots(\boldsymbol{\nu} x_n y_n)Q \not\longrightarrow$ it holds that $\neg\mathsf{wait}(x_i, y_i, Q)$ for every x_i.
- *lock free* if for every Q such that $P \longrightarrow^* (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)\ldots(\boldsymbol{\nu} x_n y_n)Q$ and $\mathsf{wait}(x_i, y_i, Q)$ there exists R such that $Q \longrightarrow^* R$ and $\mathsf{sync}(x_i, y_i, R)$ hold.

This way, a process is deadlock free if there are not stable states with pending inputs or outputs; a process is lock free if it is able to eventually perform a synchronization in any pending input or output.

5.2. Verification in Maude

To verify deadlock freedom and lock freedom for typed processes, we can use Maude's reachability tool `search` and the built-in LTL model checker `modelCheck`. These tools can only be used for processes whose set of reachable states is finite [4,10]. We remark that infinite state processes in the sπ calculus can only be defined using input-guarded replication. Correspondingly, termination is only guaranteed for processes P for which the set $\mathsf{Reach}(P) = \{Q \mid P \longrightarrow^* Q\}$ is finite.

We start by representing the previous predicates over process terms as functions in Maude over instances of the sorts `Trm` and `Chan`:

```
ops in out in* : Chan Trm -> Bool .
ops sync wait : Chan Chan Trm -> Bool .
op wait-aux : QidSet Trm -> Bool .
eq in(a, lin a(x) . Q | R) = true .
eq in(a, P) = false [owise] .
eq in*(a, un a(x) . Q | R) = true .
eq in*(a, P) = false [owise] .
eq out(a, a < v > . Q | R) = true .
eq out(a, P) = false [owise] .
eq sync(a, b, P) = (in(a, P) or in*(a, P)) and out(b, P) .
eq wait(a, b, P) = (in(a, P) or out(b, P)) and not sync(a, b, P) .
eq wait-aux(mt, P) = false .
eq wait-aux((x y) nl, P) = wait(x{0}, y{0}, P) or wait(y{0}, x{0}, P) or wait-aux(nl, P) .
```

Above, we use function `wait-aux` to determine if a group of pairs of co-variables contains a pair for which there is a pending communication.

The deadlock freedom property imposes that there should be no stable states in which there are pending communications. Consequently, we can use the Maude command `search` as follows to find counterexamples:

```
search init =>! new* [nl:QidSet] P:Trm such that wait-aux(nl:QidSet, P:Trm) .
```

Above, `init` denotes the initial state for the process to be checked. We recall that the `search` command with the arrow `=>!` looks for final (stable) states. In this way, `init` is deadlock free if the search returns no solution. Solutions to this search are states in which there is no further reduction and the wait property holds for at least a pair of co-variables.

The lock freedom property requires checking some intermediate states and so the reachability tool cannot be used. Liveness properties such as lock freedom can be verified by model checking [7]. Given a process $(\boldsymbol{\nu} x_1, y_1)\ldots(\boldsymbol{\nu} x_n, y_n)P$ in sπ, lock freedom can be stated by means of the following *Computation Tree Logic* (CTL) formula:

$$\bigwedge_{(x_i, y_i) \in vars(P)} \mathsf{A}\Box(\mathsf{wait}(x_i, y_i, Q) \longrightarrow \mathsf{E}\Diamond\mathsf{sync}(x_i, y_i, R)) \tag{1}$$

Formula (1) says that in every execution path of P, whenever a state (process) Q satisfies $\mathsf{wait}(x_i, y_i, Q)$, there is a path in which a state R satisfying $\mathsf{sync}(x_i, y_i, R)$ is reachable.

Currently, Maude does not support the verification of CTL formulas (which may require verifying the existence of specific paths); instead, it supports the verification of LTL formulas. Consequently, we state the following LTL formula:

$$\bigwedge_{(x_i, y_i) \in vars(P)} \Box(\Diamond\mathsf{wait}(x_i, y_i, Q) \longrightarrow \Diamond\mathsf{sync}(x_i, y_i, R)) \tag{2}$$

which is different to (1) and can be read as: for every execution path of P, whenever a state Q holding the $\mathsf{wait}(x_i, y_i, Q)$ property is reached, then in every path starting in Q the property $\mathsf{sync}(x_i, y_i, R)$ will hold in some reachable process R. The difference is subtle: while Formula (2) states that $\mathsf{sync}(x_i, y_i, R)$ must hold for *all execution paths* where $\mathsf{wait}(x_i, y_i, Q)$ holds, Formula (1) states that $\mathsf{sync}(x_i, y_i, R)$ must hold for *some execution path*.

We can verify Formula (2) by using Maude's built-in LTL model checker. This formula is sound, i.e., if P satisfies (2) then P is lock free. Completeness is not guaranteed in general, however; it holds for well-typed processes with no replicated

input (see Appendix C). Consequently, we can use Maude to verify lock freedom in finite-state processes without infinite execution paths.

Below, we define the Maude predicates `psync` and `pwait` that we will use in the LTL model checker:

```
ops pwait psync : Chan Chan -> Prop [ctor] .
eq new* [(x y) nl] P |= pwait(x{0}, y{0}) = wait(x{0}, y{0}, P) or wait(y{0}, x{0}, P) .
eq new* [(x y) nl] P |= psync(x{0}, y{0}) = sync(x{0}, y{0}, P) or sync(y{0}, x{0}, P) .
```

In the predicates `psync` and `pwait`, we use normalized processes, i.e., processes where the nested scope restrictions are flattened in an equivalent process that uses the operator `new*`. This assumption simplifies the definitions. Both `psync` and `pwait` predicates use the functions `in`, `in*`, `out`, `sync`, and `wait` as expected according to the definition.

The Kripke structure that is generated for Maude will use such a normalized process term as the initial state. The Maude predicates `pwait` and `psync` hold with respect to a pair of dual variables if there is a pending communication and there is a synchronization in the process associated to a state. The lock freedom property imposes for each variable that if in any state there is a pending communication then eventually there will be a synchronization. Formalizing lock freedom requires to check each possible subject; accordingly, the LTL formula associated to this property depends on the variables being used in the process. We define a function `build-lock-formula` that takes the used variables and builds the corresponding LTL formula:

```
op build-lock-formula : QidSet -> Formula .
eq build-lock-formula(mt) = True .
eq build-lock-formula((x y) nl) = [] (<> pwait(x{0}, y{0}) -> <> psync(x{0}, y{0})) /\
                                  build-lock-formula(nl) .
```

This way, function `build-lock-formula` recursively builds the Maude encoding for the LTL formula in (2). The resulting LTL formula corresponds to the conjunction of the LTL subformulas associated to each dual variable. It is worth highlighting that the Maude operators `->` and `/\` stand for the logical implication and conjunction, respectively, whereas operators `[]` and `<>` stand for the *always* and *eventually* temporal operators, respectively.

The model checker can be used as follows:

```
red modelCheck(init, build-lock-formula(vars)) .
```

where `init` and `vars` stand for the process term and a set of pairs of co-variables, respectively. If `init` is lock free then the invocation of `modelCheck` will produce `true`. Otherwise, the invocation will show a counterexample with a sequence of rules that produces a state where the formula is not fulfilled; counterexamples are shown as a pair consisting of two lists of transitions separated by a comma:

```
 counterexample(
{State_0, RuleLabel_0}{State_1, RuleLabel_1} ... {State_i, RuleLabel_i},
 {State_i+1, RuleLabel_i+1} ... {State_n, RuleLabel_n})
```

Above, each transition `{State_i, RuleLabel_i}` gives the current state and the rule to be applied. The first list of transitions corresponds to a finite execution path starting with `init`, and the transition list after the comma describes a loop; in the literature, this kind of counterexamples is often referred to as "lasso-shaped". In general, when an LTL formula φ is not satisfied by a finite Kripke structure, then there is a counterexample of φ having a lasso shape [6]. However, since we are dealing with finite state processes, the counterexamples can be interpreted as a list separated by a comma in which a pending synchronization is never performed. Examples can be found in Section 5.3.

5.3. Examples

We give a couple of examples of well-typed processes in sπ, with different lock- and deadlock-freedom properties. Appendix C develops additional examples.

$$P_1 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3)(\overline{x_3}\,\mathsf{true}.\overline{x_1}\,\mathsf{true}.\overline{x_2}\,\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_3(z).\mathsf{lin}\ y_2(x).\mathsf{lin}\ y_1(w).\mathbf{0})$$

$$P_2 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} ab)(\overline{x_1} b.\mathbf{0} \mid \overline{a}\,\mathsf{true}.\mathbf{0} \mid \mathsf{un}\ y_1(z).\overline{x_2} z.\mathbf{0} \mid \mathsf{un}\ y_2(w).\overline{x_1} w.\mathbf{0})$$

Process P_1 reduces to a deadlock immediately after a synchronization on the co-variables x_3, y_3, whereas P_2 is a process where the variable b is repeatedly shared through communications on x_1, y_1, x_2, y_2. The process is not lock-free: b is never used to synchronize with its co-variable a. Fig. 5 gives the Maude terms associated to these processes.

Notice that process P_1 is well-typed, as shown in Example 4.3. We analyze `P1` using Maude by executing:

```
ops P1 P2 : -> Trm .
eq P1 = new* [('y1' 'x1')('y2' 'x2')('y3' 'x3')]
              ('x3'{0} < True > . 'x1'{0} < True > . 'y1'{0} < False > . nil |
               lin 'y3'{0}('z') . lin 'y2'{0}('x') . lin 'x2'{0}('w') . nil) .
eq P2 = new* [('x1' 'y1')('x2' 'y2')('a' 'b')]
              ('x1'{0} < 'b'{0} > . nil | 'a'{0} < True > . nil |
               un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
               un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil ) .
```

Fig. 5. Processes in Maude.

```
search P1 =>! new* [nl:QidSet] P:Trm such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P1, build-lock-formula(('y1' 'x1')('y2' 'x2')('y3' 'x3'))) .
```

We obtain the following results, which confirm that P1 is not deadlock free and not lock free:

```
search in TEST : P1 =>! new*[nl:QidSet]P:Trm such that wait-aux(nl:QidSet, P:Trm) = true .

Solution 1 (state 1)
states: 2  rewrites: 178 in 0~ms cpu (0~ms real) (~ rewrites/second)
    nl:QidSet --> ('x3' 'y3') ('y1' 'x1') 'y2' 'x2'
    P:Trm --> 'x1'{0} < True > . 'x2'{0} < False > . nil |
    lin 'y2'{0}('x') . lin 'y1'{0}('w') . nil

No more solutions.

result ModelCheckResult: counterexample(
    {new*[('x3' 'y3') ('y1' 'x1') 'y2' 'x2']
    'x3'{0} < True > . 'x1'{0} < True > . 'x2'{0} < False > . nil |
    lin 'y3'{0}('z') . lin 'y2'{0}('x') . lin 'y1'{0}('w') . nil,'LINCOM},
    {new*[('x3' 'y3') ('y1' 'x1') 'y2' 'x2']
    'x1'{0} < True > . 'x2'{0} < False > . nil |
    lin 'y2'{0}('x') . lin 'y1'{0}('w') . nil,deadlock})
```

The solution for the deadlock search tells us that the process

$$(\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3)(\overline{x_1}\,\mathsf{true}.\overline{x_2}\,\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_2(x).\mathsf{lin}\ y_1(w).\mathbf{0})$$

is deadlocked and reachable from P_1. Model checking shows that after one reduction step with the Rule LINCOM, P_1 reaches a process which is deadlocked and therefore P_1 is also locked.

Consider now a similar execution for process P2. This process can be implemented in Maude as follows:

```
ops TypeX1 TypeX2 Typea : -> Type .
eq TypeX1 = (u['i']un !(un ? bool . end) . var('i')) .
eq TypeX2 = (u['j']un !(un ? bool . end) . var('j')) .
eq Typea = (u['k']un !(bool) . var('k')) .
eq P2 = new['x1' 'y1' : TypeX1]new['x2' 'y2' : TypeX2]new['a' 'b' : Typea]
              ('x1'{0}< 'b'{0} > . nil | un 'y1'{0}('z') . 'x2'{0}< 'z'{0} > . nil |
               un 'y2'{0}('z') . 'x1'{0}< 'z'{0} > . nil) .
```

We first verify that it is well-typed by using function `well-typed`:

```
reduce in TEST : well-typed(P2) .
rewrites: 251 in 0~ms cpu (0~ms real) (~ rewrites/second)
result Bool: true
```

Now we analyze its (dead)lock properties:

```
search P2 =>! new* [nl:QidSet] P:Trm such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P2, build-lock-formula(('x1' 'y1')('x2' 'y2')('a' 'b'))) .
```

Maude returns the following results, which confirm that P2 is an infinite, deadlock free process:

```
search in TEST : P2 =>! new*[nl:QidSet]P:Trm such that wait-aux(nl:QidSet, P:Trm) = true .

No solution.

result ModelCheckResult: counterexample(nil,
   {new*[('a' 'b') ('x1' 'y1') 'x2' 'y2']
```

$$
\begin{aligned}
n &::= a, b \mid s, \overline{s} \\
u, w &::= n \mid x, y, z \\
V, W &::= u \mid \lambda x.\, P \mid x, y, z \\
P, Q &::= u!\langle V\rangle.P \mid u?(x).P \mid u \triangleleft l.P \mid u \triangleright \{l_i : P_i\}_{i\in I} \mid V\, u \mid \quad \mid P \mid Q \mid (\nu\, n)P \mid \mathbf{0} \mid X \mid \mu X.P
\end{aligned}
$$

Fig. 6. Syntax of HOπ.

```
    'a'{0} < True > . nil | 'x1'{0} < 'b'{0} > . nil |
    un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
    un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil, 'UNCOM}
{new*[('a' 'b') ('x1' 'y1') 'x2' 'y2']
    'a'{0} < True > . nil | 'x2'{0} < 'b'{0} > . nil |
    un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
    un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil, 'UNCOM})
```

Because the model checking technique for lock freedom is not complete in general, we cannot ensure that P_2 is locked from the fact that there is a counterexample.

6. The case of higher-order session processes

In *higher-order* process calculi, processes can communicate names but also *abstractions* (functions from names to processes). Here we consider HOπ, a higher-order process calculus with session constructs; this is an interesting formalism, as it distills and supports directly the essential features of functional languages with concurrency. The fundamental theory of HOπ (behavioral equivalence, relative expressiveness) has been thoroughly investigated by Kouzapas et al. [17,18]. In this section we illustrate how our approach to executable specifications and type checking in Maude for sπ can be suitably adapted to HOπ.

We largely follow the structure and results that we developed for sπ in Sections 2–5. In Section 6.1 we summarize the syntax, semantics, and type system of HOπ, following [18]. Then, in Section 6.2 we present its executable semantics in Maude. The implementation of session type checking in Maude requires an algorithmic type system, which existed for sπ but not for HOπ. Hence, in Section 6.3 we first develop an algorithmic type system for HOπ and prove its correspondence with the type system in Section 6.1. Subsequently, we present and illustrate the Maude implementation of this type system. Finally, in Section 6.4 we adapt the approach for deadlocks and locks detection from sπ to HOπ.

6.1. The typed process model HOπ

6.1.1. Syntax and semantics

Names $a, b, c, \ldots$ (resp. $s, \overline{s}, \ldots$) range over shared (resp. session) names. Names $m, n, t, \ldots$ are session or shared names. Dual endpoints are $\overline{n}$ with $\overline{\overline{s}} = s$ and $\overline{a} = a$. Variables are denoted with $x, y, z, \ldots$, and recursive variables are denoted with $X, Y, \ldots$. An abstraction $\lambda x.\, P$ is a process P with name parameter x. *Values* $V, W, \ldots$ include identifiers $u, v, \ldots$ and abstractions $\lambda x.\, P$ (first- and higher-order values, resp.).

The syntax of HOπ calculus is presented in Fig. 6. Process terms $P, Q, \ldots$ include usual prefixes for sending and receiving values V. Processes $u \triangleleft l.P$ and $u \triangleright \{l_i : P_i\}_{i\in I}$ constructs for selection and branching. Process $V\, u$ denotes application; it substitutes name u on the abstraction V. Typing ensures that V is not a name. Recursion $\mu X.P$ binds the recursive variable X in P. Constructs for inaction $\mathbf{0}$, parallel composition $P_1 \mid P_2$, and name restriction $(\nu\, n)P$ are standard.

Notation 1. We shall write $* P$ to denote a replicated process P, representable as $\mu X.(P \mid X)$.

In sπ, session name restriction was based on co-variables and denoted $(\boldsymbol{\nu} xy)P$. Differently, session name restriction in HOπ is based on dual endpoints s and $\overline{s}$ and denoted $(\nu\, s)P$; the idea is that this construct simultaneously binds s and $\overline{s}$ in P. Functions $\texttt{fv}(P)$, $\texttt{fn}(P)$, and $\texttt{fs}(P)$ denote, respectively, the sets of free variables, names, and session names in P, and are defined as expected. We assume V in $u!\langle V\rangle.P$ does not include free recursive variables X. If $\texttt{fv}(P) = \emptyset$, we call P *closed*.

In a statement, a name (resp. variable) is *fresh* if it is not among the names (resp. variables) of the objects (processes, actions, etc.) of the statement. We shall follow Barendregt's convention: all (session) names and variables in binding occurrences, in any mathematical context, are pairwise distinct but also distinct from free (session) names and variables.

Operational semantics The operational semantics of HOπ is defined in terms of a *reduction relation*, denoted $\longrightarrow$, whose rules are given in Fig. 7 (top). We briefly describe the rules. Rule [App] defines name application. Rule [Pass] defines a shared interaction at n (with $\overline{n} = n$) or a session interaction. Rule [Sel] is the standard rule for labeled choice/selection. Other rules are standard π-calculus rules. Reduction is closed under *structural congruence*, noted $\equiv$ and given in Fig. 7 (bottom). We

$$(\lambda x.\, P)\, u \longrightarrow P\{{}^{u}/x\} \qquad [\text{App}]$$
$$n!\langle V\rangle.P \mid \overline{n}?(x).Q \longrightarrow P \mid Q\{{}^{V}/x\} \qquad [\text{Pass}]$$
$$n \triangleleft l_j.Q \mid \overline{n} \triangleright \{l_i : P_i\}_{i\in I} \longrightarrow Q \mid P_j \quad (j \in I) \qquad [\text{Sel}]$$
$$P \longrightarrow P' \Rightarrow (\nu\, n)P \longrightarrow (\nu\, n)P' \qquad [\text{Res}]$$
$$P \longrightarrow P' \Rightarrow P \mid Q \longrightarrow P' \mid Q \qquad [\text{Par}]$$
$$P \equiv Q \longrightarrow Q' \equiv P' \Rightarrow P \longrightarrow P' \qquad [\text{Cong}]$$
$$P \mid \mathbf{0} \equiv P \quad P_1 \mid P_2 \equiv P_2 \mid P_1 \quad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \quad (\nu\, n)\mathbf{0} \equiv \mathbf{0}$$
$$P \mid (\nu\, n)Q \equiv (\nu\, n)(P \mid Q)\ (n \notin \mathtt{fn}(P)) \quad \mu X.P \equiv P\{{}^{\mu X.P}/X\} \quad P \equiv Q \text{ if } P \equiv_\alpha Q$$

Fig. 7. Operational semantics of HOπ.

$$\begin{array}{lcl} U & ::= & C \mid L \\ C & ::= & S \mid \langle S\rangle \mid \langle L\rangle \\ L & ::= & C\to\diamond \mid C\multimap\diamond \\ S & ::= & !\langle U\rangle.S \mid ?(U).S \mid \oplus\{l_i : S_i\}_{i\in I} \mid \&\{l_i : S_i\}_{i\in I} \mid \mu\mathsf{t}.S \mid \mathsf{t} \mid \mathtt{end} \end{array}$$

Fig. 8. Syntax of session types for HOπ.

write $\equiv_\alpha$ to denote α-conversion and assume the expected extension of $\equiv$ to values V. We write $\longrightarrow^*$ for a multi-step reduction.

Example 6.1. Consider the HOπ process

$$P = a!\langle \lambda x.\, x!\langle t\rangle.\mathbf{0}\rangle.\mathbf{0} \mid \overline{a}?(z).(z\, h_1) \mid \overline{a}?(w).(w\, h_2) \mid \overline{h_1}?(v).Q_1 \mid \overline{h_2}?(r).Q_2$$

where Q_1 and Q_2 are left unspecified. Suppose the first and second sub-processes synchronize on a: by applying Rules [Pass] and [App] in Fig. 7, we would have:

$$P \longrightarrow\longrightarrow h_1!\langle t\rangle.\mathbf{0} \mid \overline{a}?(w).(w\, h_2) \mid \overline{h_1}?(v).Q_1 \mid \overline{h_2}?(r).Q_2$$

Then, by applying Rule Pass, we obtain:

$$P \longrightarrow^* \overline{a}?(w).(w\, h_2) \mid Q_1\{{}^{v}/t\} \mid \overline{h_2}?(r).Q_2$$

Now suppose that the first reduction step from P concerned the first and third sub-process. Then we would have:

$$P \longrightarrow^* \overline{a}?(z).(z\, h_1) \mid \overline{h_1}?(v).Q_1 \mid Q_2\{{}^{v}/r\}$$

6.1.2. Session types for HOπ

The syntax of types for HOπ is given in Fig. 8. We write $\diamond$ to denote the process type. Value type U includes first-order types C and higher-order types L. Types $C\to\diamond$ and $C\multimap\diamond$ denote *shared* and *linear* higher-order types, respectively. Session types, denoted by S, follow the session type syntax for sπ, given in Section 4, with the extension that carried types U may be higher-order. Shared channel types are denoted $\langle S\rangle$ and $\langle L\rangle$.

As expected, the session type system for HOπ depends on a notion of duality on types. As in sπ, duality is obtained by swapping ! by ?, ? by !, $\oplus$ by &, and & by $\oplus$, including the fixed point construction. In this section, we write S_1 dual S_2 to mean that S_1 and S_2 are dual; the formal definition is coinductive, and given in [17].

We consider shared, linear, and session *environments*, denoted Γ, Λ, and Δ, resp.:

$$\begin{array}{lcl} \Gamma & ::= & \emptyset \mid \Gamma, x : C\to\diamond \mid \Gamma, u : \langle S\rangle \mid \Gamma, u : \langle L\rangle \mid \Gamma, X : \Delta \\ \Lambda & ::= & \emptyset \mid \Lambda, x{:}C\multimap\diamond \\ \Delta & ::= & \emptyset \mid \Delta, u{:}S \end{array}$$

Γ maps variables and shared names to value types, and recursive variables to session environments; it admits weakening, contraction, and exchange principles. Λ maps variables to linear higher-order types; Δ maps session names to session types. Both Λ and Δ are only subject to exchange. The domains of Γ, Λ, and Δ are assumed pairwise distinct. We write Δ_1, Δ_2 for the disjoint union of Δ_1 and Δ_2. We write $\Gamma\backslash x$ to denote the environment obtained from Γ by removing the assignment $x : U\to\diamond$, for some U. Similarly, we write $\Delta_1\backslash\Delta_2$ and $\Lambda_1\backslash\Lambda_2$ with the expected reading.

$$\text{(Sess)}\ \Gamma;\emptyset;\{u:S\}\vdash u\triangleright S \qquad \text{(Sh)}\ \Gamma,u:U;\emptyset;\emptyset\vdash u\triangleright U$$

$$\text{(LVar)}\ \Gamma;\{x:C\multimap\diamond\};\emptyset\vdash x\triangleright C\multimap\diamond \qquad \text{(RVar)}\ \Gamma,X:\Delta;\emptyset;\Delta\vdash X\triangleright\diamond$$

$$\text{(Abs)}\ \frac{\Gamma;\Lambda;\Delta_1\vdash P\triangleright\diamond \quad \Gamma;\emptyset;\Delta_2\vdash x\triangleright C}{\Gamma\backslash x;\Lambda;\Delta_1\backslash\Delta_2\vdash \lambda x.\,P\triangleright C\multimap\diamond} \qquad \text{(App)}\ \frac{\Gamma;\Lambda;\Delta_1\vdash V\triangleright C\rightsquigarrow\diamond \quad \rightsquigarrow\in\{\multimap,\rightarrow\} \quad \Gamma;\emptyset;\Delta_2\vdash u\triangleright C}{\Gamma;\Lambda;\Delta_1,\Delta_2\vdash V\,u\triangleright\diamond}$$

$$\text{(Prom)}\ \frac{\Gamma;\emptyset;\emptyset\vdash V\triangleright C\multimap\diamond}{\Gamma;\emptyset;\emptyset\vdash V\triangleright C\rightarrow\diamond} \qquad \text{(EProm)}\ \frac{\Gamma;\Lambda,x:C\multimap\diamond;\Delta\vdash P\triangleright\diamond}{\Gamma,x:C\rightarrow\diamond;\Lambda;\Delta\vdash P\triangleright\diamond} \qquad \text{(End)}\ \frac{\Gamma;\Lambda;\Delta\vdash P\triangleright T \quad u\notin \mathrm{dom}(\Gamma,\Lambda,\Delta)}{\Gamma;\Lambda;\Delta,u:\mathsf{end}\vdash P\triangleright\diamond}$$

$$\text{(Rec)}\ \frac{\Gamma,X:\Delta;\emptyset;\Delta\vdash P\triangleright\diamond}{\Gamma;\emptyset;\Delta\vdash \mu X.P\triangleright\diamond} \qquad \text{(Par)}\ \frac{\Gamma;\Lambda_i;\Delta_i\vdash P_i\triangleright\diamond \quad i=1,2}{\Gamma;\Lambda_1,\Lambda_2;\Delta_1,\Delta_2\vdash P_1\mid P_2\triangleright\diamond} \qquad \text{(Nil)}\ \frac{}{\Gamma;\emptyset;\emptyset\vdash \mathbf{0}\triangleright\diamond}$$

$$\text{(Send)}\ \frac{u:S\in\Delta_1,\Delta_2 \quad \Gamma;\Lambda_1;\Delta_1\vdash P\triangleright\diamond \quad \Gamma;\Lambda_2;\Delta_2\vdash V\triangleright U}{\Gamma;\Lambda_1,\Lambda_2;((\Delta_1,\Delta_2)\setminus u:S),u:!\langle U\rangle.S\vdash u!\langle V\rangle.P\triangleright\diamond}$$

$$\text{(Req)}\ \frac{\Gamma;\Lambda;\Delta_1\vdash P\triangleright\diamond \quad \Gamma;\emptyset;\emptyset\vdash u\triangleright\langle\mathcal{U}\rangle \quad \Gamma;\emptyset;\Delta_2\vdash V\triangleright\mathcal{U} \quad \mathcal{U}\in\{S,L\}}{\Gamma;\Lambda;\Delta_1,\Delta_2\vdash u!\langle V\rangle.P\triangleright\diamond}$$

$$\text{(Rcv)}\ \frac{\Gamma;\Lambda_1;\Delta_1,u:S\vdash P\triangleright\diamond \quad \Gamma;\Lambda_2;\Delta_2\vdash x\triangleright U}{\Gamma\backslash x;\Lambda_1\backslash\Lambda_2;\Delta_1\backslash\Delta_2,u:?(U).S\vdash u?(x).P\triangleright\diamond}$$

$$\text{(Acc)}\ \frac{\Gamma;\Lambda_1;\Delta_1\vdash P\triangleright\diamond \quad \Gamma;\emptyset;\emptyset\vdash u\triangleright\langle\mathcal{U}\rangle \quad \Gamma;\Lambda_2;\Delta_2\vdash x\triangleright\mathcal{U} \quad \mathcal{U}\in\{S,L\}}{\Gamma\backslash x;\Lambda_1\backslash\Lambda_2;\Delta_1\backslash\Delta_2\vdash u?(x).P\triangleright\diamond}$$

$$\text{(Bra)}\ \frac{\forall i\in I \quad \Gamma;\Lambda;\Delta,u:S_i\vdash P_i\triangleright\diamond}{\Gamma;\Lambda;\Delta,u:\&\{l_i:S_i\}_{i\in I}\vdash u\triangleright\{l_i:P_i\}_{i\in I}\triangleright\diamond} \qquad \text{(Sel)}\ \frac{\Gamma;\Lambda;\Delta,u:S_j\vdash P\triangleright\diamond \quad j\in I}{\Gamma;\Lambda;\Delta,u:\oplus\{l_i:S_i\}_{i\in I}\vdash u\triangleleft l_j.P\triangleright\diamond}$$

$$\text{(ResS)}\ \frac{\Gamma;\Lambda;\Delta,s:S_1,\overline{s}:S_2\vdash P\triangleright\diamond \quad S_1\ \mathsf{dual}\ S_2}{\Gamma;\Lambda;\Delta\vdash(\nu\,s)P\triangleright\diamond} \qquad \text{(Res)}\ \frac{\Gamma,a:\langle S\rangle;\Lambda;\Delta\vdash P\triangleright\diamond}{\Gamma;\Lambda;\Delta\vdash(\nu\,a)P\triangleright\diamond}$$

Fig. 9. Typing rules for $\mathsf{HO}\pi$.

Given the above intuitions for environments, the typing judgments for values V and processes P are denoted $\Gamma;\Lambda;\Delta\vdash V\triangleright U$ and $\Gamma;\Lambda;\Delta\vdash P\triangleright\diamond$, respectively.

Fig. 9 gives the typing rules. We now describe some of them; see [17] for a full account. The shared type $C\rightarrow\diamond$ is derived using Rule (Prom) only if the value has a linear type with an empty linear environment. Rule (EProm) allows us to freely use a shared type variable as linear. Abstraction values are typed with Rule (Abs). Application typing is governed by Rule (App): we expect the type C of an application name u to match the type of the application variable x (i.e., $C\multimap\diamond$ or $C\rightarrow\diamond$). In Rule (Send), the type U of value V should appear as a prefix in the session type $!\langle U\rangle.S$ of u. Rule (Rcv) is its dual. Rules (Req) and (Acc) type interaction along shared names; the type of the sent/received object (S and L, resp.) should match the type of the sent/received subject ($\langle S\rangle$ and $\langle L\rangle$, resp.).

We state *type soundness* for $\mathsf{HO}\pi$, as established in [17]. We require two auxiliary definitions. First, we focus on *balanced* session environments:

Definition 6.1 *(Balanced environments).* We say that a session environment Δ is *balanced* if whenever $s:S_1,\overline{s}:S_2\in\Delta$ then S_1 dual S_2.

Second, we define a notion of reduction for session environments:

Definition 6.2. We define the relation $\longrightarrow$ on session environments Δ as:

$$\Delta,s:!\langle U\rangle.S_1,\overline{s}:?(U).S_2\longrightarrow\Delta,s:S_1,\overline{s}:S_2$$
$$\Delta,s:\oplus\{l_i:S_i\}_{i\in I},\overline{s}:\&\{l_i:S_i'\}_{i\in I}\longrightarrow\Delta,s:S_k,\overline{s}:S_k'\ (k\in I)$$

$$\mathsf{Client} \stackrel{\text{def}}{=} (\nu\, h_1)(\nu\, h_2)(s_1!\langle \lambda x.\, P_{xy}\{h_1/y\}\rangle.s_2!\langle \lambda x.\, P_{xy}\{h_2/y\}\rangle.\mathbf{0} \mid \overline{h_1}?(x).\overline{h_2}?(y).\mathtt{if}\ x \le y\ \mathtt{then}\ \overline{h_1} \triangleleft \mathsf{accept}.\overline{h_2} \triangleleft \mathsf{reject}.\mathbf{0}\ \mathtt{else}\ \overline{h_1} \triangleleft \mathsf{reject}.\overline{h_2} \triangleleft \mathsf{accept}.\mathbf{0})$$

$$P_{xy} \stackrel{\text{def}}{=} x!\langle \mathsf{room}\rangle.x?(quote).y!\langle quote\rangle.y \triangleright \left\{ \begin{array}{l} \mathsf{accept} : x \triangleleft \mathsf{accept}.x!\langle \mathsf{credit}\rangle.\mathbf{0}\ , \\ \mathsf{reject} : x \triangleleft \mathsf{reject}.\mathbf{0} \end{array} \right\}$$

Fig. 10. A possible HOπ implementation of the Hotel Booking scenario [17].

We write $\longrightarrow^*$ to denote multi-step reduction.

We then have:

Theorem 6.1 *(Type soundness [17]). Suppose $\Gamma; \emptyset; \Delta \vdash P \triangleright \diamond$ with Δ balanced. Then $P \longrightarrow P'$ implies $\Gamma; \emptyset; \Delta' \vdash P' \triangleright \diamond$ and $\Delta = \Delta'$ or $\Delta \longrightarrow \Delta'$ with Δ' balanced.*

6.1.3. An example

We illustrate processes and types by recalling the *hotel booking scenario* developed in [17,18]. The scenario involves a Client process that wants to book a hotel room. Client narrows the choice down to two hotels, and requires a quote from the two in order to decide. The round-trip time (RTT) required for taking quotes from the two hotels is not optimal, so the client sends mobile processes to both hotels to automatically negotiate and book a room.

Fig. 10 gives an implementation of this scenario in HOπ. We write $\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ to denote a conditional process that executes P_1 or P_2 depending on boolean expression e (this process is encodable using labeled choice). The implementation is given by process Client: it sends two abstractions with body P_{xy}, one to each hotel, using sessions s_1 and s_2. In P_{xy}, name x is meant to be instantiated by the hotel as the negotiating endpoint, whereas name y is used to interact with Client. Intuitively, process P_{xy}: (i) sends the room requirements to the hotel; (ii) receives a quote from the hotel; (iii) sends the quote to Client; (iv) expects a choice from Client whether to accept or reject the offer; (v) if the choice is accept then it informs the hotel and performs the booking; otherwise, if the choice is reject then it informs the hotel and ends the session. Client instantiates two copies of P_{xy} as abstractions on session x. It uses fresh endpoints h_1, h_2 to substitute channel y in P_{xy}. This enables communication with the mobile code(s): Client uses the dual endpoints $\overline{h_1}$ and $\overline{h_2}$ to receive the negotiation result from the two remote instances of P and then inform the two processes for the final booking decision.

To illustrate the type system of HOπ, we give types to the client processes. Assume

$$S = !\langle \mathsf{quote}\rangle.\&\{\mathsf{accept} : \mathtt{end}, \mathsf{reject} : \mathtt{end}\}$$
$$U = !\langle \mathsf{room}\rangle.?(\mathsf{quote}).\oplus\{\mathsf{accept} : !\langle \mathsf{credit}\rangle.\mathtt{end}, \mathsf{reject} : \mathtt{end}\}$$

where quote, room, and credit are (first-order) base types. We then have:

$$\emptyset; \emptyset;\ y : S \vdash \lambda x.\, P_{xy} \triangleright U \multimap \diamond$$
$$\emptyset; \emptyset;\ s_1 : !\langle U \multimap \diamond\rangle.\mathtt{end}, s_2 : !\langle U \multimap \diamond\rangle.\mathtt{end} \vdash \mathsf{Client} \triangleright \diamond$$

6.2. Rewriting semantics for HOπ

In this section we present the rewriting semantics for HOπ. We base our presentation on the rewriting semantics for sπ, given in Section 3, highlighting differences where appropriate.

6.2.1. Syntax

As for sπ, we use the Maude sorts `Value`, `Chan`, `Trm`, which correspond to the syntactic categories values, names, and terms. Similarly, we also have the auxiliary sorts `Choice` and `ChoiceSet` and some additional sorts (`Variable`, `SessionName`, and `SharedName`).

```
sorts Trm Guard Choice Choiceset Chan Value SessionName SharedName Variable .
subsort Chan < Value .
subsort SessionName < Chan .
subsort SharedName < Chan .
subsort Variable < Value .
subsort Choice < Choiceset .

op _{_} : Qid Nat -> SessionName [ctor prec 1] .
op _(_) : Qid Nat -> SharedName [ctor prec 1] .
op lambda(_)._ : Qid Trm -> Value [ctor prec 5] .
```

```
op _<_> : Chan Value -> Guard [ctor prec 6] .
op _(_) : Chan Qid -> Guard [ctor prec 4] .
op nil : -> Trm [ctor] .
op new[_]_ : Qid Trm -> Trm [ctor prec 10] .
op _{_} : Value Chan -> Trm [ctor prec 18] .
op rec : Qid -> Trm [ctor] .
op u[_]._ : Qid Trm -> Trm [ctor prec 20] .
op _|_  :  Trm Trm -> Trm [ctor assoc comm prec 12 id: nil] .
op _ << _ . _ : Chan Qid Trm -> Trm [ctor prec 15] .
op _ >> {_} : Chan Choiceset -> Trm [ctor prec 17] .
op _._ : Guard Trm -> Trm [ctor prec 7] .
op _:_ : Qid Trm -> Choice .
op empty : -> Choiceset [ctor] .
op _;_ : Choiceset Choiceset -> Choiceset [ctor assoc comm id: empty] .
```

Following the syntax in Section 6, values can be names, variables or abstractions. The distinction between names and values as well as the association of shared and session names like types of names are both obtained by means of the subsort relation. There is a specific constructor for each type of name: `_{_}` for session names and `_(_)` for shared names. Both constructors take a `Qid` and a natural number. The constructor for abstractions, denoted `lambda(_)._`,takes a `Qid` and a `Trm`, as expected.

Constructors for guards are as in Section 3 but without the qualifier in the input, which is not needed here. The constructor associated to the production rules for the process **0** and the guarded, parallel, selection, and branching terms are also as in Section 3. The constructor for restriction `new[_]_` only uses one `Qid`, as co-names are implicitly declared in HOπ. We add constructors for the recursive variables and for process application: the former is denoted `rec` and takes a `Qid`, whereas the latter is denoted `_{_}` and relies on a `Value` and a `Chan`, as HOπ has first-order abstractions. The definition of the constructor for recursion is as expected.

6.2.2. Substitutions and structural congruence

Similarly as sπ, the semantics of HOπ relies on substitutions of names with values. We also adopt the CINNI calculus [25] to define explicit substitutions. In addition to the definitions given in Section 3, we add some equations for dealing with shared names and the co-name operator.

```
eq  [ a := v ] a(0) = v .
eq  [ a := v ] a(s(n)) = a(n) .
ceq [ a := v ] b(n) = b(n)  if a =/= b .
eq  [ shiftup a ] a(n) = a(s(n)) .
ceq [ shiftup a ] b(n) = b(n)  if a =/= b .
eq  [ lift a S ] a(0) = a(0) .
eq  [ lift a S ] a(s(n)) = [ shiftup a ] S a(n) .
ceq [ lift a S ] b(n) = [ shiftup a ] S b(n) if a =/= b .
eq  S *(x) = *(S x) .
```

As before, we specify the equations that enable the explicit substitutions for the process terms according to the syntax of HOπ. Each rule is concerned to a specific production rule. Next we report the equations associated to the production rules that differ from of the ones given in Section 3:

```
eq  S (new [x] P) = new [x] ([lift x S] P) .
eq  S (a < V > . P) = (S a) < apply-subst(S, V) > . (S P) .
eq  S rec(x) = rec(x) .
eq  S (u[x] . P) = u[x] . ([lift x S] P) .
eq  S (V {a}) = apply-subst(S, V) {(S a)} .
eq  apply-subst(S, a) = (S a) .
eq  apply-subst(S, lambda(x) . P) = lambda(x) . ([lift x S] P) .
eq  apply-subst(S, val(x)) = val(x) .
eq  apply-subst(S, V) = V [owise] .
```

Notice that, for output terms, the substitution must be applied over the subject name, the value being sent, and the continuation process. The auxiliary operation `apply-subst` applies a substitution over each type of value (names, abstractions, and variables).

For structural congruence, we follow a similar approach to the one for sπ. We declare the associative, commutative, and identity axioms for parallel composition by marking the respective constructor with the Maude equational attributes `assoc`, `comm`, and `id`, respectively. Consequently, we only need to specify some equations for the remaining rules in Fig. 7.

```
eq new[x] nil = nil .
ceq P | new[x] Q = new [x] (Q | [shiftup x] P)
    if P =/= nil /\ Q =/= nil /\ x{0} in freevars(P) .
```

6.2.3. Operational semantics

Following the implementation of sπ, to improve the efficiency of the rewriting specification, we adapt the auxiliary operator `new*` defined in Section 3. In the case of HOπ, this operator declares a list of names (and implicitly their co-names), rather than just a single name. It is worth highlighting that we use a set of `Qid` (rather than a list of `Qid`) since the order of the names being declared is irrelevant.

```
sorts QidSet .
subsort Qid < QidSet .
op mt : -> QidSet .
op __ : QidSet QidSet -> QidSet [ctor comm assoc id: mt] .
op new* [_] _ : QidSet Trm -> Trm [ctor] .
```

A process term is converted into standard form by using the explicit congruence rules and as a result of the associative, commutative and identity equational attributes. Subsequently, the scope of every unguarded occurrence of the new operator is extended to the outermost level and the operator `flatten` is used to state the nested `new` declarations as a flat declaration `new*`.

```
op flatten : Trm -> Trm .
eq flatten(new [x] P) = flatten(new* [x] P) .
eq flatten(new* [nl] new [x] P) = flatten(new* [nl x] P) .
eq flatten(new* [nl] new* [nl'] P) = flatten(new* [nl nl'] P) .
eq flatten(P) = P [owise] .
```

The reduction rules for HOπ in Fig. 7 (top) can then be stated as follows:

```
crl [FLAT] : P => P' if P' := flatten(P) /\ P =/= P' .
rl [SESS-COM1] : new* [x nl] (x{N} < V > . P | *(x{N}) (z) . Q | R) =>
                 new* [x nl] (P | [z := V] Q | R) .
rl [SESS-COM2] : new* [x nl] (*(x{N}) < V > . P | x{N} (z) . Q | R) =>
                 new* [x nl] (P | [z := V] Q | R) .
rl [SH-COM] : new* [x nl] (x(N) < V > . P | x(N) (z) . Q | R) =>
              new* [x nl] (P | [z := V] Q | R) .
rl [SESS-SEL1] : new* [x nl] ((x{N} << w . P) |
                              (*(x{N}) >> { (w : Q) ; CH }) | R) =>
                 new* [x nl] (P | Q | R) .
rl [SESS-SEL2] : new* [x nl] ((*(x{N}) << w . P) |
                              (x{N} >> { (w : Q) ; CH }) | R) =>
                 new* [x nl] (P | Q | R) .
rl [SH-SEL] : new* [x nl] ((x(N) << w . P) |
                           (x(N) >> { (w : Q) ; CH }) | R) =>
              new* [x nl] (P | Q | R) .
rl [APP] : new* [nl] (((lambda(x) . P) {a}) | Q) => new* [nl] ([x := a] P | Q) .
rl [REC] : new* [nl] ((u[x] . P) | Q) => new* [nl] (unfold(u[x] . P) | Q) .
```

We briefly discuss the rules. Rule `FLAT` rewrites a process term into a normalized process term. Rules `SESS-COM1`, `SESS-COM2`, and `SH-COM` correspond to the specification of reduction rules related to input-output communication. In particular, Rules `SESS-COM1` and `SESS-COM2` deal with communication by means of session names: they only differ in the usage of the co-name `*(x{N})` for the input or the output. Rule `SH-COM` handles communication through shared names. Rules `SESS-SEL1`, `SESS-SEL2`, and `SH-SEL` are specifications for the reduction rules concerned with labeled choices. As before, we distinguish between synchronizations on session names (Rules `SESS-SEL1` and `SESS-SEL2`) and synchronizations on shared names (Rule `SH-SEL`).

Rules `APP` and `REC` implement process application and recursion. The latter rule explicitly applies the structural congruence rule associated to recursive process. In this case, the auxiliary operator `unfold` replaces each occurrence of the variable `x` for the term `u[x].P`. We specify this congruence rule as a rewriting rule (rather than as an equation) to prevent infinite unfolding. This way, we only unfold process terms where the recursion process is at the top level. The operator `unfold` and the auxiliary operators `replace` and `replace-aux` are defined as follows:

```
op unfold : Trm -> Trm .
op replace : Qid Trm Trm -> Trm .
op replace-aux : Qid Trm Choiceset -> Choiceset .

eq unfold(u[x] . P) = replace(x, u[x] . P, P) .
eq unfold(P) = P [owise] .
eq replace(x, P, nil) = nil .
eq replace(x, P, new[y] Q) = new[y] replace(x, P, Q) .
eq replace(x, P, rec(x)) = P .
eq replace(x, P, (lambda(y) . Q) {a}) = (lambda(y) . replace(x, P, Q)) {a} .
```

$$\Theta_1 \div \emptyset = \Theta_1 \qquad \frac{u \notin dom(\Theta_1)}{\Theta_1 \div u = \Theta_1} \qquad \frac{\Theta_1 \div N = \Theta_2, u : \mathsf{end}}{\Theta_1 \div (N, u) = \Theta_2}$$

Fig. 11. Context difference for HOπ.

```
eq replace(x, P, G . Q) = G . replace(x, P, Q) .
eq replace(x, P, a >> {CH}) = a >> { replace-aux(x, P, CH) } .
eq replace(x, P, a << x . Q) = a << x . replace(x, P, Q) .
eq replace(x, P, u[y] . Q) = u[y] . replace(x, P, Q) .
eq replace(x, P, Q | R) = replace(x, P, Q) | replace(x, P, R) .
eq replace(x, P, Q) = Q [owise] .
eq replace-aux(x, P, empty) = empty .
eq replace-aux(x, P, (y : Q) ; CH) = (y : replace(x, P, Q)) ; replace-aux(x, P, CH) .
```

We can use our rewriting specification for HOπ to execute a sequence of reductions over a process term by using the Maude command `rew`. Moreover, we can use the `search` command to perform reachability analyses.

6.3. Algorithmic type checking

As mentioned above, to implement HOπ in Maude we must start by developing an algorithmic type checking system for HOπ. We are confronted with the same difficulties present in the case of sπ, namely the non-deterministic splitting operation, and the problem of guessing the type to be included in the context when dealing with bound names, as in Rules ResS, Res, Abs, and Rec (see Fig. 9).

Definitions Following Vasconcelos [30], we avoid the context splitting when type checking a complex process by considering judgments as a function such that, given an input context and a process (or value), type-checks the process (or value) and returns the unused part of the input context. The resulting output context can be passed on to type-check the next process in a parallel composition. We use judgments of the form

$$\Gamma; \Theta_1 \Vdash D \triangleright T; \Theta_2$$

where D denotes either a process or a value and $T = \diamond$ in the former case. Γ, Θ_1, and P denote the input to the algorithm, T stands for the resulting type, and Θ_2 denotes the residual context. For notational convenience, Θ merges the contexts Λ and Δ used in the typing system of Section 6.1.2, which admit the exchange principle. In the following, we refer to Θ as the linear context; elements of Θ are denoted $s : \theta$.

To avoid guessing types, we add explicit annotations to the scope restriction, the abstraction, and the recursive constructors. Thus, we write $(\nu s : S)P$, where the session name s will have type S in P whereas $\overline{s}$ will have type S', with S' being the dual type of S. Process $(\nu a : \langle S \rangle)P$ means that only $a : \langle S \rangle$ has to be included in the shared context, since $\overline{a} = a$. Given a set of names N, we write $\mu(X : N).P$ to specify that P uses the names in N with the corresponding type association in the shared and linear contexts. Both $\lambda x : S.\, P$ and $\lambda x : \langle S \rangle.\, P$ have the expected meaning.

To manage linear resources we use the auxiliary function $\div$. Given a context Θ, a (possibly empty) set of names N, and a channel u, then $\Theta \div u$ returns the context Θ with the channel u removed if $u : \mathsf{end}$ (see Fig. 11). This enables us to check that a linear channel is only removed from the output context when all its resources have been consumed and is of type end. Note that for any other type, the operation $\div$ is undefined, and then the rule itself would not be considered defined and could not be part of a valid typing derivation. Thus, in writing $\Gamma; \Theta_1 \Vdash D \triangleright T; \Theta_2$ we assume that both Θ_1 and Θ_2 are well-defined.

Typing rules Figs. 12 and 13 give the rules of the algorithmic type system for HOπ. Before discussing some of them, we introduce an auxiliary notation. Given a linear context Θ, the function `split` splits it into its higher-order and first-order parts. Given $\Theta = \Theta_1, \Theta_2$, we write

$$\mathtt{split}(\Theta) = \Theta_1 * \Theta_2$$

to denote that Θ_1 and Θ_2 are the higher-order and first-order fragments of Θ, respectively.

Rules A-Sess, A-Sh, A-LVar, and A-Nil allow any linear context to pass through the judgment. Rule A-RVar checks that Θ_1 is a subcontext of the input and then it outputs the remaining context. Here Θ_1 cannot contain higher order types; this is checked using the function `split`.

Rule A-Send has three premises. The first one checks that u has an output type in Θ_1. Then it takes Θ_2, the unused part of the first judgment together with $u : S$, to type-check V. The output Θ_3 of this premise is then used to type-check P, which produces an output Θ_4. The output of the rule is then $\Theta_4 \div u$; thus, if u consumed all the resources, i.e., its type in Θ_4 is end, the process is well typed. Note that $u : S$ is added to the input context of V since this is allowed according to Rule Send in Section 6.1; it is also important to remark that this does not produce any conflict with the type of the channel

$$\textsc{(A-Nil)}\quad \Gamma; \Theta_1 \Vdash 0; \Theta_1 \qquad \textsc{(A-Sess)}\quad \Gamma; \Theta_1, u : S \Vdash u \triangleright S; \Theta_1 \qquad \textsc{(A-Sh)}\quad \Gamma, u : U; \Theta_1 \Vdash u \triangleright U; \Theta_1$$

$$\textsc{(A-RVar)}\ \frac{\mathtt{split}(\Theta_1) = \emptyset * \Theta_1}{\Gamma, X : \Theta_1; \Theta_2, \Theta_1 \Vdash X; \Theta_2} \qquad \textsc{(A-LVar)}\quad \Gamma; \Theta_1, x : C \multimap \diamond \Vdash x \triangleright C \multimap \diamond; \Theta_1$$

$$\textsc{(A-SAbs)}\ \frac{\Gamma; \Theta_1, x : S \Vdash P; \Theta_2}{\Gamma; \Theta_1 \Vdash \lambda x : S. P; \Theta_2 \div x} \qquad \textsc{(A-ShAbs)}\ \frac{\Gamma, x : \langle U \rangle; \Theta_1 \Vdash P; \Theta_2}{\Gamma; \Theta_1 \Vdash \lambda x : \langle U \rangle. P; \Theta_2}$$

$$\textsc{(A-Rec)}\ \frac{\Gamma, X : \Theta_1; \Theta_1, \Theta_2 \Vdash P; \Theta_2 \quad N \subseteq dom(\Theta_1) \quad \mathtt{split}(\Theta_1) = \emptyset * \Theta_1}{\Gamma; \Theta_1, \Theta_2 \Vdash \mu(X : N).P; \Theta_2}$$

$$\textsc{(A-Send)}\ \frac{\Gamma; \Theta_1 \Vdash u \triangleright\ !\langle U \rangle.S; \Theta_2 \quad \Gamma; \Theta_2, u : S \Vdash V \triangleright U; \Theta_3 \quad \Gamma; \Theta_3 \Vdash P; \Theta_4}{\Gamma; \Theta_1 \Vdash u!\langle V \rangle.P; \Theta_4 \div u}$$

$$\textsc{(A-SRcv)}\ \frac{\Gamma; \Theta_1 \Vdash u \triangleright\ ?(U).S; \Theta_2 \quad \Gamma; \Theta_2, u : S, x : U \Vdash P; \Theta_3}{\Gamma; \Theta_1 \Vdash u?(x).P; \Theta_3 \div \{u, x\}}$$

$$\textsc{(A-ShRcv)}\ \frac{\Gamma, x : \langle U \rangle; \Theta_1 \Vdash u \triangleright\ ?(U).S; \Theta_2 \quad \Gamma, x : \langle U \rangle; \Theta_2, u : S \Vdash P; \Theta_3}{\Gamma; \Theta_1 \Vdash u?(x).P; \Theta_3 \div u}$$

$$\textsc{(A-App)}\ \frac{\Gamma; \Theta_1 \Vdash V \triangleright C \rightsquigarrow \diamond; \Theta_2 \quad \Gamma; \Theta_4 \Vdash u \triangleright C; \Theta_5 \quad \mathtt{split}(\Theta_2) = \Theta_3 * \Theta_4}{\Gamma; \Theta_1 \Vdash V\, u; \Theta_3, \Theta_5}$$

$$\textsc{(A-Sel)}\ \frac{\Gamma; \Theta_1 \Vdash u \triangleright \oplus\{l_i : S_i\}_{i \in I}; \Theta_2 \quad \Gamma; \Theta_2, u : S_j \Vdash P; \Theta_3}{\Gamma; \Theta_1, u : \oplus\{l_i : S_i\}_{i \in I} \Vdash u \triangleleft l_j.P; \Theta_3 \div u}$$

$$\textsc{(A-Bra)}\ \frac{\Gamma; \Theta_1 \Vdash u \triangleright \&\{l_i : S_i\}_{i \in I} \Theta_2 \quad \Gamma; \Theta_2, u : S_i \Vdash P_i; \Theta_3^i \quad \forall i, j\ \mathtt{NotEnd}(\Theta_3^i) = \mathtt{NotEnd}(\Theta_3^j)}{\Gamma; \Theta_1 \Vdash u \triangleright \{l_i : P_i\}_{i \in I}; \cap_{i \in I} \Theta_3^i \div u}$$

Fig. 12. Algorithmic type checking rules for $\mathsf{HO}\pi$ (Part 1).

u, since given the case that the first premise is verified, u is not part of the output Θ_2 (by Rule A-Sess). If $u : S$ is not used to type-check V, then $u : S$ would be in the output context Θ_3, hence it would be available to type-check P.

Example 6.2. We give an algorithmic type derivation for the process $u!\langle x \rangle.u!\langle y \rangle.\mathbf{0}$, with $\Gamma = \emptyset$ and $\Theta_1 = x : S, y : S', u : !\langle S \rangle.!\langle S' \rangle.\mathsf{end}$, where S and S' are session types.

$$\emptyset; x : S,\ y : S',\ u : !\langle S \rangle.!\langle S' \rangle.\mathsf{end} \Vdash u \triangleright !\langle S \rangle.!\langle S' \rangle.\mathsf{end};\ x : S,\ y : S' \tag{3}$$

$$\emptyset; x : S,\ y : S',\ u : !\langle S' \rangle.\mathsf{end} \Vdash x \triangleright S;\ u : !\langle S' \rangle.\mathsf{end},\ y : S' \tag{4}$$

$$\emptyset; u : !\langle S' \rangle.\mathsf{end},\ y : S' \Vdash u \triangleright !\langle S' \rangle.\mathsf{end};\ y : S' \tag{5}$$

$$\emptyset; u : \mathsf{end},\ y : S' \Vdash y \triangleright S';\ u : \mathsf{end} \tag{6}$$

$$\emptyset; u : \mathsf{end} \Vdash \mathbf{0};\ u : \mathsf{end} \tag{7}$$

Then we have:

$$\frac{(5) \quad (6) \quad (7)}{\emptyset; u : !\langle S' \rangle.\mathsf{end},\ y : S' \Vdash u!\langle y \rangle.\mathbf{0};\ \{u : \mathsf{end}\} \div u = \emptyset} \tag{8}$$

$$\frac{(3) \quad (4) \quad (8)}{\emptyset; x : S,\ y : S',\ u : !\langle S \rangle.!\langle S' \rangle.\mathsf{end} \Vdash u!\langle x \rangle.u!\langle y \rangle.\mathbf{0};\ \emptyset \div u = \emptyset} \tag{9}$$

$$\text{(A-PAR)}\ \frac{\Gamma;\Theta_1 \Vdash P;\Theta_2 \quad \Gamma;\Theta_2 \Vdash Q;\Theta_3}{\Gamma;\Theta_1 \Vdash P \mid Q;\Theta_3} \qquad \text{(A-PROM)}\ \frac{\Gamma;\Theta_1 \Vdash V \triangleright C \multimap \diamond;\Theta_1}{\Gamma;\Theta_1 \Vdash V \triangleright C \rightarrow \diamond;\Theta_1} \qquad \text{(A-EPROM)}\ \frac{\Gamma;\Theta_1, x:C \multimap \diamond \Vdash P \triangleright \diamond;\Theta_2}{\Gamma, x:C \rightarrow \diamond;\Theta_1 \Vdash P \triangleright \diamond;\Theta_2}$$

$$\text{(A-RESS)}\ \frac{\Gamma;\Theta_1, s_1:S_1, \bar{s_2}:S_2 \Vdash P;\Theta_2 \quad S_1\ \mathsf{dual}\ S_2}{\Gamma;\Theta_1 \Vdash (\nu\, s:S_1)P;\Theta_2 \div \{s_1,s_2\}} \qquad \text{(A-RES)}\ \frac{\Gamma, a:\langle S\rangle;\Theta_1 \Vdash P;\Theta_2}{\Gamma;\Theta_1 \Vdash (\nu\, a:\langle S\rangle)P;\Theta_2}$$

$$\text{(A-SACC)}\ \frac{\Gamma', u:\langle U\rangle;\Theta_1', x:C \Vdash u \triangleright \langle U\rangle;\Theta_1', x:C \quad \Gamma', u:\langle U\rangle, \Theta_1', x:C \Vdash P;\Theta_2}{\Gamma;\Theta_1' \Vdash u?(x).P;\Theta_2 \div x}$$

$$\text{(A-SHACC)}\ \frac{\Gamma', u:\langle U\rangle, x:\langle C\rangle;\Theta_1 \Vdash u \triangleright \langle U\rangle;\Theta_1 \quad \Gamma', u:\langle U\rangle, x:\langle C\rangle;\Theta_1 \Vdash P;\Theta_2}{\Gamma', u:\langle U\rangle;\Theta_1 \Vdash u?(x).P;\Theta_2}$$

$$\text{(A-REQ)}\ \frac{\Gamma', u:\langle U\rangle;\Theta_1 \Vdash u \triangleright \langle U\rangle;\Theta_1 \quad \Gamma', u:\langle U\rangle;\Theta_3 \Vdash V \triangleright U;\Theta_4 \quad \Gamma', u:\langle U\rangle;\Theta_2,\Theta_4 \Vdash P;\Theta_5 \quad \mathtt{split}(\Theta_1) = \Theta_2 * \Theta_3}{\Gamma', u:\langle U\rangle;\Theta_1 \Vdash u!\langle V\rangle.P;\Theta_5}$$

Fig. 13. Algorithmic type checking rules for $\mathsf{HO}\pi$ (Part 2).

In a bottom-up reading, judgment (3) verifies that u has an output type. Judgment (4) takes the output of the previous one together with the continuation type of u, $!\langle S'\rangle.\mathsf{end}$, to type-check the value x. Again, the previous output is taken as the context input to type-check the continuation process $u!\langle y\rangle.\mathbf{0}$ in (8). Because the continuation is itself an output process, the same procedure is done again in (5), (6), and (7). Note that in judgment (7) the output context is $u:\mathsf{end}$, and then by Rule A-SEND and definition of $\div$, the output context of (8) is $\{u:\mathsf{end}\} \div u = \emptyset$; the same reasoning applies to (8) and (9) to obtain the output $\emptyset \div u = \emptyset$.

On the other hand, it is easy to see that taking as input, e.g., $\Theta_1' = x:S, y:S', w:\hat{S}, u:!\langle S\rangle.!\langle S'\rangle.!\langle \hat{S}\rangle.\mathsf{end}$, produces an undefined output, since u does not consume all the resources. Similarly, an input $\Theta_1'' = x:S, u:!\langle S\rangle.\mathsf{end}$ would not successfully type-check the process.

Rule Rcv in Fig. 9 is broken down in two different algorithmic rules: Rule A-SRcv and Rule A-SHRcv. The former is used when x is typed as either session or higher-order types; the latter when x has shared type. Both rules first check that the channel u has an input type, and then take the resulting output to type-check P together with the type continuation of u; to type-check P, $x:U$ is also included either in Γ or Θ depending on U. Rule A-SRcv checks that both u and x have consumed all the resources and then removes them from the output context. Rule A-SHRcv checks that u has consumed all the resources (its type is end) and it also removes u and x from the respective context.

Similar to Rule Rcv, Rule ABS in Fig. 9 is broken into two algorithmic rules: A-SABS and A-SHABS. The former is used when x has a session type and the latter when x has a shared type. The two rules check that P is typable with the input context. Rule A-SABS checks that x has consumed all the resources and then removes it from the output context.

Rule A-SEL first checks that u has type $\oplus\{l_i:S_i\}_{i\in I}$, and then type-checks P using the remaining portion of Θ_1 and the continuation type $u:S_j$, resulting with an output Θ_3; the rule then checks that $u:\mathsf{end}$. Rule A-BRA first checks that the channel u has a branch type, and then uses the remaining (linear) context and the respective typing $u:S_j$ to type-check each branch. The procedure succeeds if all the output contexts Θ_3^i are the same or they only differ in the channels with type end. This is checked by the following function

$$\mathtt{NotEnd}(\Theta) = \{u:T \mid u \in dom(\Theta) \wedge T \neq \mathsf{end}\}$$

The output of the rule is $\cap_{i\in I}\Theta_3^i \div u$: hence, it contains the common unused part, as well as channels of type end not used in any branch.

Rule A-REC type-checks P provided that Θ_1 is a subset of the linear input context that does not contain higher-order types. To check this, we use the auxiliary function $\mathtt{split}$. Rule A-APP type-checks V and then splits the remaining context Θ_2 in the higher-order part Θ_3 and the first-order part Θ_4. Context Θ_4 is used to type-check the channel u. The output of the rule is Θ_3, Θ_5, since Θ_3 might be necessary to type check other processes.

The following theorem states the correspondence between $\mathsf{HO}\pi$ and the algorithmic system. Because we added annotations, this language is not exactly the same presented in Section 6.1. To remove them, we give a function $\mathtt{erase}$ inductively defined as:

$$\mathtt{erase}(\lambda x:U.\,P) = \lambda x.\,\mathtt{erase}(P)$$

$$\mathtt{erase}((\nu s : S)P) = (\nu s)\mathtt{erase}(P)$$
$$\mathtt{erase}((\nu a : \langle S \rangle)P) = (\nu a)\mathtt{erase}(P)$$
$$\mu(X : N).P = \mu X.\mathtt{erase}(P)$$

and as a homomorphism for the other constructs.

In addition, given a context Θ, $\mathsf{End}(\Theta)$ states that for all $u \in dom(\Theta)$, $u : \mathsf{end}$.

Theorem 6.2 *(Correctness of algorithmic type checking). Given contexts Γ, Λ, Δ, a process or abstraction D, a type T, and w (name or variable) we have the following:*

- $\Gamma; \Lambda; \Delta \vdash w \triangleright T$ *iff* $\Gamma; \Theta_1 \Vdash w \triangleright T; \emptyset$, *where* $\Theta_1 = \Lambda, \Delta$.
- $\Gamma; \Lambda; \Delta \vdash \mathtt{erase}(D) \triangleright T$ *(where either $T = \diamond$ or $T = C \rightsquigarrow \diamond$) iff* $\Gamma; \Theta_1 \Vdash D \triangleright T; \Theta_2$ *where* $\Theta_2 = \emptyset$ *or* $\mathsf{End}(\Theta_2)$*, and* $\Theta_1 = \Lambda, \Delta$.

Proof. See Appendix D. □

6.3.1. Implementation in Maude

In this section we present a Maude implementation of the algorithmic type checking system given in the previous section. Because we encode types following the same insights presented for sπ in Section 4, in the following we only highlight important modifications.

```
sorts Type SessionType SharedType LinearType Context SessionContext
TupleTypeContext ChoiceT ChoiceTset .
subsort SessionType < LinearType < Type .
subsort SharedType < Type .
subsort ChoiceT < ChoiceTset .
subsort SessionContext < Context .
op ?_._ : Type SessionType -> SessionType .
op !_._ : Type SessionType -> SessionType .
op +{_} : ChoiceTset -> SessionType .
op &{_} : ChoiceTset -> SessionType .
op u [_] _ : Qid SessionType -> SessionType .
op var : Qid -> SessionType .
op end : -> SessionType .
op lin proc(_,_) : Type Type -> LinearType .
op un proc(_,_) : Type Type -> SharedType .
op [_]: Type -> SharedType .
op <> : -> Type .
ops nil invalid-context : -> Context .
op nil : -> SessionContext .
op _:_ : SharedName SharedType -> Context [ctor] .
op _:_ : Variable SharedType -> Context [ctor] .
op _:_ : Variable SessionContext -> Context [ctor] .
op _,_ : Context Context -> Context [ctor assoc comm id: nil] .
op _:_ : SessionName LinearType -> SessionContext [ctor] .
op _:_ : Variable LinearType -> SessionContext [ctor] .
op _,_ : SessionContext SessionContext -> SessionContext [ctor assoc comm id: nil] .
```

Again each production rule is given as a specific constructor. General contexts are associated to sort `Context`, which deals with shared contexts. We use the subsort relation to specify that linear contexts are a specific kind of context, containing only session types. Type `<>` denotes the process type. Types $C \multimap \diamond$ and $C \rightarrow \diamond$ are given as the constructors `lin proc(_,_)` and `un proc(_,_)` respectively. This way, e.g., a value $\lambda x : S.\, P$ will be typed as `lin proc(S,<>)`.

The algorithm for type checking values follows the same intuitions as the one for sπ. Function `type-value` receives an instance of the sort `Context`, an instance of the sort `SessionContext` and a value, which could be a name, a variable or an abstraction. It returns an instance of the sort `TupleTypeContext`, which has `[_ _]` as the only constructor.

This constructor groups a type and a (output) context. Rules `A-SESS` and `A-SH` first check whether the channel is recursive, to perform a step of the `unfold` operation when necessary. In the equation for Rule `A-SABS`, `S` ranges over terms of sort `SessionType`, thus avoiding nested abstractions, since `lin proc(T,T')` is of sort `LinearType`. It also calls the function `type-term` (explained below) to type-check the abstracted process.

```
op type-value : Context SessionContext Value -> TupleTypeContext .
eq type-value(C, ((u : S), SC), u) = [SC unfold(S)] .                    --- [A-SESS]
eq type-value(C, ((u : L), SC), u) = [SC L] .                            --- [A-SESS]
eq type-value(((n : [T]), C), SC, n) = [SC [unfold(T)]] .                --- [A-SH]
eq type-value(((n : (un proc(T, T'))), C), SC, n) = [SC (un proc(T, T'))] .    --- [A-SH]
eq type-value(((val(x) : (un proc(T, T'))), C), SC, val(x)) =
```

```
op type-term : Context SessionContext Trm -> TupleTypeContext .
ceq type-term(C, SC, u[x : CX] . P) = type-term((C, val(x) : SC'), SC, P)          --- [A-REC]
  if SC' := get-context(CX, SC) /\
     SC1 ; SC2 := split(SC', nil ; nil) /\ SC1 == nil .
ceq type-term((C, (val(x) : SC')), (SC', SC), rec(x)) = [SC <>]                     --- [A-RVAR]
  if SC1 ; SC2 := split(SC', nil ; nil) /\ SC1 == nil .
ceq type-term(C, SC, u < V > . P) = [(SC' / u) <>]                                  --- [A-SEND]
  if [SC1 (! T . S)] := type-value(C, SC, u) /\
     [SC2 T'] := type-value(C, (SC1 , u : S) , V) /\
     [SC' <>] := type-term(C,(SC2), P) /\ equal(T, T') /\
     (SC' / u) =/= invalid-context .
ceq type-term(C, SC, u(y) . P) = [((SC' / u) / y{0}) <>]                            --- [A-SRCV]
  if [SC1 (? L . S)] := type-value(C, SC, u) /\
     [SC' <>] := type-term(C, (SC1, (y{0} : L), (u : S)), P) /\
     ((SC' / u) / y{0}) =/= invalid-context .
ceq type-term(C, SC, V{a}) = [(SC', SC2) <>]                                        --- [A-APP]
  if [SC1 (lin proc(T, <>))] := type-value(C, SC, V) /\
     SC2 ; SC3 := split(SC1, nil ; nil) /\ [SC' T] := type-value(C, SC3, a) .
ceq type-term(C, SC, V{a}) = [(SC', SC1) <>]                                        --- [A-APP]
  if SC1 ; SC2 := split(SC, nil ; nil) /\
    [SC (un proc(T, <>))] := type-value(C, SC, V) /\
    [SC' T] := type-value(C, SC2, a) .
ceq type-term(C, SC, n < V > . P) = [SC' <>]                                        --- [A-REQ]
    if SC1 ; SC2 := split(SC, nil ; nil) /\ [SC [T]] := type-value(C, SC, n) /\
       [SC3 T'] := type-value(C, SC2, V) /\
       [SC' <>] := type-term(C, (SC1, SC3), P) /\ equal(T, T') .
ceq type-term(C, SC, n(y) . P) = [(SC' / 'y'{0}) <>]
    if [nil [S]] := type-value(C, nil, n) /\
       [SC' <>] := type-term(C, (SC, (y{0} : S)), P) .                              --- [A-SACC]
ceq type-term((C, (val(x) : (un proc(T, <>)))), SC, P) = [SC' <>]
    if [SC' <>] := type-term(C, (SC, (val(x) : (lin proc(T, <>)))), P).             --- [A-EPROM]
eq type-term(C, SC, P) = ill-typed [owise] .
```

Fig. 14. Type checking for processes in Maude.

```
   [SC (un proc(T, T'))] .                                                        --- [A-SH]
eq type-value(C, ((val(x) : lin proc(T, T')), SC), val(x)) =
   [SC (lin proc(T, T'))] .                                                       --- [A-LVAR]
ceq type-value(C, SC, lambda(x : S) . P) = [(SC' / x{0}) (lin proc(S, <>))]
    if [SC' <>] := type-term(C, (SC, (x{0} : S)) , P) .                          --- [A-SABS]
ceq type-value(C, SC, lambda(x : U) . P) = [SC' (lin proc(U, <>))]
    if [SC' <>] := type-term((C, (x(0) : U)), SC, P) .                           --- [A-SHABS]
```

The algorithm for type checking processes is implemented as the function `type-term`, which also receives an instance of the sort `Context`, a linear context (an instance of sort `SessionContext`), and an instance of sort `Trm`. Like function `type-value`, `type-term` produces an instance of the sort `TupleTypeContext`.

Fig. 14 presents the implementation of some of the rules for the algorithmic type system; the remaining cases are either similar to the ones presented in Section 4 or similar to the ones presented here. Each equation in Fig. 14 corresponds to a typing rule in Fig. 12. The operator / stands for the operator $\div$ in Fig. 11.

Rule `A-REC` uses an auxiliary function `get-context` that receives a set of names and a context, and returns the restriction of such a context to the provided set of names. In this way, the context to be assigned to `val(x)` is extracted. It also uses the auxiliary function `split`, which is the Maude implementation for the function `split` used in the previous section. Accordingly, `split` takes as parameter a context `SC` and returns an instance of the sort `SessionContextPair` of the form `SC1;SC2`, where `SC1` is the subcontext of `SC` that contains the higher-order names, and `SC2` contains the remainder context.

Rule `A-APP` has two cases, depending on the possible types for the value `V`: `lin proc(T,<>)` or `un proc(T,<>)`. In both cases, the type of the name `a` is checked by using the subcontext where there are no higher-order types. Such a context is obtained using function `split`. If the type of the abstraction value `V` is linear then function `split` is applied over the linear context that is returned from type checking `V`; otherwise, if the value `V` is a shared abstraction, then function `split` takes the original context `SC`.

Rule `A-REQ` is defined similarly. In order to type-check the sent value `V`, it is passed to the first-order part of the linear context, which is obtained by applying the function `split` over the input linear context `SC`. The additional aspects of that rule are as in the algorithm for sπ for Rule `A-OUT`.

Finally, the Maude implementation for the rules `A-SEND` and `A-SRCV`, concerned to the input and output in session names, are quite similar to the implementation of the analogous rules in the sπ calculus. In both rules, the subject name `u` is removed from the output context `SC'` through the operator /. Nevertheless, it is required that the type of `u` is `end`; otherwise the operator / produces as result the constant `invalid-context` and the type checking is not successful.

6.3.2. Example

We type-check the mobile process from the hotel booking example discussed in Section 6.1.3. The implementation in Maude is as follows:

```
op r : -> Trm .
ops u s : -> SessionType .
eq u = ! Quote . ? Quote . +{('accept' : ! NNat . end ) ; ('reject' : end)} .
eq s = ! Quote . &{('accept' : end) ; ('reject' : end)} .
eq r = 'x'{0} < Room > . 'x'{0}('z') . 'y'{0} < Room > .
          ('y'{0} >> {('accept' : 'x'{0} << 'accept' . 'x'{0}< 24 > . nil) ;
                     ('reject' : 'x'{0} << 'reject' . nil)}) .
```

For the sake of convenience, we include types `NNaT` and `Quote` as basic types; we type-check this process by executing `type-value(nil, 'y'{0} : s, lambda('x' : u). r)` and we obtain the following result:

```
reduce in TEST : type-value(nil, 'y'{0} : s, lambda('x' : u). r) .
rewrites: 105 in 0~ms cpu (0~ms real) (~ rewrites/second)
result TupleTypeContext: [nil lin proc(! Quote . ? Quote .
                            +{('accept' : ! NNat . end) ; 'reject' : end},<>)]
```

Note that we run `type-value` since the term to be checked is an abstraction. In addition, it is important to highlight that the result corresponds to an instance of the sort `TupleTypeContext` of the form `[nil lin proc(T,<>)]`, where the output (linear) context `nil` tells us that all the information provided in the input context was consumed and the rest corresponds to the (expected) type of the value.

Similarly, the process Client in Fig. 10 is implemented as follows:

```
ops l1 l2 : -> Value .
ops p q : -> Trm .
ops t1 t2 u s : -> SessionType .
eq u = ! Quote . ? Quote . +{('accept' : ! NNat . end ) ; ('reject' : end)} .
eq s = ! Quote . &{('accept' : end) ; ('reject' : end)} .
eq t1 = ! (lin proc(u,<>)) . end .
eq t2 = ! (lin proc(u,<>)) . end .
eq l1 = lambda('x' : u ) . 'x'{0} < Room > . 'x'{0}('z') . 'h1'{0}< Room > .
            ('h1'{0} >> {('accept' : 'x'{0} << 'accept' . 'x'{0} < 24 > . nil) ;
                         ('reject' : 'x'{0} << 'reject' . nil)}) .        ---[Mobile Process 1]
eq l2 = lambda('x' : u ) . 'x'{0} < Room > . 'x'{0}('z') . 'h2'{0}< Room > .
            ('h2'{0} >> {('accept' : 'x'{0} << 'accept' . 'x'{0} < 24 > . nil) ;
                         ('reject' : 'x'{0} << 'reject' . nil)}) .        ---[Mobile Process 2]
eq p = 's1'{0}< l1 > . 's2'{0}< l2 > . nil .
eq q = *('h1'{0})('z') . *('h2'{0})('w') .
      ('c'{0} >> {('XLessThanY' : *('h1'{0}) << 'accept' . *('h2'{0}) << 'reject' . nil) ;
                  ('XGreaterThanY' : *('h1'{0}) << 'accept' . *('h2'{0}) << 'reject' . nil) }) .
```

Here `l1` and `l2` refer to the mobile processes to be sent to the hotels. Terms `p` and `q` refer to the left-hand side and right-hand side processes of the parallel composition. Thus, by executing

```
reduce in TEST : type-term(nil, ('s1'{0} : t1),
                               ('c'{0} : &{('XGreaterThanY' : end) ; 'XLessThanY' : end}),
                               's2'{0} : t2,
                                  new['h1' : s]new['h2' : s]p | q) .
rewrites: 671 in 0~ms cpu (0~ms real) (~ rewrites/second)
result TupleTypeContext: [nil <>]
```

we obtain the expected result, i.e., a `TupleTypeContext [nil <>]` which indicates that the process was successfully type-checked.

Finally, if we had defined mistakenly `u` as

```
eq u = ! NNat . ? Quote . +{('accept' :! NNat . end ) ;('reject' : end)} .
```

We would have obtained an `ill-typed` message:

```
reduce in TEST : type-value(nil, 'y'{0} : s, lambda('x' : u). p) .
rewrites: 89 in 0~ms cpu (0~ms real) (~ rewrites/second)
result TupleTypeContext: ill-typed
```

6.3.3. Executable type soundness

Theorem 6.1 ensures that the type system for HOπ is *sound*: well-typed processes are guaranteed to always respect the ascribed session protocols. Also, the algorithmic type system was proven to be correct (Theorem 6.2). Therefore, as in the case of sπ, we can also integrate both the rewriting specification of the operational semantics and the implementation of the algorithmic type checking for HOπ. Hence, we only execute well-typed processes. Thus, we adapt the definition of the functions `well-typed` and `erase` to HOπ. We remark that function `well-typed` checks whether a process does not have typing errors:

```
op well-typed : Trm -> Bool .
ceq well-typed(P) = true if [SC <>] := type-term(nil, nil, P) /\ consumed(SC) .
eq well-typed(P) = false [owise] .
```

This function applies the algorithm for type checking `type-term` over a process `P` and returns `true` when type-checking succeeds. This occurs when the resulting type is `<>` and the output context `SC` is *consumed*, i.e. all of its linear names have type end. Function `erase`, given above, inductively removes the annotations in restriction, abstraction and recursive subprocesses.

We add one rewriting rule to enable the execution of annotated processes, i.e. processes using $(\nu n : T)P$, $\lambda x : T.\, P$ and $\mu(X : N).P$ instead of $(\nu n)P$, $\lambda x.\, P$ and $\mu X.P$, respectively.

```
rl [TYPED] : new [x : T] P => if well-typed(new [x : T] P)
                              then erase(new [x] P) else ill-typed-process fi .
```

Thereby, if process `new [x : T] P` is well-typed then it is rewritten as an equivalent process in which each annotation is dropped through the function `erase`. Otherwise, the process `new [x : T] P` is rewritten as the constant `ill-typed-process`.

6.4. Deadlock-freedom

Similar to sπ, the type system for HOπ does not exclude locks and deadlocks. We extend the approach for their detection, following Section 5. We focus on the main details specific to HOπ processes, and omit the aspects that are similar to those for sπ.

Deadlock and lock freedom are formulated following the notion of pending communication given in [20]. We adapt the notion of reduction contexts given in Section 5 as follows:

$$\mathcal{C} ::= [\,] \;\mid\; (\mathcal{C} \mid P) \;\mid\; (\nu x)\mathcal{C}$$

Likewise, we adapt the definition of the auxiliary predicates in, out, sync, and wait given in Section 5 to the HOπ syntax:

$$\mathsf{in}(u, P) \overset{\text{def}}{\iff} P \equiv \mathcal{C}[u?(x).Q\,] \wedge u \notin \mathrm{bv}(\mathcal{C})$$

$$\mathsf{out}(u, P) \overset{\text{def}}{\iff} P \equiv \mathcal{C}[u!\langle V\rangle.Q\,] \wedge u \notin \mathrm{bv}(\mathcal{C})$$

$$\mathsf{sync}(u, P) \overset{\text{def}}{\iff} \mathsf{in}(u, P) \wedge \mathsf{out}(\overline{u}, P)$$

$$\mathsf{wait}(u, P) \overset{\text{def}}{\iff} (\mathsf{in}(u, P) \vee \mathsf{out}(\overline{u}, P)) \wedge \neg\mathsf{sync}(u, P)$$

We remark that predicate in (resp. out) holds if the process P can input (resp. output) in the free name u. Similarly, predicate sync holds whether a synchronization through name u (and its co-name $\overline{u}$) is immediately possible. Predicate wait holds when there is an output or input in a free name u but an immediate synchronization is not possible.

With these elements in mind, we say process P is

- *deadlock free* if for every Q such that $P \longrightarrow^* (\nu x_1)(\nu x_2)\dots(\nu x_n)Q \not\longrightarrow$ it holds that $\neg\mathsf{wait}(x_i, Q)$ for every x_i.
- *lock free* if for every Q such that $P \longrightarrow^* (\nu x_1)(\nu x_2)\dots(\nu x_n)Q$ and $\mathsf{wait}(x_i, Q)$ there exists R such that $Q \longrightarrow^* R$ and $\mathsf{sync}(x_i, R)$ hold.

We represent the auxiliary predicates in, out, wait and sync as functions in Maude as follows:

```
ops in out : Chan Trm -> Bool .
ops sync wait : Chan Trm -> Bool .
op wait-aux : QidSet Trm -> Bool .

eq in(a, a(x) . P | R) = true .
eq in(a, P) = false [owise] .
eq out(a, a < V > . P | R) = true .
eq out(a, P) = false [owise] .
```

```
eq sync(a, P) = in(a, P) and out(*(a), P) .
eq wait(a, P) = (in(a, P) or out(*(a), P)) and not sync(a, P) .
eq wait-aux(mt, P) = false .
eq wait-aux(x nl, P) = wait(x{0}, P) or wait(*(x{0}), P) or wait-aux(nl, P) .
```

Function `wait-aux` corresponds to the extension of function `wait` to a set of variables.

Now, as in Section 5, the checking of the deadlock free property is essentially a reachability problem. In this way, it is necessary to determine whether there is not a reachable final process with an input or output pending for synchronization. As before, we can check the deadlock free property by leveraging the Maude reachability tool `search`. Given a process term `init` to be checked, in order to verify the deadlock free property, we write:

```
search init =>! new* [nl:QidSet] P:Trm such that wait-aux(nl:QidSet, P:Trm) .
```

It is not possible to check the lock property by using the reachability tool since the verification of this property requires the checking of some intermediate states. As in Section 5, this property corresponds to a formula in CTL, which is not supported by Maude. For this reason, we adapt the LTL formula in (2) to predicates for the HOπ calculus. Given an HOπ process $(\nu\ x_1)\dots(\nu\ x_n)P$, the corresponding LTL formula is as follows:

$$\bigwedge_{(x_i)\in vars(P)} \Box(\Diamond \mathsf{wait}(x_i, Q) \longrightarrow \Diamond \mathsf{sync}(x_i, R)) \tag{10}$$

As before, this formula is sound and but not complete—completeness is only guaranteed for processes without the recursion operator. We can use the LTL model checker to verify if a process is lock free. In order to describe the LTL formula, we need the predicates `pwait` and `psync`, which are defined next. Additionally, we give an alternative definition for the auxiliary function `build-lock-formula`, now adapted to HOπ.

```
ops pwait psync : Chan -> Prop [ctor] .

eq new* [x nl] P |= pwait(x{0}) = wait(x{0}, P) or wait(*(x{0}), P) .
eq new* [x nl] P |= pwait(x(0)) = wait(x(0), P) .
eq new* [x nl] P |= psync(x{0}) = sync(x{0}, P) or sync(*(x{0}), P) .
eq new* [x nl] P |= psync(x(0)) = sync(x(0), P) .

op build-lock-formula : QidSet -> Formula .
eq build-lock-formula(mt) = True .
eq build-lock-formula(x nl) = [] (<> pwait(x{0}) -> <> psync(x{0})) /\
                                 build-lock-formula(nl) .
```

Then, given the process term `init` and the associated names `vars`, the model checker can be used as follows:

```
red modelCheck(init, build-lock-formula(vars)) .
```

7. Closing remarks

Related work Maude has been successfully used for the analysis of specifications in different models of concurrency, including process calculi (such as CCS [34] and the π-calculus [28]) and Petri Nets [24]. Our work builds upon and extends this line of developments.

To our knowledge, we are the first to represent session-typed π-calculi using Maude. Prior works have used rewriting logic to investigate the operational semantics for variants of the π-calculus. Viry [33,32] defines the reduction semantics of a synchronous π-calculus as a rewrite theory, which is implemented in ELAN. The work of Thati et al. [28] considers an untyped, asynchronous π-calculus, whose labeled transition semantics is implemented as a rewrite theory, which is used to formalize an associated may-testing preorder. The work of Pitsiladis and Stefaneas [22] concerns a typed process calculus but in a different context, in which types are used to enforce privacy properties. Their work gives a Maude implementation of the labeled transition semantics of a privacy-oriented variant of the π-calculus and a Maude implementation of its associated type system, which is implemented as a membership equational theory.

Bartoletti et al. [1,2] rely on Maude in their study of formal languages for the analysis and verification of contract-oriented systems. In their setting, contracts describe behavior that is required and offered by services; they model these contracts as processes in a process calculus called CO_2. There are high-level similarities between contracts in CO_2 and session-typed processes in sπ and HOπ; arguably the key distinguishing aspect between them is that in the contract-oriented setting agents can behave *dishonestly*, i.e., they can behave differently from their declared contracts, whereas session-typed processes are assumed to always stick to their ascribed protocol. This makes enforcing honesty (and identifying dishonest agents) a key issue for correctness. Similarly to our developments, the authors formalize CO_2 in Maude and develop a model checking technique for soundly approximating honesty.

One salient aspect of our work is the use of model checking to (soundly) characterize deadlock and lock freedom properties. Ensuring deadlock-freedom using type systems based on sessions is known to be challenging, especially in the presence of useful constructs like *delegation* (session passing) and *session interleaving* (as in, e.g., a single process that implements two or more sessions); the central issue is how to detect and rule out circular dependencies between different sessions. A number of previous works have developed sophisticated static approaches; see, for instance [3,5,9,29,31]. In contrast, our work develops a complementary approach based on model checking of LTL properties, following the formalization given by Padovani [20].

Conclusion Session types were introduced in the late 90s [15], and have been widely studied since then. In this paper, we have developed implementations of session-based concurrency in Maude. To our knowledge, we are the first to consider Maude as a target system for session-based process languages. We first considered sπ, a session-typed π-calculus proposed by Vasconcelos in [30]. We developed an executable specification of its operational semantics and associated algorithmic type-checking system. We integrated both specifications closely following his formulation. Because typing in [30] does not exclude deadlocks, we leverage built-in tools in Maude and executable specifications to detect well-typed dead-locked processes. We then moved to consider HOπ, a higher-order session π-calculus, which incorporates the passing of abstractions within session protocols. We showed how our approach to executable specifications, type checking, and deadlock detection in Maude applies also to HOπ. This required developing a new algorithmic type-checking system for HOπ. In our view, these developments establish a promising starting point to the automated analysis of message-passing concurrency specifications.

Future work Our work unlocks many exciting directions for future work; we briefly discuss some of them. We intend to adapt our equational theories to leverage the confluence checker tool available in Maude. We also plan to extend our executable specifications to accommodate *subtyping* [12,11] and to analyze processes that implement *multiparty session types*, as formalized in [27]. Likewise, we plan to consider connections between our executable implementations and type systems derived from the Curry-Howard correspondence between session types and linear logic [8,35]. More in general, it would be relevant to compare the expressivity and features of our Maude developments against existing libraries and tools for session types (see, e.g., [21,16]).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We are grateful to Camilo Rocha for his useful suggestions on this work, to the WRLA'22 attendees for constructive remarks, and to the anonymous reviewers of WRLA'22 and of the present paper for their careful reading and useful suggestions, which led to substantial improvements.

This work has been partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software). Carlos Ramírez has been partially supported by the Minciencias project PROMUEVA (BPIN 2021000100160). Juan C. Jaramillo has been partially supported by the Minciencias Call for PhDs Abroad No. 906.

Appendix A. Appendix to §3: Operational Correspondence

Here we prove that the transition system associated to the rewrite theory in our Maude specification coincides with the reduction semantics for sπ. Given an sπ process P, we use the notation $[\![P]\!]$ to denote its representation in Maude. We state the operational correspondence between them with two theorems, completeness and soundness:

Theorem A.1 *(Completeness). Let P an sπ process and let $[\![P]\!]$ be its corresponding representation in the rewrite theory (Σ, E, ϕ, R) of Section 3. Then, if $P \longrightarrow P'$ then there is a rewriting rule $l : [\![P]\!] \to [\![P']\!]$ that can be applied.*

Proof. By induction on the reduction $P \longrightarrow P'$, with a case analysis on the last rule being applied. We have three base cases, corresponding to the forms of direct communication in the calculus (Rules [R-LinCom], [R-UnCom], and [R-Case]) and three inductive cases (Rules [R-Par], [R-Res] and [R-Struct]). For each case, we show the correspondence with a rewriting rule $l : [\![P]\!] \to [\![P']\!]$.

1. Rule [R-LinCom]: This rule in the operational semantics of the calculus (see Fig. 2) is stated as:

$$\overline{(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \mathsf{lin}\ y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid R)}$$

Then, we must show that there is a rewrite rule that corresponds to this rule, i.e., we need to determine a rewrite rule in our rewrite theory such that:

$$[\![(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \mathsf{lin}\ y(z).Q \mid R)]\!] \longrightarrow [\![(\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid R)]\!]$$

The correspondence with the rewriting rule labeled LINCOM is quite intuitive:

```
new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R =>
                           new* [(x y) nl] P | [z := v] Q | R .
```

Clearly, it holds that

$[\![(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \mathsf{lin}\ y(z).Q \mid R)]\!] =$

```
new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R
```

and

$[\![(\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid R)]\!] =$ `new* [(x y) nl] P | [z := v] Q | R`

where nl $=$ mt. It is easy to check that in virtue of the mathematical induction, any possible reduction involving subprocess R corresponds to a different application of some reduction rule.

2. Rule [R-UnCom]: Then the reduction proceeds as follows:

$$\frac{}{(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \mathsf{un}\ y(z).Q \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid \mathsf{un}\ x(y).Q \mid R)}$$

Again, we must show a corresponding rule in our rewrite theory such that:

$$[\![(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \mathsf{un}\ y(z).Q \mid R)]\!] \longrightarrow [\![(\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid \mathsf{un}\ x(y).Q \mid R)]\!]$$

The correspondence with the rewrite rule UNCOM arises immediately:

```
new* [(x y) nl] x{N} < v > . P | un y{N}(z) . Q | R =>
        new* [(x y) nl] P | [z := v] Q | un y{N}(z) . Q | R .
```

Then, it is evident that

$[\![(\boldsymbol{\nu}xy)(\overline{x}v.P \mid \mathsf{un}\ y(z).Q \mid R)]\!] =$ `new* [(x y) nl] xN < v > . P | un yN(z) . Q | R`

and
$[\![(\boldsymbol{\nu}xy)(P \mid Q[v/z] \mid \mathsf{un}\ x(y).Q \mid R)]\!] =$ `new* [(x y) nl] P | [z := v] Q | un yN(z) . Q | R`

where nl $=$ mt. It is easy to check that in virtue of the mathematical induction, any possible reduction involving subprocess R corresponds to a different application of some reduction rule.

3. Rule [R-Case] Then the reduction proceeds as follows:

$$\frac{j \in I}{(\boldsymbol{\nu}xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : Q_i\}_{i \in I} \mid R) \longrightarrow (\boldsymbol{\nu}xy)(P \mid Q_j \mid R)}$$

As in the previous cases, there is also a direct correspondence with the rule CASE in the rewrite theory:

```
new* [(x y) nl] (x{N} << w . P) | (y{N} >> { (w : Q) CH }) | R =>
           new* [(x y) nl] P | Q | R .
```

As in the other cases, any possible reduction involving subprocess R corresponds to a different application of some reduction rule.

4. Rules [R-Par], [R-Res]: These rules capture the compositionality of the operational semantics but in themselves they do not express any additional alternative of reduction. The effect of these rules is obtained for free in Maude as an effect of the equational theory underlying the rewrite theory and the fact that Maude allows subterm rewriting.
   - Rule [R-Par]: This rule in the operational semantics of the calculus is stated as:

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}$$

   We assume as inductive hypothesis that for a process $P \longrightarrow P'$ it holds that there is a rewriting rule $l : [\![P]\!] \rightarrow [\![P']\!]$ that can be applied. We must show that there is a rewrite rule such that:

$$[\![P \mid Q]\!] \longrightarrow [\![P' \mid Q]\!]$$

Now, given that the rewriting rules in a system module of Maude allow to rewrite specific subterms of a term, and given that $[\![P]\!]$ is a subterm of $[\![P \mid Q]\!]$, the latter will be rewritten to $[\![P' \mid Q]\!]$ by means the application of the same rewrite rule l.

- Similarly, in the case of the Rule [R-Res], which is stated as:

$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu}xy)P \longrightarrow (\boldsymbol{\nu}xy)P'}$$

We also assume as inductive hypothesis that for a process $P \longrightarrow P'$ we have a rewriting rule $l : [\![P]\!] \to [\![P']\!]$ that can be applied. We must show that there is a rewrite rule such that:

$$[\![(\boldsymbol{\nu}xy)P]\!] \longrightarrow [\![(\boldsymbol{\nu}xy)P']\!]$$

Again, the Maude term $[\![(\boldsymbol{\nu}xy)P]\!]$ includes the subterm $[\![P]\!]$ and consequently, the term $[\![(\boldsymbol{\nu}xy)P]\!]$ will be rewritten to $[\![(\boldsymbol{\nu}xy)P']\!]$ by using the rewrite rule l.

5. Rule [R-Struct]: This rule captures the effect of the operational semantics modulo the structural congruence relation. In this way, any possible reduction of a process P such that $P \equiv Q$ is also possible for process Q. This rule is clearly correspondent to the rewrite rule `FLAT`:

```
P => P' if P' := flatten(P) /\ P =/= P' .
```

We recall that the function `flatten` produces an equivalent process where the occurrences of the scope restriction operator are taken to the top-level. □

Theorem A.2 *(Soundness). Let* `T`, `T'` *be instances of the sort* `Trm` *in the rewrite theory* (Σ, E, ϕ, R) *of Section 3 and* $\texttt{T}^C$ *be the canonical form of* `T`*. Then, if there is a rewrite rule* $l : \texttt{T}^C \to \texttt{T}'$ *that can be applied then there exist processes* P, Q *such that* $\texttt{T} = [\![P]\!]$, $\texttt{T}' = [\![Q]\!]$*, and:*

- $P \equiv Q$ *or*
- $P \longrightarrow Q$

The rewriting rule $l : \texttt{T} \to \texttt{T'}$ *can be preceded by a sequence of applications of some equations in the equational theory underlying the rewrite theory.*

Proof. By a case analysis on the rewriting rule being applied over the Maude term `T` and the correspondence with an sπ reduction. We have four cases corresponding to the rewrite rules `FLAT`, `LINCOM`, `UNCOM`, and `CASE` in the rewrite theory. For each case, we show the correspondence with a specific reduction over processes P, Q related to the Maude term being rewritten. We remark that there is a one-to-one correspondence between the process terms (see Section 2) and the instances of the Maude sort `Trm` (see Section 3). For this reason, for each Maude term `P` there is a process P such that $\texttt{P} = [\![P]\!]$.

1. Rule `FLAT`: This rule is stated in Maude (see Section 3) as:

```
crl [FLAT] : P => P' if P' := flatten(P) /\ P =/= P' .
```

Here, the Maude term `P` is rewritten as the term `P'`, which is obtained by the function `flatten`. As already mentioned, the function `flatten` produces an equivalent process where the occurrences of the scope restriction operator are taken to the top-level. Consequently, the Maude term `P'` is the Maude representation of a process Q such that $Q \equiv P$, as expected.

2. Rule `LINCOM`: This rule is stated in Maude (see Section 3) as:

```
new* [(x y) nl] x{N} < v > . P | lin y{N}(z) . Q | R =>
                              new* [(x y) nl] P | [z := v] Q | R .
```

Again, in virtue of the one-to-one correspondence among the process terms syntax and the sort and constructors in the equational theory, we have that there exist processes P and Q such that:

$$[\![P]\!] = \texttt{new* [(x y) nl] x\{N\} < v > . P | lin y\{N\}(z) . Q | R}$$

$$[\![Q]\!] = \texttt{new* [(x y) nl] P | [z := v] Q | R}$$

$\mathsf{dual}(T) = \mathsf{dual}(T, \epsilon)$	$\mathsf{dual}(\mu X T, \sigma) = \mu X \mathsf{dual}(T, [\mu X T/X]; \sigma)$
$\mathsf{dual}(\mathsf{end}, \sigma) = \mathsf{end}$	$\mathsf{dual}(X, \sigma) = X$
$\mathsf{dual}(q?T.U, \sigma) = q!(T\sigma).\mathsf{dual}(U, \sigma)$	$\mathsf{dual}(q!T.U, \sigma) = q?(T\sigma).\mathsf{dual}(U, \sigma)$
$\mathsf{dual}(q \oplus \{l_i : T_i\}_{i\in I}, \sigma) = q\&\{l_i : \mathsf{dual}(T_i, \sigma)\}_{i\in I}$	$\mathsf{dual}(q\&\{l_i : T_i\}_{i\in I}, \sigma) = q \oplus \{l_i : \mathsf{dual}(T_i, \sigma)\}_{i\in I}$

Fig. 15. Message-closure duality.

Clearly, process P must be equivalent to a process matching a pattern $(\boldsymbol{\nu}xy)\ldots\overline{x}v.P' \mid \mathsf{lin}\ y(z).Q' \mid R$ and process Q must be equivalent to a process matching a pattern $(\boldsymbol{\nu}xy)\ldots P' \mid Q' \mid R$. Consequently, $P \longrightarrow Q$, as expected.

3. Rule `UNCOM`: This rule is stated as follows (see Section 3):

```
new* [(x y) nl] x{N} < v > . P | un y{N}(z) . Q | R =>
          new* [(x y) nl] P | [z := v] Q | un y{N}(z) . Q | R .
```

As before, there exist processes P and Q for which holds:

$[\![P]\!] =$ `new* [(x y) nl] x{N} < v > . P | un y{N}(z) . Q | R`

$[\![Q]\!] =$ `new* [(x y) nl] P | [z := v] Q | un y{N}(z) . Q | R`

Then, process P is of the form $(\boldsymbol{\nu}xy)\ldots\overline{x}v.P' \mid \mathsf{un}\ \ y(z).Q' \mid R$ and process Q is of the form $(\boldsymbol{\nu}xy)\ldots P' \mid Q' \mid \mathsf{un}\ y(z).Q' \mid R$ and $P \longrightarrow Q$, as expected.

4. Rule `CASE`: This rule is stated in the rewrite theory as:

```
new* [(x y) nl] (x{N} << w . P) | (y{N} >> { (w : Q) CH }) | R =>
             new* [(x y) nl] P | Q | R .
```

Once again, in virtue of the one-to-one correspondence among the process term syntax and the Maude sorts and constructors, we have that there exist processes P and Q such that:

$[\![P]\!] =$ `new* [(x y) nl] (x{N} « w . P) | (y{N} » (w : Q) CH ) | R`

$[\![Q]\!] =$ `new* [(x y) nl] P | Q | R`

Process P is equivalent to a process matching the pattern $(\boldsymbol{\nu}xy)\ldots x \triangleleft l_j.P' \mid y \triangleright \{l_i : Q_i\}_{i\in I} \mid R$ and process Q is equivalent to a process matching the pattern $(\boldsymbol{\nu}xy)\ldots P' \mid Q_j \mid R$. Thereafter, we have that $P \longrightarrow Q$, as expected. □

Appendix B. Appendix to §4: Omitted material on algorithmic type checking

The duality operation on session types is essential to enforce communication correctness; it is defined in Fig. 15 where σ is a substitution and ϵ is the empty substitution. In Maude, the operation `mclo-dual`, given next, enables us to obtain the dual of a type. It takes an instance of the sort `Type` and produces an instance of the same sort:

```
op firstv : -> Qid .
op e : -> Type .
sort Qidset Typeset .
subsort Type < Typeset .
subsort Qid < Qidset .
var QS : Qidset .
var TS : Typeset .
op replace-dual-aux(_,_,_) : Qidset Typeset Type -> Type .
op replace-dual-aux-strict(_,_,_) : Qidset Typeset Type -> Type .
op mclo-dual : Type -> Type .
op mclo-dual : Qidset Typeset Type -> Type .
op mclo-dual-aux : Qidset Typeset ChoiceTset -> ChoiceTset .
op TpEmpty : -> Typeset .
op QiEmpty : -> Qidset .
op _,_ : Typeset Typeset -> Typeset [assoc id: TpEmpty] .
op _,_ : Qidset Qidset -> Qidset [assoc id: QiEmpty] .
eq mclo-dual(T) = mclo-dual(firstv , e, T) .
eq mclo-dual(QS ,TS , var(x)) = var(x) .
eq mclo-dual(QS, TS, end) = end .
eq mclo-dual(QS, TS, bool) = bool .
eq mclo-dual(QS, TS ,  q ! T . T') =
```

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

$$\Gamma \div \emptyset = \Gamma \qquad \frac{\Gamma_1 \div L = \Gamma_2, x : T \quad \mathsf{un}(T)}{\Gamma_1 \div (L, x) = \Gamma_2} \qquad \frac{\Gamma_1 \div L = \Gamma_2 \quad x \notin \mathsf{dom}(\Gamma_2)}{\Gamma_1 \div (L, x) = \Gamma_2}$$

Fig. 16. Definition of context update (top) and context difference (bottom).

```
  q ? (replace-dual-aux(QS, TS, T)) . mclo-dual(QS, TS, T') .
eq mclo-dual(QS, TS , q ? T . T') =
  q ! (replace-dual-aux(QS, TS, T)) . mclo-dual(QS, TS, T') .
eq mclo-dual(QS, TS, (u[x] T) ) = u[x] mclo-dual( (QS, x),(TS ,(u[x] T) ), T ) .
eq mclo-dual-aux(QS,TS, empty) = empty .
eq mclo-dual-aux(QS, TS, (y : T) CHT) = (y : mclo-dual(QS, TS, T)) mclo-dual-aux(QS,TS, CHT) .
eq mclo-dual(QS, TS, q +{CHT}) = q &{(mclo-dual-aux(QS,TS,CHT))} .
eq mclo-dual(QS, TS, q &{CHT}) = q +{(mclo-dual-aux(QS,TS,CHT))} .
eq replace(firstv , e, T) = T .
eq replace-dual-aux(firstv , e, T) = T .
eq replace-dual-aux(QiEmpty, TpEmpty, T) = T .
eq replace-dual-aux((x , QS) , (T , TS) , (T')) =
   replace-dual-aux(QS, TS, replace(x,T ,T')) [owise] .
```

In essence, `mclo-dual` calculates the dual of a recursive type, replacing recursive variables when they occur in an input or an output (see [13] for a discussion on this transformation). When it is applied to a non-recursive type, it returns the usual dual type, i.e., it swaps ! and ? and + and &. In the first two arguments, function `mclo-dual : Qidset Typeset Type -> Type` accumulates some substitutions that are initialized by the equation:

```
 eq mclo-dual(T) = mclo-dual(firstv , e, T) .
```

New substitutions are created by the equation:

```
eq mclo-dual(QS, TS, (u[x] T)) = u[x] mclo-dual((x , QS),( (u[x] T) , TS), T) .
```

Function `replace-dual-aux` forwards the substitutions to `replace` so that a list of substitutions is applied to a type.

Another important operation in the algorithmic type checking is *context update*, denoted '+'. This operation enables us to extend a type context with a new type for a variable; it is used when checking the type of input, output, branching, and selection processes. The formal definition is shown in Fig. 16 (top). There are two rules: the first one requires linear variables not to be in the context; the second one requires that unrestricted types remain unchanged.

The context update operation is implemented in Maude by means of the operator `_+_:_`, which takes a type context, a value and a type and produces a new context.

```
op _+_:_ : Context Value Type -> Context .
ceq C + v : T = (C, (v : T)) if not v in C .
ceq (C, (v : T1)) + v : T2 = (C, (v : T2'))
   if unrestricted(T1) /\ T1' := unfold(T1) /\ T2' := unfold(T2) /\ equal(T1', T2') .
eq C + v : T = invalid-context [owise] .
```

Each rule is associated to a previous equation and it proceeds as expected. Particularly, for the case of unrestricted variables, given that some types are infinite then before the update the unrestricted types are unfolded (cf. operation `unfold`). Type unfolding is the mechanism that we use to deal with infinite types. Given a type T, it is defined as follows: if T is a recursive type $\mu a.U$ then unfolding means performing the substitution $U[\mu a.U/a]$. Otherwise, T remains unaltered. The types `T1` and `T2` involved in the update operation are unfolded and the resulting types must be equal. Lastly, we add an additional equation to produce the constant `invalid-context` when the context update can not be performed.

The *context difference* function, denoted '$\div$', is the mechanism that prevents the use of linear variables in several threads. This operation removes the variables in a set from a typing context. This function is defined inductively as shown in Fig. 16 (bottom). It is implemented in Maude as follows:

```
op _/_ : Context Chanset -> Context .
eq C / mt = C .
eq ((a : T), C) / (a L1) = C / L1 .
eq C / L1 = C [owise] .
```

On the other hand, the function `erase` (used in Section 4.3) inductively analyzes a process; when it reaches an annotated subprocess (i.e. a process `new [x y : T] P`), it returns a non-annotated process (i.e., a process `new [x y] P`). This function is defined in Maude as follows:

```
op  erase : Trm -> Trm .
op  erase-aux : Choiceset -> Choiceset .
eq  erase(nil) = nil .
eq  erase(a < v > . P) = a < v > . erase(P) .
eq  erase(q a(x) . P) = q a(x) . erase(P) .
eq  erase(if v then P else Q fi) = if v then erase(P) else erase(Q) fi .
ceq erase(P | Q) = erase(P) | erase(Q) if P =/= nil and Q =/= nil .
eq  erase(a >> {CH}) = a >> { erase-aux(CH) } .
eq  erase(a << x . P) = a << x . erase(P) .
eq  erase-aux(empty) = empty .
eq  erase-aux((x : P) CH) = (x : erase(P)) erase-aux(CH) .
eq  erase(new [x y] P) = new [x y] erase(P) .
eq  erase(new [x y : T] P) = new [x y] erase(P) .
```

Appendix C. Appendix to §5: Omitted material on lock and deadlock detection

Here we prove that our approach to the lock free verification relying on the LTL formula in Section 5 is sound for arbitrary sπ processes. Moreover, we also prove that this approach is complete for finite sπ processes, i.e., processes without replicated input. Finally, we show some examples of the usage of the Maude tools for deadlock and lock detection as presented in Section 5.

C.1. Soundness and completeness for lock detection

Given an sπ process $(\boldsymbol{\nu} x_1, y_1)\dots(\boldsymbol{\nu} x_n, y_n)P$ such that $\emptyset \vdash P$, lock freedom can be stated by means of the following LTL formula:

$$\phi = \bigwedge_{(x_i, y_i)\in vars(P)} \Box(\Diamond \mathsf{wait}(x_i, y_i, Q) \longrightarrow \Diamond \mathsf{sync}(x_i, y_i, R))$$

It is worth to remark that predicates wait and sync are represented in the Kripke structure in Maude by means of the predicates `pwait` and `psync`. Moreover, in the following we use the notation $\rho \in \mathsf{Path}(\mathcal{K})_P$ to denote the set of execution paths in $\mathcal{K}$ starting in P.

Theorem C.1 *(Soundness). Let $\mathcal{K}$ be the Kripke structure in Maude for process P. If $\mathcal{K}, P \models \phi$ then P is lock free in the sense stated in Section 5.*

Proof. We assume $\mathcal{K}, P \models \phi$ and suppose for the sake of contradiction that P is a locked process. Then, there is a process Q such that $P \longrightarrow^* Q$, $Q \models \mathtt{pwait}(x_j, y_j)$ for some $1 \le j \le n$, and there is no process R such that $Q \longrightarrow^* R$ and $R \models \mathtt{psync}(x_j, y_j)$. Let $\rho \in \mathsf{Path}(\mathcal{K})_P$ an execution path such that $\rho(m) = Q$ for some $m \in \mathbb{N}$. By definition $\rho \not\models \phi$, and thereby $\mathcal{K}, P \not\models \phi$ which contradicts the hypothesis. We then conclude that P is lock free. $\square$

Theorem C.2 *(Completeness for processes without replicated input). Let $\mathcal{K}$ be the Kripke structure for process P. If P has no replicated input and is lock free, then $\mathcal{K}, P \models \phi$.*

Proof. We assume that P is lock free; for the sake of contradiction we suppose $\mathcal{K}, P \not\models \phi$. Then, there is an execution path $\rho \in \mathsf{Path}(\mathcal{K})_P$ such that $\rho \not\models \Box(\Diamond \mathtt{pwait}(x_j, y_j) \longrightarrow \Diamond \mathtt{psync}(x_j, y_j))$ for some $1 \le j \le n$, that is, there is an $m \in \mathbb{N}$ such that $\rho(m) \models \mathtt{pwait}(x_j, y_j)$ and $\rho(k) \not\models \Diamond \mathtt{psync}(x_j, y_j)$, for all $k \ge m$. Given that P does not have replicated input, each of its execution paths is finite; hence, there is a process R in the execution path ρ such that $P \longrightarrow^* R \not\longrightarrow$ and $R \models \mathtt{pwait}(x_j, y_j)$, then P is deadlocked, therefore it is locked as well, which is a contradiction. We thus conclude that if P is lock free then $\mathcal{K}, P \models \phi$. $\square$

C.2. Examples

We now present a few examples of well-typed processes in sπ, with different lock- and deadlock-freedom properties:

$$P_1 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3)(\overline{x_3}\mathsf{true}.\overline{x_1}\,\mathsf{true}.\overline{x_2}\,\mathsf{false}.\mathbf{0} \mid \mathsf{lin}\ y_3(z).\mathsf{lin} y_2(x).\mathsf{lin} y_1(w).\mathbf{0})$$

$$P_2 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} ab)(\overline{x_1} b.\mathbf{0} \mid \overline{a}\,\mathsf{true}.\mathbf{0} \mid \mathsf{un}\ y_1(z).\overline{x_2} z.\mathbf{0} \mid \mathsf{un}\ y_2(w).\overline{x_1} w.\mathbf{0})$$

$$P_3 = (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)(\overline{z}x.\text{lin } y(v).\mathbf{0} \mid \text{lin } w(t).\overline{t}\text{ true}.\mathbf{0})$$

$$P_4 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3)(\boldsymbol{\nu} x_4 y_4)(\boldsymbol{\nu} x_5 y_5)(\boldsymbol{\nu} x_6 y_6)(\boldsymbol{\nu} x_7 y_7)$$
$$\big(\overline{x_2}\text{ true.un } x_1(w).\overline{w}x_4.\mathbf{0} \mid \text{lin } y_2(b).\text{lin} y_4(z).\mathbf{0} \mid \text{lin } y_6(a).\text{lin } y_5(b).\mathbf{0} \mid$$
$$\overline{y_1}x_3.\text{lin } y_3(z).\overline{z}x_7.\mathbf{0} \mid \overline{x_5}\text{ true}.\overline{x_6}\text{ false}.\mathbf{0}\big)$$

$$P_5 = (\boldsymbol{\nu} x_1 y_1)(\boldsymbol{\nu} x_2 y_2)(\boldsymbol{\nu} x_3 y_3)$$
$$(\overline{x_1}\text{ true}.\mathbf{0} \mid y_1(z)\overline{x_1}\text{ true}.\mathbf{0} \mid x_2 \triangleright \{T : \overline{x_3}\text{ true}.\mathbf{0}, F : \overline{x_3}\text{ false}.\mathbf{0}\} \mid y_2 \triangleleft F.\mathbf{0} \mid \text{lin } y_3(k).\mathbf{0})$$

Processes P_1 and P_2 were discussed in the main text. We first check that P_3, P_4, and P_5 are typable. Processes P_3 and P_4 are implemented as follows:

```
---- P3
ops Typez Typey : -> Type .
ops P31 P32 : -> Trm .
eq Typez = lin !(lin ! bool . end) . end .
eq Typey = lin ? bool . end .
eq P31 = 'z'{0}< 'x'{0} > . lin 'y'{0}('v') . nil .
eq P32 = lin 'w'{0}('t') . 't'{0} < True > . nil .
eq P3 = new['z' 'w' : Typez] new['y' 'x' : Typey](P31 | P32) .
----- P4
ops P41 P42 P43 P44 P45 : -> Trm .
eq Typex14 = (u['a1'] un ?( u['a3'] un !( u['a4'] un ! ( end ) .
   var('a4')) . var('a3')) . var('a1')) .
eq Typex24 = (lin ! (bool) . end) .
eq Typex34 = (u['a3'] un ! ( u['a4'] un ! ( end ) . var('a4') ) . var('a3')) .
eq Typex44 = (u['a4'] un ! ( end ) . var('a4')) .
eq Typex54 = (lin ! (bool) . end) .
eq Typex64 = (lin ! (bool) . end) .
eq Typex74 = end .

eq P41 = 'x2'{0} < True > . un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . nil .
eq P42 = lin 'y2'{0}('b') . lin 'y4'{0}('z') . nil .
eq P43 = 'y1'{0} < 'x3'{0} > . lin 'y3'{0}('z') . 'z'{0} < 'x7'{0} > . nil .
eq P44 = 'x5'{0} < True > . 'x6'{0} < False > . nil .
eq P45 = lin 'y6'{0}('a') . lin 'y5'{0}('b'). nil .

eq P4 = new['x1' 'y1' : Typex14 ] new['x2' 'y2' : Typex24] new['x3' 'y3' : Typex34]
         new['x4' 'y4' : Typex44]new['x5' 'y5' : Typex54] new['x6' 'y6' :  Typex64]
          new[ 'x7' 'y7' : Typex74](P41 | P42 | P43 | P44 | P45) .
```

Process P_5 is implemented as follows:

```
eq Typex51 = (u['a'] un ! bool . var('a')) .
eq Typex52 = (u['b'] un &{('T' : var('b'))('F' : var('b'))}) .
eq Typex53 = lin ! bool . end .
eq P52 = ('x2'{0}>>{('T' : 'x3'{0}< True > . nil)
 ('F' : 'x3'{0}< False > . nil)}) | ('y2'{0} << 'T' . nil) | lin 'y3'{0}('k') . nil .
eq P51 = 'x1'{0}< True > . nil | un 'y1'{0}('z') . 'x1'{0}< True > . nil .
eq P5 = new['x1' 'y1' : Typex51 ]
          new['x2' 'y2' : Typex52]
              new['x3' 'y3' : Typex53] (P51 | P52) .
```

Now we check these processes are well-typed:

```
reduce in TEST : well-typed(P3) .
rewrites: 103 in 0~ms cpu (0~ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in TEST : well-typed(P4) .
rewrites: 504 in 0~ms cpu (0~ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in TEST : well-typed(P5) .
```

```
ops P1 P2 P3 P4 P5 : -> Trm .
eq P1 = new* [('y1' 'x1')('y2' 'x2')('y3' 'x3')]
              ('x3'{0} < True > . 'x1'{0} < True > . 'y1'{0} < False > . nil |
               lin 'y3'{0}('z') . lin 'y2'{0}('x') . lin 'x2'{0}('w') . nil) .
eq P2 = new* [('x1' 'y1')('x2' 'y2')('a' 'b')]
              ('x1'{0} < 'b'{0} > . nil | 'a'{0} < True > . nil |
               un 'y1'{0}('z') . 'x2'{0} < 'z'{0} > . nil |
               un 'y2'{0}('w') . 'x1'{0} < 'w'{0} > . nil ) .
eq P3 = new* [('z' 'w')('x' 'y')]
         ('z'{0}< 'x'{0} > . lin 'y'{0}('v') . nil) | lin 'w'{0}('t') . 't'{0} < True > . nil) .
eq P4 = new*[('x1' 'y1')('x2' 'y2')('x3' 'y3')('x4' 'y4')('x5' 'y5')('x6' 'y6')('x7' 'y7')]
             'x2'{0} < True > . un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . nil |
             lin 'y2'{0}('b') . lin 'y4'{0}('z') . nil |
             'y1'{0} < 'x3'{0} > . lin 'y3'{0}('z') . 'z'{0} < 'x7'{0} > . nil |
             'x5'{0} < True > . 'x6'{0} < False > . nil |
             lin 'y6'{0}('a') . lin 'y5'{0}('b'). nil .
eq P5 = new*[('x1' 'y1)('x2' 'y2')('x3' 'y3')]
             (x1'{0}< True > . nil | un 'y1'{0}('z') . 'x1'{0}< True > . nil |
              ('x2'{0}>>{('T' : 'x3'{0}< True > . nil)('F' : 'x3'{0}< False > . nil)}) |
              ('y2'{0} << 'T' . nil) | lin 'y3'{0}('k') . nil) .
```

Fig. 17. Processes in Maude.

```
rewrites: 322 in 0~ms cpu (0~ms real) (~ rewrites/second)
result Bool: true
```

Some intuitions for the other processes follow:

- Process P_3 is a simple lock and deadlock free process.
- Process P_4 is not deadlock free and not lock free. The deadlock is reached after some synchronizations in the pairs of co-variables x_1, y_1, x_2, y_2, x_3, y_3 and x_4, y_4.
- Process P_5 is both lock and deadlock free.

Fig. 17 gives the Maude terms associated to these processes.

C.3. Detecting (dead)locks in Maude

We give the execution of both commands with respect to process `P3`, which is deadlock and lock free:

```
search P3 =>! new* [nl:QidSet] P:Trm
                     such that wait-aux(nl:QidSet, P:Trm) .
red modelCheck(P3, build-lock-formula(('z' 'w') ('x' 'y'))) .
```

We obtain:

```
search in TEST : P3 =>! new*[nl:QidSet] P:Trm
                      such that wait-aux(nl:QidSet, P:Trm) = true .

No solution.

reduce in TEST : modelCheck(P3, build-lock-formula(('x' 'y') 'z' 'w')) .
result Bool: true
```

We perform a similar analysis for process `P4`; first we check the deadlock property:

```
search P4 =>! new* [nl:QidSet] P:Trm such that wait-aux(nl:QidSet, P:Trm) .
```

Then, we obtain:

```
search in TEST : P4 =>! new*[nl:QidSet]P:Trm such that wait-aux(nl:QidSet, P:Trm) = true .

Solution 1 (state 13)
states: 14  rewrites: 1744 in 4~ms cpu (0~ms real) (436000 rewrites/second)
nl:QidSet --> ('x2' 'y2') ('x3' 'y3') ('x4' 'y4') ('x5' 'y5') ('x6' 'y6') ('x7' 'y7') 'y1' 'x1'
P:Trm -->'x5'{0} < True > . 'x6'{0} < False > . nil | lin 'y6'{0}('a') . lin 'y5'{0}('b') . nil |
un 'x1'{0}('w') . 'w'{0} < 'x4'{0} > . nil
```

```
No more solutions.
states: 14  rewrites: 1744 in 4~ms cpu (0~ms real) (436000 rewrites/second)
```

Process P4 reaches a deadlock involving variables x5, y5, x6, y6; then we also know that the process is locked.

Now we analyze process P5 as we have done before. We obtain:

```
search in TEST : P5 =>! new*[nl:QidSet]P:Trm such that wait-aux(nl:QidSet, P:Trm) = true .

No solution.

reduce in TEST : modelCheck(P5, build-lock-formula(('x1' 'y1') ('x2' 'y2') 'x3' 'y3')) .
rewrites: 3236 in 4~ms cpu (1~ms real) (809000 rewrites/second)
result Bool: true
```

As expected, the results show the process is both lock and deadlock free.

Appendix D. Appendix to §6: Algorithmic type checking for HOπ

This section is dedicated to the proof of the Theorem 6.2, which states the correspondence between the type system in Section 6.1.2 and the algorithmic type system in Section 6.3. As usual, the proof is divided in two parts: soundness and completeness.

D.1. Preliminaries

The following lemma is necessary to ensure that strengthening is well defined.

Lemma D.1. *If* $\Gamma; \Theta_1 \Vdash D \triangleright T; \Theta_2$ *then* $\Theta_2 \subseteq \Theta_1$.

Proof. When the last rule applied is one of the following: A-Inact, A-Sess,A-Sesh, A-RVar, A-LVar or A-Prom then the result follows by a straightforward inspection of the rules involved.

Suppose that the last rule applied is A-ResS, then we have $\Gamma; \Theta_1 \Vdash (\nu s_1 : S_1)P; \Theta_2 \div \{s_1, s_2\}$; by inductive hypothesis we have $\Theta_2 \subseteq \Theta_1, s_1 : S_1, s_2 : S_2$ (1) and by definition of $\div$, $\Theta_2 \div \{s_1, s_2\} \subseteq \Theta_2$ (2). Thus, by (1), (2) and the fact that $\Theta_2 \div \{s_1, s_2\}$ does not contain s_i for $i = 1, 2$, then we know $\Theta_2 \subseteq \Theta_1$. When the derivation ends with A-Bra, we have $\Gamma; \Theta_1 \Vdash u \triangleright \{l_i : P_i\}_{i\in I}; \cap_{i\in I}\Theta_3^i \div u$, then it is clear that $\forall i \in I\ \cap_{i\in I}\ \Theta_3^i \div u \subseteq \Theta_3^i$ and $u \notin dom(\cap_{i\in I}\Theta_3^i \div u)$ (3). By inductive hypothesis $\Theta_3^i \subseteq \Theta_2, u : S$ and $\Theta_2 \subseteq \Theta_1$. By (3) $\cap_{i\in I}\Theta_3^i \div u \subseteq \Theta_2$, and then by transitivity $\cap_{i\in I}\Theta_3^i \div u \subseteq \Theta_1$. The remaining cases follow the same reasoning. □

The following theorem allows us to add extra information when necessary; it is essential to prove completeness.

Theorem D.1 *(Weakening). If* $\Gamma; \Theta_1 \Vdash D \triangleright T; \Theta_2$ *then* $\Gamma; \Theta_1, y : \theta \Vdash D \triangleright T; \Theta_2, y : \theta$*, for* $y : \theta \notin \Theta_1$.

Proof. When D is either a name or the **0** process, the proof follows from a simple analysis of the rules A-Sess, A-Sh, A-LVar and A-Nil. Proofs for the other cases follow by induction on the typing derivation:

- When the derivation ends with A-Send, by induction on u, V and P we obtain $\Gamma; \Theta_3, y : \theta \vdash P; \Theta_4, y : \theta$. Since $y : \theta \notin \Theta_1$, $y \neq u$, thus $\Theta_4, y : \theta \div u = \Theta_4', y : \theta$ for Θ_4' not containing u. When the derivation ends with A-Sel the proof follows the same reasoning.
- When the derivation ends with A-Bra by induction on u and the induction on P_i for all $i \in I$ we know $\Gamma; \Theta_2, y : \theta \Vdash P_i; \Theta_3^i, y : \theta$, then $y : \theta \in \cap_{i\in I}\Theta_3^i$. Since $y \neq u$, by definition of $\div$ we obtain $\Gamma; \Theta_1, y : \theta \Vdash u \triangleright \{l_i : P_i\}_{i\in I}; \Theta_3', y : \theta$, where $\Theta_3' = \cap_{i\in I}\Theta_3^i \div u$.
- When the last rule applied is A-SRcv, then we have $\Gamma; \Theta_1' \vdash u?(x).P; \Theta_3 \div \{u, x\}$. Take $y \neq u, x$, and by inductive hypothesis we have $\Gamma; \Theta_1', y : \theta \Vdash u \triangleright ?(U).S; \Theta_2, y : \theta$, and by induction on the second branch we have $\Gamma; \Theta_2, u : S, x : U, y : \theta \Vdash P; \Theta_3, y : \theta$, hence applying Rule A-SRcv we obtain $\Gamma; \Theta_1', y : \theta \vdash u?(x).P; \Theta_3, y : \theta \div \{u, x\}$, and by definition of $\div$ $\Theta_3, y : \theta \div \{u, x\} = \Theta_3', y : \theta$ for some Θ_3' that does not contain x or u.
- When the last rule applied is A-ResS, then take $y : \theta \neq s_1, s_2$, and by inductive hypothesis we know $\Gamma; \Theta_1, s_1 : S_1, \bar{s_2} : S_2, y : \theta \Vdash P; \Theta_2, y : \theta$, then the proof concludes by applying rule A-ResS and the fact that $y : \theta \neq s_1, s_2$. The same reasoning is used for the proof for A-SAbs.

The remaining cases follow by straightforward induction. □

The following theorem is essential for proving soundness since it enables us to eliminate extra information from the input context when necessary.

Theorem D.2 *(Strengthening). If* $\Gamma; \Theta_1 \Vdash D \triangleright T; \Theta_2, y : \theta$ *then* $\Theta_1 = \Theta_1', y : \theta$ *and* $\Gamma; \Theta_1' \Vdash D \triangleright T; \Theta_2$.

Proof. The case for names and the **0** process follows by an inspection of Rules A-Sess, A-Sh, A-LVar and A-Nil. The proof for the other cases follow by induction on the structure of the derivation.

- If the derivation ends with Rule A-Send, suppose that $\Gamma; \Theta_1 \Vdash u!\langle V\rangle.P; (\Theta_4 \div u), x : R$. Note that $(\Theta_4 \div u), x : \theta = (\Theta_4, x : \theta) \div u$; induction on P says that $\Theta_3 = \Theta_3', x : R$ and $\Gamma; \Theta_3' \Vdash P; \Theta_4$; the proof concludes by induction on V and U and the application of rule A-Send.
- If the derivation ends with rules: A-ShRcv, A-SRcv, A-Sel or A-Bra, we know that $y \neq u$ (also $y \neq x$ in the case of A-SRcv), then $\Theta_3 = \Theta_3', y : \theta$. The proof follows by induction on the subprocess(es) and then induction on the name u, and concludes by applying A-ShRcv, A-SRcv, A-Sel or A-Bra respectively.
- When the derivation ends with rule A-ResS, then $y \neq s_i, s_2$, thus by inductive hypothesis and then rule A-Ress we get the result. A similar reasoning is followed for A-SAbs. The remaining cases follow by straightforward induction. □

Note that Rule End in Fig. 9 allows to introduce new channels of type end, then the algorithm allows channels of type end to pass freely through the judgment, hence we say that the algorithm succeeds when the output context is either empty or only contains channels of type end.

D.2. Completeness and soundness

The following results establish the algorithmic correctness with respect to HOπ. In this way, this corresponds to the proof of the Theorem 6.2 in Section 6.3. As mentioned before, we divided Theorem 6.2 in two parts corresponding to the completeness and the soundness theorems. Recall that, given a linear context Θ, $\mathsf{End}(\Theta)$ means that every channel in Θ is of type end.

Theorem D.3 *(Completeness). Given contexts* Γ, Λ, Δ, *a process* D, *a type* T, *and* w *(name or variable) we have the following:*

- *If* $\Gamma; \Lambda; \Delta \vdash w \triangleright T$ *then* $\Gamma; \Theta_1 \Vdash w \triangleright T; \emptyset$, *where* $\Theta_1 = \Lambda, \Delta$.
- *If* $\Gamma; \Lambda; \Delta \vdash \mathtt{erase}(D) \triangleright T$ *(where either* $T = \diamond$ *or* $T = C \rightsquigarrow \diamond$*) then* $\Gamma; \Theta_1 \Vdash D \triangleright T; \Theta_2$ *where* $\Theta_2 = \emptyset$ *or* $\mathsf{End}(\Theta_2)$, *and* $\Theta_1 = \Lambda, \Delta$.

Proof. Completeness for names follows by a straightforward inspection of the algorithmic type checking rules. The proof for the remaining cases follows by induction on the structure of derivations for the hypothesis.

- If the derivation ends with Rule Send, then we have $\Gamma; \Lambda_1, \Lambda_2; ((\Delta_1, \Delta_2) \setminus u : S), u :!\langle U\rangle.S \vdash \mathtt{erase}(u!\langle V\rangle.P) \triangleright \diamond$, where either $u : S \in \Delta_1$ or $u : S \in \Delta_2$, thus we consider two cases. We detail the case $u : S \in \Delta_2$, that is, $\Delta_2 = \Delta_2', u : S$. By induction (on V) we have $\Gamma; \Theta_1 \Vdash V \triangleright U; \Theta_3$ where $\Theta_3 = \emptyset$ or $\mathsf{End}(\Theta_3)$, and $\Theta_1 = \Lambda_1, \Delta_1$. Induction on P tells us $\Gamma; \Theta_2 \Vdash P; \Theta_4$ (1), where either $\Theta_4 = \emptyset$ or $\mathsf{End}(\Theta_4)$, and $\Theta_2 = \Lambda_2, \Delta_2$ (and $\Theta_2' = \Lambda_2, \Delta_2'$). By weakening on V we get $\Gamma; \Theta_1, \Theta_2 \Vdash V \triangleright U; \Theta_3, \Theta_2$.
  - If $\Theta_3 = \emptyset$ we have $\Gamma; \Theta_1, \Theta_2', u : S \Vdash V \triangleright U; \Theta_2', u : S$ (2). It is clear that $\Gamma; \emptyset; u :!\langle U\rangle.S \vdash u \triangleright !\langle U\rangle.S$, and by completeness of values and weakening $\Gamma; \Theta_1, \Theta_2', u :!\langle U\rangle.S \Vdash u \triangleright !\langle U\rangle.S; \Theta_1, \Theta_2'$ (3), thus applying Rule A-Send to (1)-(3) we obtain $\Gamma; \Theta_1, \Theta_2', u :!\langle U\rangle.S \Vdash u!\langle V\rangle.P; \Theta_4 \div u$. Note that by definition of $\div$, either $\Theta_4 \div u = \emptyset$ or $\mathsf{End}(\Theta_4 \div u)$. If $\Theta_4 = \emptyset$, then $\Theta_4 \div u = \emptyset$. If $\mathsf{End}(\Theta_4)$ and $x \in dom(\Theta_4)$ then $\Theta_4 = \Theta_4', x : \mathsf{end}$ where $\mathsf{End}(\Theta_4')$ and $\Theta_4 \div u = \Theta_4'$. If $x \notin dom(\Theta_4)$ then $\Theta_4 \div u = \Theta_4$.
  - If $\mathsf{End}(\Theta_3)$ by weakening on P we have $\Gamma; \Theta_2, \Theta_3 \Vdash P; \Theta_4, \Theta_3$ (1') and $\Gamma; \Theta_1, \Theta_2', u : S \Vdash V \triangleright U; \Theta_2, \Theta_3$ (2'), thus applying A-Send to (1'),(2') and (3), we obtain $\Gamma; \Theta_1, \Theta_2', u :!\langle U\rangle.S \Vdash u!\langle V\rangle.P; \Theta_3, \Theta_4 \div u$. By definition of $\div$ either $\Theta_3, \Theta_4 \div u = \emptyset$ or $\mathsf{End}(\Theta_3, \Theta_4 \div u)$.

  Note that

$$\begin{aligned}\Theta_1, \Theta_2', !\langle U\rangle.S &= (\Lambda_1, \Delta_1), (\Lambda_2, \Delta_2'), !\langle U\rangle.S \\ &= (\Lambda_1, \Lambda_2), (\Delta_1, \Delta_2'), !\langle U\rangle.S \\ &= (\Lambda_1, \Lambda_2), ((\Delta_1, \Delta_2) \setminus u : S), !\langle U\rangle.S\end{aligned}$$

  When $u : S \in \Delta_1$ the proof es similar to the case $u : S \in \Delta_2$.
- When the derivations end with Rule Rcv there are three cases, depending on the type of x. We detail the case when $x : S'$. we have $\Gamma; \Lambda_1; \Delta_1, u :?(S').S \vdash \mathtt{erase}(u?(x).) \triangleright \diamond$, and by inductive hypothesis $\Gamma; \Theta_1, u : S, x : S' \Vdash P; \Theta_2$ (1) where either $\Theta_2 = \emptyset$ or $\mathsf{End}(\Theta_2)$, and $\Theta_1 = \Lambda_1, \Delta_1$. It is clear that $\Gamma; \emptyset; u :?(S').S \vdash u \triangleright ?(S').S$, hence by completeness of values and weakening we know that $\Gamma; \Theta_1, x : S', u :?(S').S \Vdash u \triangleright ?(S').S; \Theta_1, x : S'$ (2), then by applying A-SRcv to (1) and (2) we get $\Gamma; \Theta_1, u :?(S').S \Vdash u?(x).P; \Theta_2 \div \{x, u\}$. Definition of $\div$ says that either $\Theta_2 \div \{x, u\} = \emptyset$ or $\mathsf{End}(\Theta_2 \div \{x, u\})$. It is also clear that $\Theta_1, ?(S').S = \Lambda_1, \Delta_1, ?(S').S$. The proofs when x is of higher-order type or shared type are similar.

- If the derivation ends with Rule Par, we have $\Gamma; \Lambda_1, \Lambda_2; \Delta_1, \Delta_2 \vdash \texttt{erase}(P|Q) \triangleright \diamond$ and by induction we get $\Gamma; \Theta_1 \Vdash P; \Theta_3$ (1), and $\Gamma; \Theta_2 \Vdash Q; \Theta_4$ (2) where either $\Theta_3 = \emptyset$ or $\mathsf{End}(\Theta_3)$, either $\Theta_4 = \emptyset$ or $\mathsf{End}(\Theta_4)$, $\Theta_1 = \Lambda_1, \Delta_1$ and $\Theta_2 = \Lambda_2, \Delta_2$. If $\Theta_3 = \emptyset$, by weakening on P we get $\Gamma; \Theta_1, \Theta_2 \Vdash P; \Theta_2$ (3), then applying A-Par to (2) and (3) we obtain $\Gamma; \Theta_1, \Theta_2 \Vdash P|Q; \Theta_4$. If $\mathsf{End}(\Theta_3)$, by weakening P and Q we get $\Gamma; \Theta_1, \Theta_2 \Vdash P; \Theta_2, \Theta_3$ (4) and $\Gamma; \Theta_2, \Theta_3 \Vdash Q; \Theta_3, \Theta_4$ (5), and then applying A-Par to (4) and (5) to obtain $\Gamma; \Theta_1, \Theta_2 \Vdash P|Q; \Theta_3, \Theta_4$; note that either case for Θ_4, $\mathsf{End}(\Theta_3, \Theta_4)$. It is clear that $\Theta_1, \Theta_2 = \Lambda_1, \Lambda_2, \Delta_1, \Delta_2$.
- When the derivation ends with Rule Abs there are two cases depending on the type of x. If $x : S$, we have $\Gamma; \Lambda_1; \Delta_1 \vdash \texttt{erase}(\lambda x : S.P) \triangleright S \multimap \diamond$, then by definition of $\texttt{erase}(\,)$, $\Gamma; \Lambda_1; \Delta_1 \vdash \lambda x.\texttt{erase}(P) \triangleright S \multimap \diamond$, thus by inductive hypothesis $\Gamma; \Theta_1, x : S \Vdash P; \Theta_2$, where $\Theta_2 = \emptyset$ or $\mathsf{End}(\Theta_1)$, and $\Theta_1, x : S = \Lambda_1, \Delta_1, x : S$, hence $\Theta_1 = \Lambda_1, \Lambda_2$. By A-SAbs we have $\Gamma; \Theta_1 \vdash \lambda x : S.P; \Theta_2 \div x$ and by definition of $\div$, either $\Theta_2 \div x = \emptyset$ or $\mathsf{End}(\Theta_2 \div x)$. When x is of a shared type the proof proceeds similar to the previous case.
- When the derivation ends with Rule Sel we have $\Gamma; \Lambda_1; \Delta_1, u : \oplus\{l_i : S_i\}_{i\in I} \vdash \texttt{erase}(u \triangleleft l_j.P) \triangleright \diamond = u \triangleleft l_j.\texttt{erase}(P) \triangleright \diamond$. By inductive hypothesis $\Gamma; \Theta_1, u : S_j \Vdash P; \Theta_2$ (1) for some $j \in I$ with $\Theta_2 = \emptyset$ or $\mathsf{End}(\Theta_2)$, and $\Theta_1, u : S_j = \Lambda_1, \Delta_1, u : S_j$, thus $\Theta_1 = \Lambda_1, \Delta_1$. It is clear that $\Gamma; \emptyset; u : \oplus\{l_i : S_i\}_{i\in I} \vdash u \triangleright \oplus\{l_i : S_i\}_{i\in I}$, hence by completeness of values and weakening $\Gamma; \Theta_1, u : \oplus\{l_i : S_i\}_{i\in I} \Vdash u \triangleright \oplus\{l_i : S_i\}_{i\in I}; \Theta_1$ (2), then applying A-Sel to (1) and (2) we obtain $\Gamma; \Theta_1, u : \oplus\{l_i : S_i\}_{i\in I} \Vdash u \triangleleft l_j.P; \Theta_2 \div u$. Definition of $\div$ says that either $\Theta_2 \div u = \emptyset$ or $\mathsf{End}(\Theta_4)$.
- When the derivation ends with Rule Acc the proof depends on the type of x. If $x : S$, we have $\Gamma; \Lambda_1; \Delta_1 \vdash \texttt{erase}(u?(x).P) \triangleright \diamond$; by induction we have $\Gamma; \emptyset \Vdash u \triangleright \langle S \rangle; \emptyset$ (1) and $\Gamma; \Theta_1, x : S \Vdash P; \Theta_2$ (2), where $\Theta_1, x : S = \Lambda_1, \Delta_1, x : S$, thus $\Theta_1 = \Lambda_1, \Delta_1$. By weakening on (1) we obtain $\Gamma; \Theta_1 \Vdash u \triangleright \langle U \rangle; \Theta_1$ (3). The proof concludes by applying rule A-SAcc and definition of $\div$. When x is typed at higher-order types the proof is follows the same intuition. When x is of a shared type, the proof proceeds similarly but applies A-ShAcc instead of A-SAcc.
- Suppose that the derivation ends with Rule Bra, then we have $\Gamma; \Lambda_2; \Delta_2, u : \&\{l_i : S_i\}_{i\in I} \vdash \texttt{erase}(u \triangleright \{l_i : P_i\})_{i\in I} \triangleright \diamond$, and by properties of $\texttt{erase}(\,)$, we have $\Gamma; \Lambda_2; \Delta_2, u : \&\{l_i : S_i\}_{i\in I} \vdash u \triangleright \{l_i : \texttt{erase}(P_i)\}_{i\in I} \triangleright \diamond$. By induction on each branch we have $\Gamma; \Theta_2, u : S_i \Vdash P_i; \Theta_3^i$ where $\Theta_2, u : S_j = \Lambda_2, \Delta_2, u : S_j$ and either $\Theta_3^i = \emptyset$ or $\mathsf{End}(\Theta_3^i)$ (1), however in each case $\texttt{NotEnd}(\Theta_3^i) = \texttt{NotEnd}(\Theta_3^j) = \emptyset$ for all $i, j \in I$. We have that $\Gamma; \emptyset; u : \&\{l_i : S_i\}_{i\in I} \vdash u \triangleright \&\{l_i : S_i\}_{i\in I}$, thus by completeness of values and weakening $\Gamma; \Theta_2, u : \&\{l_i : S_i\}_{i\in I} \Vdash u \triangleright \&\{l_i : S_i\}_{i\in I}; \Theta_2$. The proof concludes by applying Rule A-Bra to obtain $\Gamma; \Theta_1 \Vdash u \triangleright \{l_i : P_i\}_{i\in I}; \cap_{i\in I}\Theta_3^i \div u$. By (1) we know that $\cap_{i\in I}\Theta_3^i = \emptyset$ or $\mathsf{End}(\cap_{i\in I}\Theta_3^i)$, then by definition of $\div$, $\cap_{i\in I}\Theta_3^i \div u = \emptyset$ or $\mathsf{End}(\cap_{i\in I}\Theta_3^i \div u)$.
- When the derivation ends with rule Req, we have $\Gamma; \Lambda_2; \Delta_1, \Delta_2 \vdash \texttt{erase}(u!\langle V \rangle.P) \triangleright \diamond$. By induction on each branch we obtain $\Gamma; \emptyset \Vdash u \triangleright \langle U \rangle; \emptyset$, $\Gamma; \Theta_1 \Vdash V \triangleright U; \Theta_3$, and $\Gamma; \Theta_2 \Vdash P; \Theta_4$ where $\Theta_i = \emptyset$ or $\mathsf{End}(\Theta_i)$ for $i = 3, 4$, $\Theta_1 = \Delta_1$, and $\Theta_2 = \Lambda_2, \Delta_2$; it is clear that $\texttt{split}(\Theta_1) = \emptyset * \Theta_1$. The result follows from weakening u and V and then applying Rule A-Req.
- When the derivation ends with rule RVar we have $\Gamma, X : \Delta; \emptyset; \Delta \vdash X \triangleright \diamond$, then taking $\Theta = \Delta$ is clear that $\texttt{split}(\Theta) = \emptyset * \Theta$, thus we have $\Gamma, X : \Theta; \Theta \Vdash X; \emptyset$.
- If the derivation ends with Rule Rec, we have $\Gamma; \emptyset; \Delta \vdash \texttt{erase}(\mu X.P) \triangleright \diamond = \mu X.\texttt{erase}(P) \triangleright \diamond$, and by inductive hypothesis we obtain $\Gamma, X : \Theta; \Theta \Vdash P; \emptyset$, where $\Theta = \Delta$; it is clear that $\texttt{split}(\Theta) = \emptyset * \Theta$, thus by A-Req $\Gamma; \Theta \Vdash \mu(X : N).P; \emptyset$.
- When the derivation ends with Rule App then we have $\Gamma; \Lambda_1; \Delta_1, \Delta_2 \vdash V u \triangleright \diamond$, and by inductive hypothesis on each branch $\Gamma; \Theta_1 \Vdash V; \Theta_3$ where either $\Theta_3 = \emptyset$ or $\mathsf{End}(\Theta_3)$, and $\Theta_1 = \Lambda_1, \Delta_1$. $\Gamma; \Theta_2 \Vdash u; \Theta_4$ where either $\Theta_4 = \emptyset$ or $\mathsf{End}(\Theta_4)$, and $\Theta_2 = \Delta_2$ (1). If $\Theta_3 = \emptyset$ we weaken V, thus $\Gamma; \Theta_1, \Theta_2 \Vdash V; \Theta_2$ and $\texttt{split}(\Theta_2) = \emptyset * \Theta_2$, hence by applying Rule A-App $\Gamma; \Theta_1, \Theta_2 \Vdash V u; \Theta_4$. Note that $\Theta_1, \Theta_2 = \Lambda_1, \Delta_1, \Delta_2$. If $\mathsf{End}(\Theta_3)$ (2) we weaken both V and u to obtain $\Gamma; \Theta_1, \Theta_2 \Vdash V; \Theta_3, \Theta_2$ and $\Gamma; \Theta_3, \Theta_2 \Vdash u; \Theta_3, \Theta_4$. By (1) and (2) we have that $\texttt{split}(\Theta_3; \Theta_2) = \emptyset * \Theta_3, \Theta_2$, hence by applying Rule A-App we obtain the result.

The remaining cases follow by straightforward induction. □

The following lemma is used in the proof of soundness since it allows us to relate the output of the full process with the output of the last subprocess.

Lemma D.2.

- $\Theta \div u = \emptyset$ *then* $\Theta = \emptyset$ *or* $\Theta = u : \mathsf{end}$.
- $\mathsf{End}(\Theta \div u)$ *then* $\Theta = \Theta', u : \mathsf{end}$ *with* $\mathsf{End}(\Theta')$ *or* $u \notin \Theta$ *and* $\mathsf{End}(\Theta)$.

Proof. By definition of the function $\div$. □

The following is the soundness theorem, which corresponds to the converse of Theorem D.3. This theorem is necessary to complete the proof of Theorem 6.2.

Theorem D.4 *(Soundness). Given contexts Γ, Θ, a process D, a type T, and w (name or variable) we have the following:*

- *If* $\Gamma; \Theta \Vdash w \triangleright T; \emptyset$ *then* $\Gamma; \Lambda; \Delta; \vdash w \triangleright T$*, where* $\Lambda, \Delta = \Theta$.
- *If* $\Gamma; \Theta \Vdash D \triangleright T; \Theta_2$*, for* D *a process or an abstraction and* $\Theta_2 = \emptyset$ *or* $\mathsf{End}(\Theta_2)$*, then* $\Gamma; \Lambda; \Delta; \vdash \mathtt{erase}(D) \triangleright T$*, where* $\Lambda, \Delta = \Theta$.

Proof. The proof of soundness for names follows by a straightforward inspection of the algorithmic type checking rules and the definition of the function $\mathtt{erase}()$.

The proof for processes and abstractions follows by induction on the typing derivation.

- If the derivation ends with A-Send, we have $\Gamma; \Theta_1 \Vdash u!\langle V\rangle.P; \Theta_4 \div u$ with $\Theta_4 \div u = \emptyset$ or $\mathsf{End}(\Theta_4 \div u)$; by Lemma D.2 we know that either $\Theta_4 = \emptyset$ or $\mathsf{End}(\Theta_4)$ and then by induction $\Gamma; \Lambda_3; \Delta_3 \vdash \mathtt{erase}(P) \triangleright \diamond$ (1), where $\Lambda_3, \Delta_3 = \Theta_3$. We also have $\Gamma; \Theta_2, u : S \Vdash V \triangleright U; \Theta_3$, and by strengthening $\Gamma; (\Theta_2, u : S) \setminus \Theta_3 \Vdash V \triangleright U; \emptyset$. Inductive hypothesis on V says $\Gamma; \Lambda_2; \Delta_2 \vdash \mathtt{erase}(V) \triangleright U$ (2), where $\Lambda_2, \Delta_2 = (\Theta_2, u : S) \setminus \Theta_3$, then applying Send to (1) and (2), and by properties of $\mathtt{erase}()$ we obtain $\Gamma; \Lambda_2, \Lambda_3; ((\Delta_2, \Delta_3) \setminus u : S), u :!\langle U\rangle.S \vdash \mathtt{erase}(u!\langle V\rangle.P) \triangleright \diamond$
  Note that by $\Gamma; \Theta_1 \Vdash u \triangleright !\langle U\rangle.S; \Theta_2$ we know that $\Theta_1 = \Theta_1', u :!\langle U\rangle.S$ and $\Theta_1' = \Theta_2$. Note that

$$(\Theta_2, u : S) \setminus \Theta_3 = \Lambda_2, \Delta_2$$

$$((\Theta_2, u : S) \setminus \Theta_3), \Theta_3 = \Lambda_2, \Delta_2, \Lambda_3, \Delta_3$$

$$\Theta_2, u : S = \Lambda_2, \Delta_2, \Lambda_3, \Delta_3$$

$$\Theta_2, u : S = \Lambda_2, \Lambda_3, (\Delta_2, \Delta_3)$$

$$\Theta_2 = \Lambda_2, \Lambda_3, (\Delta_2, \Delta_3) \setminus u : S$$

  thus $\Theta_1 = \Theta_2, u :!\langle U\rangle.S = \Lambda_2, \Lambda_3, ((\Delta_2, \Delta_3) \setminus u : S), u :!\langle U\rangle.S$.
- If the derivation ends with A-SRcv, we have $\Gamma; \Theta_1 \Vdash u?(x).P; \Theta_3 \div \{x, u\}$ for some context Θ_3 and $\Theta_3 \div \{x, u\} = \emptyset$ or $\mathsf{End}(\Theta_3 \div \{x, u\})$; by Lemma D.2 and induction we obtain $\Gamma; \Lambda_2; \Delta_2, u : S, x : S' \vdash \mathtt{erase}(P) \triangleright \diamond$, where $\Lambda_2, \Delta_2, u : S, x : S' = \Theta_2, u : S, x : S'$, then $\Lambda_2, \Delta_2 = \Theta_2$, thus applying Rcv we obtain $\Gamma; \Lambda_2; \Delta_2, u :?(S').S \vdash \mathtt{erase}(u?(x).) \triangleright \diamond$ (1). By $\Gamma; \Theta_1 \Vdash u \triangleright ?(S').S; \Theta_2$ we know that $\Theta_1 = \Theta_1', u :?(S').S$ and $\Theta_1' = \Theta_2$, hence $\Theta_1 = \Lambda_2, \Delta_2, u :?(S').S$. The proof when $x : C \multimap \diamond$ is similar.
- When the derivation ends with Rule A-ShRcv, the proof is similar to A-SRcv.
- When the derivation ends with Rule A-Par suppose we have $\Gamma; \Theta_1 \Vdash P | Q; \Theta_3$ with $\Theta_3 = \emptyset$ or $\mathsf{End}(\Theta_3)$ then by induction we have $\Gamma; \Lambda_2; \Delta_2 \vdash \mathtt{erase}(Q) \triangleright \diamond$ where $\Lambda_2, \Delta_2 = \Theta_2$; applying strengthening to $\Gamma; \Theta_1 \Vdash P; \Theta_2$ and then induction we obtain $\Gamma; \Lambda_1; \Delta_1 \vdash \mathtt{erase}(P) \triangleright \diamond$, where $\Lambda_1, \Delta_1 = \Theta_1 \setminus \Theta_2$ then by Rule Par and properties of $\mathtt{erase}()$ we obtain $\Gamma; \Lambda_1, \Lambda_2; \Delta_1, \Delta_2 \vdash \mathtt{erase}(P | Q) \diamond$ and $\Lambda_1, \Lambda_2; \Delta_1, \Delta_2 = (\Theta_1 \setminus \Theta_2), \Theta_2 = \Theta_1$.
- When the derivation ends with Rule A-SAbs then we have $\Gamma; \Theta_1 \Vdash \lambda x : S.P; \Theta_2 \div x$, where $\Theta_2 \div x = \emptyset$ or $\mathsf{End}(\Theta_2 \div x)$; by Lemma D.2 we know $\Theta_2 = \emptyset$ or $\mathsf{End}(\Theta_2)$. Thus by induction $\Gamma; \Lambda_1; \Delta_1, x : S \vdash \mathtt{erase}(P)$ (1), where $\Lambda_1, \Delta_1, x : S = \Theta_1, x : S$, then by Rule Abs $\Gamma; \Lambda_1; \Delta_1 \vdash \lambda x.\mathtt{erase}(P)$, hence $\Gamma; \Lambda_1; \Delta_1 \vdash \mathtt{erase}(\lambda x.P)$.
- When the derivation ends with Rule A-ShAbs, the proof is similar to the one for Rule A-SAbs.
- When the derivation ends with the Rule A-Sel, we have $\Gamma; \Theta_1 \Vdash u \triangleleft l_j.P; \Theta_3 \div u$ with $\Theta_3 \div u = \emptyset$ or $\mathsf{End}(\Theta_3 \div u)$, and by Lemma D.2 and inductive hypothesis $\Gamma; \Lambda_2; \Delta_2, u : S_j \vdash \mathtt{erase}(P) \triangleright \diamond$, where $\Lambda_2, \Delta_2, u : S_j = \Theta_2, u : S_j$. Thus by applying Rule Sel we obtain $\Gamma; \Lambda_2; \Delta_2, u : \oplus\{l_i : S_i\}_{i \in I} \vdash \mathtt{erase}(u \triangleleft l_j.P) \triangleright \diamond$. By $\Gamma; \Theta_1 \Vdash u \triangleright \oplus\{l_i : S_i\}; \Theta_2$ we know $\Theta_1 = \Theta_1', u : \oplus\{l_i : S_i\}$, $\Theta_2 = \Theta_1'$, hence $\Theta_1 = \Lambda_2, \Delta_2, u : \oplus\{l_i : S_i\}$.
- When the derivation ends with Rule A-Bra we have $\Gamma; \Theta_1 \Vdash u \triangleright \{l_i : P_i\}_{i \in I}; \cap_{i \in I} \Theta_3^i \div u$ where $\cap_{i \in I} \Theta_3^i \div u = \emptyset$ or $\mathsf{End}(\cap_{i \in I} \Theta_3^i \div u)$.
  - in the first case $\cap_{i \in I} \Theta_3^i = \emptyset$ or $\cap_{i \in I} \Theta_3^i = u : \mathsf{end}$. Since $\forall i, j \in I$ $\mathtt{NotEnd}(\Theta_3^i) = \mathtt{NotEnd}(\Theta_3^j)$, $\cap_{i \in I} \Theta_3^i = \emptyset$ implies that $\forall i \in I$ $\mathtt{NotEnd}(\Theta_3^i) = \emptyset$. The same reasoning is applied when $\cap_{i \in I} \Theta_3^i = u : \mathsf{end}$.
  - If $\mathsf{End}(\cap_{i \in I} \Theta_3^i \div u)$ we know that either $\cap_{i \in I} \Theta_3^i = \Theta, u : \mathsf{end}$ for some Θ and $\mathsf{End}(\Theta)$ or $u \notin dom(\cap_{i \in I} \Theta_3^i)$ and $\mathsf{End}(\cap_{i \in I} \Theta_3^i)$. In any case $\forall i \in I$ $\mathtt{NotEnd}(\Theta_3^i) = \emptyset$.

  thus then (by induction on every branch) $\Gamma; \Lambda_2; \Delta_2, u : S_i \vdash \mathtt{erase}(P_i) \triangleright \diamond$, where $\Lambda_2, \Delta_2, u : S_i = \Theta_2, u : S_i$ then we apply Rule Bra to obtain $\Gamma; \Lambda_2; \Delta_2, u : \&\{l_i : S_i\}_{i \in I} \vdash u \triangleright \{l_i : \mathtt{erase}(P_i)\}$. By $\Gamma; \Theta_1 \Vdash \&\{l_i : S_i\}_{i \in I}; \Theta_2$ we know $\Theta_1 = \Theta_1', u : \&\{l_i : S_i\}_{i \in I}$ and $\Theta_2 = \Theta_1'$, hence $\Theta_1 = \Lambda_2, \Delta_2, u : \&\{l_i : S_i\}_{i \in I}$.
- When the derivation ends with Rule A-SAcc, then we have $\Gamma; \Theta_1 \Vdash u?(x).P; \Theta_2 \div x$ where $\Theta_2 \div x = \emptyset$ or $\mathsf{End}(\Theta_2 \div x)$, and by Lemma D.2 $\Theta_2 = \emptyset$ or $\mathsf{End}(\Theta_2)$, then by inductive hypothesis $\Gamma; \Lambda_1; \Delta_1, x : S \vdash \mathtt{erase}(P) \triangleright \diamond$, where $\Lambda_1, \Delta_1, x : S = \Theta_1, x : S$ and θ is a Session type. Strengthening on u gives $\Gamma; \emptyset \Vdash u \triangleright \langle S\rangle; \emptyset$. Induction tells us that $\Gamma; \emptyset; \emptyset \vdash u \triangleright \langle \theta\rangle$, then result follows by applying Rule Acc.
- If the derivation ends with Rule A-ShAcc, the proof is similar to the previous case.
- When the derivation ends with Rule A-RVar we have $\Gamma, X : \Theta_1; \Theta_1, \Theta_2 \Vdash X; \Theta_2$ where $\mathtt{split}(\Theta_1) = \emptyset * \Theta_1$ and either $\Theta_2 = \emptyset$ or $\mathsf{End}(\Theta_2)$. If $\Theta_2 = \emptyset$ we take $\Delta = \Theta_1$ and then we have $\Gamma, X : \Delta; \emptyset; \Delta \vdash X \triangleright \diamond$. If $\mathsf{End}(\Theta_2)$, we apply strengthening and then we take $\Delta = \Theta_1$.
- When the derivation ends with Rule A-Rec we have $\Gamma; \Theta_1, \Theta_2 \Vdash \mu(X : N).P; \Theta_2$, where $\Theta_2 = \emptyset$ or $\mathsf{End}(\Theta_2)$. If $\Theta_2 = \emptyset$ then $\Gamma; \Theta_1 \Vdash \mu(X : N).P; \emptyset$ where $\mathtt{split}(\Theta_1) = \emptyset * \Theta_1$, hence by inductive hypothesis $\Gamma, X : \Delta_1; \emptyset; \Delta_1 \vdash \mathtt{erase}(P) \triangleright \diamond$

where $\Delta_1 = \Theta_1$, then by Rule Rec $\Gamma; \emptyset; \Delta_1 \vdash \mu X.\texttt{erase}(P) \triangleright \diamond = \texttt{erase}(\mu X.P) \triangleright \diamond$. If $\mathsf{End}(\Theta_2)$, then we first strengthen the derivation and then follow the previous reasoning.
- When the derivation ends with Rule A-Req, then we have $\Gamma; \Theta_1 \Vdash u!\langle V \rangle.P; \Theta_5$ where $\Theta_5 = \emptyset$ or $\mathsf{End}(\Theta_5)$. By inductive hypothesis we have $\Gamma; \Lambda_2; \Delta_2 \vdash \texttt{erase}(P) \triangleright \diamond$ (1), where $\Lambda_2, \Delta_2 = \Theta_2, \Theta_4$ and $\texttt{split}(\Theta_1) = \Theta_2 * \Theta_3$. By strengthening on both $\Gamma; \Theta_1 \Vdash u \triangleright \langle U \rangle; \Theta_1$ and $\Gamma; \Theta_3 \Vdash V \triangleright U; \Theta_4$ and then by inductive hypothesis we have both $\Gamma; \emptyset; \emptyset \vdash u \triangleright \langle U \rangle$ (2) and $\Gamma; \emptyset; \Delta_1 \vdash \texttt{erase}(V) \triangleright U$ (3), where $\Delta_1 = \Theta_3 \setminus \Theta_4$. Applying Req we obtain $\Gamma; \Lambda_2; \Delta_1, \Delta_2 \vdash \texttt{erase}(u!\langle V \rangle.P)$. Note that $\Lambda_2, \Delta_1, \Delta_2 = (\Theta_3 \setminus \Theta_4), \Theta_2, \Theta_4 = \Theta_2, \Theta_3 = \Theta_1$.
- When the derivation ends with Rule A-App, then $\Gamma; \Theta_1 \Vdash V u; \Theta_3, \Theta_5$ where either $\Theta_3, \Theta_5 = \emptyset$ or $\mathsf{End}(\Theta_3, \Theta_5)$. If $\Theta_3, \Theta_5 = \emptyset$ then $\texttt{split}(\Theta_2) = \emptyset * \Theta_4$ (1), and then by inductive hypothesis $\Gamma; \emptyset; \Delta_2 \vdash u \triangleright C$ where $\Delta_2 = \Theta_4$. Strengthening V and induction produces $\Gamma; \Lambda_1; \Delta_1 \vdash V \triangleright C \rightsquigarrow \diamond$ where $\Theta_1 \setminus \Theta_2 = \Lambda_1, \Delta_1$, thus by App $\Gamma; \Lambda_1; \Delta_1, \Delta_2 \vdash V u \triangleright \diamond$. By (1) is clear that $\Theta_2 = \Theta_4$, thus $\Lambda_1, \Delta_1, \Delta_2 = (\Theta_1 \setminus \Theta_2), \Theta_4 = (\Theta_1 \setminus \Theta_2), \Theta_2 = \Theta_1$. When $\mathsf{End}(\Theta_3, \Theta_5)$ we strengthen u and then the reasoning is the same as before.

The remaining cases follow by straightforward induction. □

References

[1] Massimo Bartoletti, Maurizio Murgia, Alceste Scalas, Roberto Zunino, Modelling and verifying contract-oriented systems in Maude, in: Santiago Escobar (Ed.), Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 8663, Springer, 2014, pp. 130–146.

[2] Massimo Bartoletti, Maurizio Murgia, Alceste Scalas, Roberto Zunino, Verifiable abstractions for contract-oriented systems, J. Log. Algebraic Methods Program. 86 (1) (2017) 159–207.

[3] Marco Carbone, Søren Debois, A graphical approach to progress for structured communication in web services, in: Simon Bliudze, Roberto Bruni, Davide Grohmann, Alexandra Silva (Eds.), Proceedings Third Interaction and Concurrency Experience: Guaranteed Interaction, ICE 2010, Amsterdam, the Netherlands, 10th of June 2010, in: EPTCS, vol. 38, 2010, pp. 13–27.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Carolyn Talcott, All About Maude - a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic, Springer-Verlag, Berlin, Heidelberg, 2007.

[5] Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Luca Padovani, Global progress for dynamically interleaved multiparty sessions, Math. Struct. Comput. Sci. 26 (2) (2016) 238–302.

[6] E.M. Clarke, O. Grumberg, D. Kroening, D. Peled, H. Veith, Model Checking, second edition, Cyber Physical Systems Series, MIT Press, 2018.

[7] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, Roderick Bloem, Handbook of Model Checking, 1st edition, Springer Publishing Company, Incorporated, 2018.

[8] Luís Caires, Frank Pfenning, Bernardo Toninho, Linear logic propositions as session types, Math. Struct. Comput. Sci. 26 (3) (2016) 367–423.

[9] Ornela Dardha, Simon J. Gay, A new linear logic for deadlock-free session-typed processes, in: Christel Baier, Ugo Dal Lago (Eds.), Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, in: Lecture Notes in Computer Science, vol. 10803, Springer, 2018, pp. 91–109.

[10] Steven Eker, José Meseguer, Ambarish Sridharanarayanan, The Maude LTL model checker, in: Fabio Gadducci, Ugo Montanari (Eds.), Fourth International Workshop on Rewriting Logic and Its Applications, WRLA2002, Pisa, Italy, 19-21, 2002, in: Electronic Notes in Theoretical Computer Science, vol. 71, Elsevier, 2002, pp. 162–187.

[11] Simon J. Gay, Subtyping supports safe session substitution, in: Sam Lindley, Conor McBride, Philip W. Trinder, Donald Sannella (Eds.), A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, in: Lecture Notes in Computer Science, vol. 9600, Springer, 2016, pp. 95–108.

[12] Simon J. Gay, Malcolm Hole, Subtyping for session types in the pi calculus, Acta Inform. 42 (2–3) (2005) 191–225.

[13] Simon J. Gay, Peter Thiemann, Vasco T. Vasconcelos, Duality of session types: the final cut, Electron. Proc. Theor. Comput. Sci. 314 (apr 2020) 23–33.

[14] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, Gianluigi Zavattaro, Foundations of session types and behavioural contracts, ACM Comput. Surv. 49 (1) (2016) 3:1–3:36.

[15] Kohei Honda, Vasco T. Vasconcelos, Makoto Kubo, Language primitives and type disciplines for structured communication-based programming, in: ESOP'98, vol. 1381, Springer, 1998, pp. 22–138.

[16] Keigo Imai, Nobuko Yoshida, Shoji Yuen Session-ocaml, A session-based library with polarities and lenses, in: Jean-Marie Jacquet, Mieke Massink (Eds.), Coordination Models and Languages - 19th IFIP WG 6.1 International Conference, COORDINATION 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10319, Springer, 2017, pp. 99–118.

[17] Dimitrios Kouzapas, Jorge A. Pérez, Nobuko Yoshida, Characteristic bisimulation for higher-order session processes, Acta Inform. 54 (3) (2017) 271–341.

[18] Dimitrios Kouzapas, Jorge A. Pérez, Nobuko Yoshida, On the relative expressiveness of higher-order session processes, Inf. Comput. 268 (2019).

[19] Robin Milner, Joachim Parrow, David Walker, A calculus of mobile processes, I, Inf. Comput. 100 (1) (1992) 1–40.

[20] Luca Padovani, Deadlock and lock freedom in the linear π-calculus, in: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Association for Computing Machinery, New York, NY, USA, 2014.

[21] Luca Padovani, A simple library implementation of binary sessions, J. Funct. Program. 27 (e4) (2017).

[22] Georgios V. Pitsiladis, Petros S. Stefaneas, Implementation of privacy calculus and its type checking in Maude, in: Tiziana Margaria, Bernhard Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II, in: Lecture Notes in Computer Science, vol. 11245, Springer, 2018, pp. 477–493.

[23] Carlos Alberto Ramírez Restrepo, Jorge A. Pérez, Executable semantics and type checking for session-based concurrency in Maude, in: Kyungmin Bae (Ed.), Rewriting Logic and Its Applications, 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2–3, 2022, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 13252, Springer, 2022, pp. 230–250.

[24] Mark-Oliver Stehr, José Meseguer, Peter Csaba Ölveczky, Rewriting logic as a unifying framework for Petri nets, in: Hartmut Ehrig, Gabriel Juhás, Julia Padberg, Grzegorz Rozenberg (Eds.), Unifying Petri Nets, Advances in Petri Nets, in: Lecture Notes in Computer Science, vol. 2128, Springer, 2001, pp. 250–303.

[25] Mark-Oliver Stehr, CINNI - a generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi, in: Kokichi Futatsugi (Ed.), The 3rd International Workshop on Rewriting Logic and Its Applications, WRLA 2000, Kanzawa, Japan, September 18-20, 2000, in: Electronic Notes in Theoretical Computer Science, vol. 36, Elsevier, 2000, pp. 70–92.
[26] D. Sangiorgi, D. Walker, The Pi-Calculus: A Theory of Mobile Processes, Cambridge University Press, 2003.
[27] Alceste Scalas, Nobuko Yoshida, Less is more: multiparty session types revisited, Proc. ACM Program. Lang. 30 (POPL) (2019) 30:1–30:29.
[28] Prasanna Thati, Koushik Sen, Narciso Martí-Oliet, An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0, Electron. Notes Theor. Comput. Sci. 71 (2002) 261–281.
[29] Bernardo Toninho, Nobuko Yoshida, Interconnectability of session-based logical processes, ACM Trans. Program. Lang. Syst. 40 (4) (2018) 17:1–17:42.
[30] Vasco T. Vasconcelos, Fundamentals of session types, Inf. Comput. 217 (2012) 52–70.
[31] Bas van den Heuvel, Jorge A. Pérez, A decentralized analysis of multiparty protocols, Sci. Comput. Program. 222 (2022) 102840.
[32] Patrick Viry, Input/output for ELAN, in: José Meseguer (Ed.), First International Workshop on Rewriting Logic and Its Applications, RWLW 1996, Asilomar Conference Center, Pacific Grove, CA, USA, September 3-6, 1996, in: Electronic Notes in Theoretical Computer Science, vol. 4, Elsevier, 1996, pp. 51–64.
[33] Patrick Viry, A rewriting implementation of pi-calculus, Technical report, University of Pisa, 1996.
[34] Alberto Verdejo, Narciso Martí-Oliet, Implementing CCS in Maude 2, in: Fabio Gadducci, Ugo Montanari (Eds.), Fourth International Workshop on Rewriting Logic and Its Applications, WRLA2002, Pisa, Italy, 19-21, 2002, in: Electronic Notes in Theoretical Computer Science, vol. 71, Elsevier, 2002, pp. 282–300.
[35] Philip Wadler, Propositions as sessions, J. Funct. Program. 24 (2–3) (2014) 384–418.